ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS

ALASDAIR ARMSTRONG, University of Cambridge, UK
THOMAS BAUEREISS, University of Cambridge, UK
BRIAN CAMPBELL, University of Edinburgh, UK
ALASTAIR REID, ARM Ltd., UK
KATHRYN E. GRAY, University of Cambridge (Formerly), UK
ROBERT M. NORTON, University of Cambridge, UK
PRASHANTH MUNDKUR, SRI International, US
MARK WASSELL, University of Cambridge, UK
JON FRENCH, University of Cambridge, UK
CHRISTOPHER PULTE, University of Cambridge, UK
SHAKED FLUR, University of Cambridge, UK
IAN STARK, University of Edinburgh, UK
NEEL KRISHNASWAMI, University of Cambridge, UK
PETER SEWELL, University of Cambridge, UK

Architecture specifications notionally define the fundamental interface between hardware and software: the envelope of allowed behaviour for processor implementations, and the basic assumptions for software development and verification. But in practice, they are typically prose and pseudocode documents, not rigorous or executable artifacts, leaving software and verification on shaky ground.

In this paper, we present rigorous semantic models for the sequential behaviour of large parts of the mainstream ARMv8-A, RISC-V, and MIPS architectures, and the research CHERI-MIPS architecture, that are complete enough to boot operating systems, variously Linux, FreeBSD, or seL4. Our ARMv8-A models are automatically translated from authoritative ARM-internal definitions, and (in one variant) tested against the ARM Architecture Validation Suite.

We do this using a custom language for ISA semantics, Sail, with a lightweight dependent type system, that supports automatic generation of emulator code in C and OCaml, and automatic generation of proof-assistant definitions for Isabelle, HOL4, and (currently only for MIPS) Coq. We use the former for validation, and to assess specification coverage. To demonstrate the usability of the latter, we prove (in Isabelle) correctness of a purely functional characterisation of ARMv8-A address translation. We moreover integrate the RISC-V model into the RMEM tool for (user-mode) relaxed-memory concurrency exploration. We prove (on paper) the soundness of the core Sail type system.

We thereby take a big step towards making the architectural abstraction actually well-defined, establishing foundations for verification and reasoning.

CCS Concepts: • **General and reference → Verification**; • **Theory of computation → Semantics and reasoning**; • **Computer systems organization → Architectures**; • **Software and its engineering → Assembly languages**;

Additional Key Words and Phrases: Instruction Set Architectures, Semantics, Theorem Proving

2475-1421/2019/1-ART71
https://doi.org/10.1145/3290384

ACM Reference Format:
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. *Proc. ACM Program. Lang.* 3, POPL, Article 71 (January 2019), 31 pages. https://doi.org/10.1145/3290384

1 INTRODUCTION

The architectural abstraction is a fundamental interface in computing: the architecture specification for each family of processors, ARMv8-A, AMD64, IBM POWER, Intel 64, MIPS, RISC-V, SPARC, etc., notionally defines the envelope of allowed behaviour for all hardware processor implementations of that family, providing the basic assumptions for portable software development. This decouples hardware and software implementation, as architectures are relatively stable over time, while processor implementations evolve rapidly.

In practice, industry architecture specifications have traditionally been prose documents, with decoding tables and (at best) pseudocode descriptions of instruction behaviour, while vendors have maintained internal "golden" reference models, often as large and highly confidential C++ codebases. The mainstream architectures have accumulated enormous complexity: 6300 and 4700 pages for recent ARMv8-A and Intel 64/IA-32 specification documents [ARM 2017; Intel Corporation 2017]. They comprise two main parts: the Instruction Set Architecture (ISA), describing the behaviour of each instruction in isolation, and cross-cutting aspects such as the concurrency model and interrupt behaviour. Understanding all these details is essential for achieving correct and robust behaviour of computer systems, but prose and pseudocode are simply not up to the task of precisely specifying them. These specification documents are moreover not executable as test oracles —they do not allow one to compute the set of all architecturally allowed behaviour of hardware tests, or to test software above the entire architectural envelope rather than just some specific implementation— and they do not support automatic test generation or test-suite specification coverage measurement.

Meanwhile, academic researchers in programming languages, semantics, analysis, and verification have increasingly aimed at mechanised reasoning about correctness down to the machine level, e.g. in the CakeML [Fox et al. 2017; Kumar et al. 2014; Tan et al. 2016], CerCo [Amadio et al. 2013], CompCert [Leroy 2009; Leroy et al. 2017], and CompCertTSO [Ševčík et al. 2013] verified compilers; the seL4 [Fox and Myreen 2010; Klein et al. 2014] and Hyper-V [Leinenbach and Santen 2009] verified hypervisors; the Verified Software Toolchain [Appel et al. 2017]; CertiKOS verified OS [Gu et al. 2016]; Verasco verified static analysis [Jourdan et al. 2015]; RockSalt software fault isolation system [Morrisett et al. 2012]; Bedrock [Chlipala 2013]; PROSPER [Baumann et al. 2016; Guanciale et al. 2016]; machine-code program logics [Jensen et al. 2013; Kennedy et al. 2013; Myreen 2009]; and relaxed-memory semantics [Alglave et al. 2010, 2014; Flur et al. 2017; Gray et al. 2015; Pulte et al. 2018; Sarkar et al. 2011; Sewell et al. 2010]. Binary analysis tools such as Angr [Shoshitaishvili et al. 2016], BAP [Brumley et al. 2011], TSL [Lim and Reps 2013], and Valgrind [Nethercote and Seward 2007] also need architectural models, although typically less formally expressed.

On what semantics should such work be based? Recoiling, reasonably enough, from the scale of the full 6000+ page vendor architecture documents, and from the poorly specified complexities of the concurrency models and privileged "system-mode" aspects of the architectures (virtual memory, exceptions, interrupts, security domain transitions, etc.), many groups have hand-written formal models of modest ISA fragments. These typically cover just enough of the instruction set, and in just enough detail, for their purpose: usually only some aspects of the sequential behaviour of parts of the non-privileged "user-mode" ISA, and just for one proof assistant (Coq, HOL4, or Isabelle). Some are validated against actual hardware behaviour, to varying degrees, but none are tied to a vendor reference model. The multiplicity of models, each produced by a different group for their

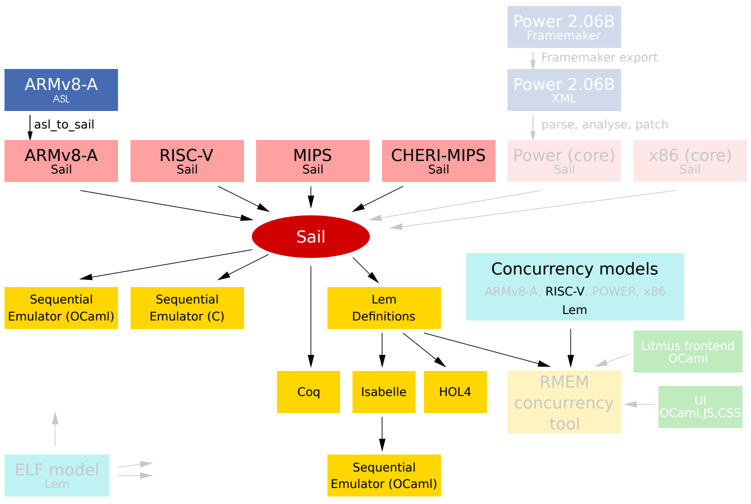

Fig. 1. Sail ISA semantics and (in yellow) the generated prover and emulator versions. The grey parts are previous concurrency and ISA models, user-mode only and not yet fully integrated into current Sail

specific purpose, is inefficient and makes it hard to amortise any validation investment. A few go beyond user-mode fragments, including seL4, PROSPER, and the ACL2 X86isa model [Goel et al. 2017]; we return to these, and other related work, in §9. Emulators such as QEMU [qem 2017] and gem5 [gem 2017] effectively also develop models, often rather complete, but these are optimised for performance and hard to use for other purposes.

In this paper, we present rigorous semantic models for the sequential behaviour of large parts of the mainstream ARMv8-A, RISC-V, and MIPS architectures, and the research CHERI-MIPS architecture, that are complete enough to boot various operating systems: Linux above the ARMv8-A model, FreeBSD above MIPS and CHERI-MIPS, and seL4 and Linux above RISC-V. These are rather large semantics by usual academic standards: approximately 23 000 lines for ARMv8-A, and a few thousand for each of the others.

ARMv8-A is the ARM application-processor architecture, specifying the processors, designed by ARM and by their architecture partners, and produced by many vendors, that are ubiquitous in mobile devices. We build on a shift within ARM over recent years to specify ISA behaviour in an ARM-internal machine-processed language, ASL. We work with two versions: a recent public release of large parts of this for ARMv8.3 [Reid 2016, 2017; Reid et al. 2016], and a currently non-public more complete version thereof; our ARM models are automatically translated from these. We moreover validate the second by testing against the ARM-internal Architecture Validation Suite. These are thus substantially more complete, authoritative, and well-validated than previous models. For RISC-V and CHERI-MIPS, the situation is rather different: these are much simpler architectures, and they are in flux, currently being designed. Our models for these (and our MIPS model underlying CHERI-MIPS) are handwritten, feeding back into the architecture design process, and validated in part by comparison with previous simulator and formal models.

To be generally useful, our models should simultaneously:

(1) be accessible to practising engineers who use existing vendor pseudocode descriptions;
(2) be automatically translatable into executable sequential emulator code, with reasonable performance, to support validation of the models and software development above them;

(3) be automatically translatable into idiomatic theorem prover definitions, to support formal mechanised reasoning about the architectures and about code above them — ideally for all the major provers, to enable use by each prover community;
(4) provide bidirectional mappings between assembly syntax and binary opcodes;
(5) provide the fine-grained execution information needed to integrate ISA semantics with the (user-mode) architectural relaxed-memory concurrency semantics previously developed;
(6) be well-validated, to give confidence that they do capture the architectural intent and soundly describe hardware behaviour; and
(7) be expressed in a well-engineered and robust infrastructure.

We achieve all this via a custom language for ISA semantics called Sail (§3). A previous version of this language has been used to represent modest user-mode ISA fragments for integration with concurrency models [Flur et al. 2016; Gray et al. 2015]. However, we found that it did not scale up to the use-cases and full-scale models we present in this paper. In particular, its type system, which relied on an ad-hoc constraint solver and coercion insertion, could not handle the full ARMv8-A specification. Moreover, it did not provide generation of prover definitions or emulators. In this paper, we extend Sail with the automatic translations depicted in Fig 1: from the ARM-internal ASL language into Sail, from Sail to C and OCaml emulator code (§5), and from Sail to Isabelle/HOL, HOL4, and Coq theorem-prover definitions (§4). This common infrastructure for all our architecture models saves much duplication of effort.

Moreover, we present a significant redesign and reimplementation of key parts of the Sail language itself in order to reconcile all of the above disparate goals. This is a delicate language-design problem: On the one hand, Sail has to be expressive enough to support each model idiomatically, especially the most-demanding ARMv8-A case, where the ASL source has accumulated features over time, including exceptions and complex (but not fully checked) dependent types for bitvector lengths. On the other hand, it has to be as *inexpressive* as possible in order to make Sail translatable into all the targets, in particular the non-dependently-typed provers Isabelle/HOL and HOL4, as well as the C and OCaml languages for emulator generation. We resolve this with a carefully designed lightweight dependent type system for checking vector bounds and integer ranges, inspired by Liquid types [Rondon et al. 2008], but which can be formalised in a simple, syntax-directed and single-pass style using a bidirectional approach [Dunfield and Krishnaswami 2013]. We increase confidence in the Sail type system with a (paper) formalisation and soundness proof of a core MiniSail (§6). All constraints can be shown to exist within a decidable fragment, and are resolved using the Z3 SMT solver [De Moura and Bjørner 2008]. Our translations to Isabelle/HOL, HOL4, C, and OCaml rely on *monomorphising* these dependent types where they are not target-expressible, allowing us to use the existing well-developed machine word libraries for the first two, and efficient representations for the last two.

Otherwise, Sail is essentially a first-order functional/imperative language with a simple effect system, but with abstract register and memory accesses, for sequential and concurrent interpretations. Higher-order functions are unnecessary for our ISA models and would complicate the translations to efficient C emulator code.

We validate our models with the OS boots and ARM Architectural Validation Suite mentioned above, and with other test suites, using the executable OCaml and C versions produced by Sail (§7). This also lets us assess the *specification coverage* of such OS boot executions and test suites. We also validate the RISC-V model behaviour on concurrency litmus tests using RMEM.

We evaluate the usability of our generated theorem prover definitions by conducting an example proof in Isabelle/HOL about one of the most complex parts of the ARMv8-A specification, the translation from virtual to physical memory addresses. We prove correctness of a simple purely functional characterisation of address translation, under suitable preconditions (§8).

Considered as a specification or programming language, Sail is unusual in that it aims to support just a handful of specific programs — these and other architecture definitions of mainstream and research architectures — but the importance of those makes it necessary to do so well, and the specification scale and multiple demands listed above make that challenging.

Sail, along with our public ARMv8-A, RISC-V, MIPS, and CHERI-MIPS models, is publicly available under an open-source licence (https://github.com/rems-project/sail), and with an OPAM package for Sail. The version of our ARMv8-A model derived from a non-public ARM source is currently not available, but we hope that will be possible in due course, and some of the legal infrastructure needed is in place.

Contributions. In summary, this paper makes the following contributions.

- We present large well-validated ISA models for RISC-V (§2.1), CHERI-MIPS (§2.2), and ARMv8-A (§2.3). By translating the vendor-supplied ARMv8-A specifications from ASL into Sail, we bring them into a usable form, as ASL itself does not have a publicly available implementation. All models presented in this paper include system features sufficient to boot operating systems, and they have been validated against various test suites (§7).
- We substantially redesign and reimplement key parts of the Sail language (§3), including the type system, balancing the expressivity required by the models and the simplicity required by the backends. We formalise a core of Sail's type system and prove its soundness (§6).
- We provide automatic translations from Sail specifications to theorem prover definitions, for multiple provers (§4). We demonstrate their usability in Isabelle/HOL by conducting a mechanised proof about virtual memory address translation in ARMv8-A (§8).
- We provide automatic generation of emulators from Sail specifications that perform well enough to boot operating systems (§5).

Caveats and limitations. Our models cover considerably more than most formal ISA semantics of previous work, but they are still far from complete definitions of these architectures. For ARMv8-A, we translate only the AArch64 64-bit part of the architecture, not the AArch32 32-bit instructions. Including these should need only modest additional work. Our Coq generation has so far only been exercised for MIPS. Our assembly syntax support has only been exercised for RISC-V; for ARM it should be possible to generate this from ARM-supplied metadata, but that has not yet been done.

More substantially, we focus here on sequential behaviour. For RISC-V, our ISA model is integrated with the corresponding user-mode relaxed memory model, but we have not yet done that for ARM, and the relaxed-memory semantics of systems features (virtual memory, interrupts, etc.) is an open problem. Previous versions of Sail included models for modest fragments of the user-mode ISAs of IBM POWER [Gray et al. 2015], ARMv8 [Flur et al. 2016], and RISC-V and x86 (both previously unpublished); sufficient only for litmus tests and some user-mode concurrent algorithms. Those IBM POWER and x86 models have not yet been ported to the revised Sail of this paper, and that ARM model will be superseded by the one we present here when the above integration is done.

2 MODELS

The current status of our models and the generated definitions is summarised in Fig. 2.

2.1 RISC-V

Most ISAs have been proprietary. In contrast, RISC-V is an open ISA, currently under development by a broad industrial and academic community, coordinated by the RISC-V Foundation. It is subdivided

Architecture	Source	Size (LOS)	Boots	Generates		
ARMv8.3-A (public)	ARM ASL	23 000		C, OCaml	Isabelle, HOL4	
ARMv8.3-A (private)	ARM ASL	30 000	Linux	C, OCaml		
RISC-V	hand-written	5 000	seL4, Linux	C, OCaml	Isabelle, HOL4	RMEM
MIPS	hand-written	2 000	FreeBSD	C, OCaml	Isabelle, HOL4, Coq	
CHERI-MIPS	hand-written	4 000	FreeBSD	C, OCaml	Isabelle, HOL4	

Fig. 2. Status of Sail models

```
union clause ast = LOAD : (bits(12), regbits, regbits)

mapping clause encdec = LOAD(imm, rs1, rd, is_unsigned, size, false, false)
  <-> imm @ rs1 @ bool_bits(is_unsigned) @ size_bits(size) @ rd @ 0b0000011

function clause execute(LOAD(imm, rs1, rd, is_unsigned, width, aq, rl)) =
   let vaddr : xlenbits = X(rs1) + EXTS(imm) in
   if check_misaligned(vaddr, width)
   then { handle_mem_exception(vaddr, E_Load_Addr_Align); false }
   else match translateAddr(vaddr, Read, Data) {
     TR_Failure(e) => { handle_mem_exception(vaddr, e); false },
     TR_Address(addr) =>
       match width {
         BYTE => process_load(rd, vaddr, mem_read(addr, 1, aq, rl, false), is_unsigned),
         HALF => process_load(rd, vaddr, mem_read(addr, 2, aq, rl, false), is_unsigned),
         WORD => process_load(rd, vaddr, mem_read(addr, 4, aq, rl, false), is_unsigned),
         DOUBLE => process_load(rd, vaddr, mem_read(addr, 8, aq, rl, false), is_unsigned)
       }
   }
```

Fig. 3. RISC-V load instruction in Sail

into a core and many separable features. We have handwritten a RISC-V ISA model based on recent versions of the prose RISC-V specifications [RIS 2017]. Our current model implements the 64-bit (RV64) version of the ISA: the `rv64imac` dialect (integer, multiply-divide, atomic, and compressed instructions), with user, machine, and supervisor modes, and the Sv39 address translation mode (3-level page tables covering 512GiB of virtual address space).

The model is partitioned into separate files for user-space definitions, machine- and supervisor-mode parts, the physical memory interface, virtual memory and address translation, instruction definitions, and the fetch-execute-interrupt loop. The main omissions are floating-point, PMP (Physical Memory Protection), modularisation for the “unified” 32-bit/64-bit model, and factoring to build machine/user and machine-only variants.

For example, Fig. 3 shows the Sail code defining the RISC-V `LOAD` instructions: a constructor of the `ast` Sail type, a clause of the `encdec` function (mapping between a 32-bit instruction word and the corresponding `ast` value containing the opcode fields), and a clause of the `execute` function expressing its dynamic semantics. The body of that is imperative code: `X(...)` refers to the RISC-V general-purpose registers, `mem_read` is a function that performs a read of physical memory, and `process_load` handles potential access exceptions. The boolean return value of the `execute` clause indicates whether the instruction retired successfully, and is used to update the `minstret` CSR register. The `aq` and `rl` flags are used to indicate the ordering constraints of the load to the memory model. Modulo minor syntactic variations, this should be readable by anyone familiar with typical industry ISA pseudocode descriptions.

To get a sense of what is required to make an ISA semantics complete enough to boot an OS, rather than a user-mode fragment, we describe some of what we have had to do. This model is parameterisable over various platform implementation choices that the ISA allows. In particular, it supports (i) trapping as well as non-trapping modes of accesses to misaligned data addresses, and (ii) write updates as well as traps when a dirty-bit needs to be updated in a page-table entry during address translation. RISC-V also specifies various control and status registers (CSRs) as having bitfields with platform-defined behaviour on reads and writes, which allows a platform to choose legal values of a CSR bitfield, and how it handles writes to those fields. Our model supports these choices through user-specifiable *legaliser* functions that intercept read and write accesses to those CSRs that require such behaviour.

We have also endeavoured to keep other platform aspects explicitly separate from the Sail model. For example, the reservation state for Load-Reserved/Store-Conditional instructions is kept as part of the platform state, since the reservation state and progress guarantees provided are inherently platform-specific. This separation also simplifies reasoning about the RISC-V memory model.

The physical memory map for a platform is specified using the `extern` facility of the Sail language, which enables the ISA model itself to remain agnostic of the actual map, but allows the contexts of the various backend renderings of the model to provide these definitions. For example, the generated OCaml executable model is linked against modules that define the locations of valid physical memory regions, valid memory-mapped I/O regions, and the location of the timer and terminal devices. These modules also place the corresponding Device-Tree information generated from these values at the expected location in physical memory when the OCaml ISA emulator is initialised. The ISA model itself checks any physical address used for a data or instruction access against these before allowing the access or generating the appropriate memory fault exception.

Although not strictly part of the ISA specification, we have also implemented some aspects of simple memory-mapped devices in Sail (timer, terminal, device interrupt routing) as an exploration of the use of the Sail language to describe other components of a complete platform model.

Our development of the Sail model has led us to contribute improvements in the RISC-V prose specifications, e.g. in the description of page-faults expected during page-table walks, and fixes to bugs in the corresponding address translation code of the widely-used Spike reference simulator. It has also pointed out ambiguities in the specification of interrupt delegation, and cases of missing reservation yields in Spike.

2.2 MIPS and CHERI-MIPS

CHERI-MIPS [Watson et al. 2018, 2015; Woodruff et al. 2014] is an experimental research architecture that extends 64-bit MIPS with support for fine-grained memory protection and secure compartmentalisation. It provides hardware *capabilities*, compressed 128-bit values including a base virtual address, an offset, a bound, and permissions; and object capabilities that link code and data pointers. Additional tag memory, cleared by any non-capability writes, records whether each capability-sized and aligned unit of memory holds a valid capability. This and other features make them unforgeable by software: each capability must be derived from a more-permissive one. One can either use capabilities in place of all pointers ("pure capability" code) or selectively ("hybrid").

CHERI has used executable formal models of the architecture as a central design tool since 2014, largely in L3 [Fox and Myreen 2010], coupled with traditional prose and non-formal pseudocode in the ISA specification document. Executability of the formal model (at some 100s of KIPS) has been vital, both to provide a reference to test hardware implementations against, and as a platform for software development that is automatically in step with the frequent architecture changes.

```
function clause execute (CIncOffsetImmediate(cd, cb, imm)) = {
 checkCP2usable();
 let cb_val = readCapReg(cb);
 let imm64 : bits(64) = sign_extend(imm);
 if register_inaccessible(cd) then
  raise_c2_exception(CapEx_AccessSystemRegsViolation, cd)
 else if register_inaccessible(cb) then
  raise_c2_exception(CapEx_AccessSystemRegsViolation, cb)
 else if (cb_val.tag) & (cb_val.sealed) then
  raise_c2_exception(CapEx_SealViolation, cb)
 else
  let (success, newCap) = incCapOffset(cb_val, imm64) in
  if success then writeCapReg(cd, newCap)
  else writeCapReg(cd, int_to_cap(to_bits(64, getCapBase(cb_val)) + imm64))
}
```

Fig. 4. CHERI-MIPS capability increment-offset instruction in Sail

Isabelle definitions generated from L3 have been used for proofs about compressed capabilities and of security properties of the architecture as a whole. This has all provided invaluable experience for the design of Sail, and our Sail CHERI-MIPS model is now mature enough to replace both the earlier L3 model and the non-formal pseudocode; the latter using Sail-generated LaTeX.

Our MIPS Sail model is just over 2000 non-blank, non-comment lines of Sail code, including sufficient privileged architecture features to boot FreeBSD, but excluding floating point and other optional extensions. The CHERI-MIPS model extends the MIPS model with approximately 2000 more lines and includes support for either the original 256-bit capabilities or a compressed 128-bit format, with the instructions themselves being expressed in a manner that is agnostic to the exact capability format. This is important because CHERI is under continuous development and the capability format has changed many times. For example, Fig. 4 shows the Sail semantics for the CHERI `CIncOffsetImmediate` instruction, to increment the offset of a capability; it makes the various security checks (and the priority among them) explicit.

2.3 ARMv8-A

This is our most substantial example by far: ARMv8-A is a modern industry architecture, underlying almost all mobile devices. It was announced in 2011 and has been enhanced through to ARMv8.2-A (2016), ARMv8.3-A (2016), and ARMv8.4-A (2018). It includes both 64-bit (AArch64) and 32-bit (AArch32) instruction sets. ARM also define related microcontroller (-M) and real-time (-R) variants.

The ARM architecture specifications have long used a custom pseudocode metalanguage, ASL, to express instruction behaviour. ASL has evolved over time. It was initially purely a paper language, an important part of the manuals but not mechanically parsed, let alone type-checked or executed. Reid led an effort within ARM to improve this, so that "machine-readable, executable specifications can be automatically generated from the same materials used to generate ARM's conventional architecture documentation" [Reid 2016, 2017; Reid et al. 2016]. This executable version of ASL is now used within ARM in documentation, hardware validation, and architecture design, alongside other modelling approaches.

In 2017 ARM released a machine-readable version of large parts of the ARMv8.2-A ASL, later updated to 8.3 and 8.4. This describes almost all of the sequential aspects of the architecture: instructions, floating point, page table walks, taking interrupts, taking synchronous exceptions such as page faults, taking asynchronous exceptions such as bus faults, user mode, system mode, hypervisor mode, secure mode, and debug mode. This provides a remarkable opportunity to rebase research on formal verification, analysis, and testing for ARM above largely complete (for sequential code) models based on an authoritative vendor-supplied semantics. However, that public release does not include tools for executing or reasoning about the ASL code, and it is not in a form usable for mechanised proof or integration with relaxed-memory concurrency semantics.

Accordingly, we have co-designed Sail and an `asl_to_sail` translation tool that can translate these ASL specifications into Sail (itself open-source), and thence into multiple theorem-prover and emulator-code targets. We have done this both for that public 8.3 release and for an ARM-internal version of 8.3 that additionally includes semantics of the many hundreds of system registers, some of which are needed during an OS boot; we are exploring the possibilities for also releasing this.

The total size of the public v8.3 specification when translated into Sail is about 23 000 lines, including 1479 functions, and 245 registers. This includes all 64-bit instructions, which are expressed as 344 function clauses in Sail, each of which may correspond to multiple assembly mnemonics. So far we have focused on the AArch64 64-bit part of the architecture, and have not translated the (optional) AArch32 32-bit mode. For the non-public v8.3 specification, which additionally includes a full description of all the system registers, we opted to not translate the vector instructions (they add considerably to the size of the specification), as we were primarily interested in the system-level parts of that specification. However, even without vector instructions it contains approximately 30 000 lines of specification with 1279 functions and 501 registers, implementing a total of 390 instructions. In contrast to the simple RISC-V instruction shown in Fig. 3, a single ARM instruction may involve hundreds of auxiliary functions, e.g. for checks of the current exception level and suchlike. While booting Linux, we found that each instruction performs on average around 800 calls to other auxiliary functions, and around 500 primitive operations.

In addition to translating the base specification, we have also added additional hand-written specification for timers, memory-mapped I/O (e.g. for a UART), and interrupt handling based on ARM's generic interrupt controller (GIC), which is sufficient to boot Linux using the model.

Our `asl_to_sail` tool is capable of translating the majority of ASL functions directly into Sail. Both Sail and ASL are first-order imperative languages, and most constructs can be translated in a straightforward manner. The main difficulty come from translating between the two type systems. Sail and ASL both have dependent types, but constructing well-typed Sail from ASL is sometimes non-trivial due to how the type systems differ. Sail's dependent type system and how it is translated from ASL is described more fully in §3. Roughly speaking our tool uses a mix of Sail's own type inference rules and some syntax-based heuristics to synthesise Sail types from ASL types. Some manual patching is needed, so `asl_to_sail` allows for interactive patching during translation. These patches are remembered and can be applied again automatically when the tool is re-run. We had to significantly re-engineer parts of the Sail language to support the kind of incremental parsing and type-checking required by `asl_to_sail`. Translating the non-public spec required 525 lines out of approximately 30 000 to be changed in some way, which represents patches to 143 top-level definitions out of 2158 that were translated. Most of these only require small tweaks and additional type annotations, with the median number of changed lines per patched top-level definition being 3. For the public specification, we also had to manually remove the mutual recursion from the translation table walk, as the ASL code is several hundred lines long and Isabelle has performance issues with such large mutually recursive functions. Fortunately, the maximum recursion depth in this case is only two.

Fig. 5 shows a sample instruction family automatically translated from ASL into Sail, the ADD / SUB (immediate) instructions. This illustrates several of the difficulties of working with the vendor definition: computed bitvector sizes and the use of imperatively updated local variables, initially **`undefined`**. `AddWithCarry` is an auxiliary pure function, defined in ASL and also translated to Sail,

```
val aarch64_integer_arithmetic_addsub_immediate : forall ('datasize : Int).
  (int, int('datasize), bits('datasize), int, bool, bool) -> unit
  effect {escape, rreg, undef, wreg}

function aarch64_integer_arithmetic_addsub_immediate ('d,datasize,imm,'n,setflags,sub_op)
 = { assert(constraint('datasize in {8, 16, 32, 64, 128}), "datasize␣constraint");
     result : bits('datasize) = undefined;
     operand1 : bits('datasize) = if n == 31 then aget_SP() else aget_X(n);
     operand2 : bits('datasize) = imm;
     nzcv : bits(4) = undefined;
     carry_in : bits(1) = undefined;
     if sub_op then {
       operand2 = ~(operand2);
       carry_in = 0b1
     } else carry_in = 0b0;
     (result, nzcv) = AddWithCarry(operand1, operand2, carry_in);
     if setflags then (PSTATE.N @ PSTATE.Z @ PSTATE.C @ PSTATE.V) = nzcv else ();
     if d == 31 & ~(setflags) then aset_SP(result) else aset_X(d, result)
 }
```

Fig. 5. ARMv8-A ADD / SUB (immediate) Instruction in Sail, as translated from ASL

that does the required arithmetic over the mathematical integers and also computes the resulting flag values. Register accesses are indirected via other auxiliary functions, e.g. `aset_X`, which do zero-extension if needed, select the appropriate register for the current exception level, check permissions, etc.

3 THE SAIL LANGUAGE

Sail as a language has to be sufficiently expressive to idiomatically express real ISAs, but no more expressive than necessary, otherwise translations to idiomatic prover definitions and fast emulator code would be more challenging, and readability by practising engineers would suffer.

The language is statically checked, with type inference and checking both to detect specification errors and to aid the generation of target code. We also have an interactive Sail interpreter, which can be used for debugging via breakpoints and interactively stepping through the evaluation of functions, and also provides a useful reference semantics for the language.

Following existing industry ISA pseudocode (both paper languages and ASL), Sail is essentially a first-order imperative language. Avoiding higher-order functions simplifies translation into C for efficient emulator code, simplifies proof about the ISA definitions, and avoids readability difficulties for the many engineers who are not familiar with functional languages. Instruction semantics are intrinsically effectful: instructions read and write registers and memory. In the sequential world, one might imagine that each instruction atomically updates a global machine state. In a realistic relaxed-memory concurrent setting, that is no longer the case, as one has to deal with finer-grain interactions between instructions. Perhaps surprisingly, though, at least for user-mode code it has so far been possible to treat the intra-instruction semantics sequentially, albeit with care to sequence specific register and memory operations correctly (and excluding ARM load-pair) [Flur et al. 2017; Gray et al. 2015; Pulte et al. 2018; Sarkar et al. 2011]. Whether this will remain true for systems-mode concurrency is unknown, but for the moment Sail does not require or support any intra-instruction concurrency.

Instructions refer to a global collection of the architectural registers. Some ISA specifications, including ARMv8-A, also rely on imperatively updatable local variables, but general references are not used. Sail supports passing references to registers, which is occasionally useful when trying to stick closely to the appearance of some industry ISAs, but we usually find it preferable to pass numeric (integer-range) register indices instead.

Most computation is over bitvectors, integer ranges, or integers, but user-defined enumerations are also needed, as are labelled records and (non-recursive) sums. Sail includes a built-in polymorphic list type. We also support user-defined type-polymorphic functions, and sum types can also be type-polymorphic, so one can implement a standard 'option' datatype as in most functional languages, but there is relatively little use of type polymorphism in our ISA models. However, we do need dependent types for bitvector lengths, integer range sizes, and operations on these, as such types can have arbitrary numeric constraints attached to them. This is the most technically challenging aspect of the language design, discussed in the next subsection. Operations on subvectors, and registers and record types with named sub-vector-bitfields, are also needed, including complex l-values for updates to specific parts of complex register state.

Architecture specifications commonly leave some bits loosely specified in specific contexts, or have broader loosely specified behaviour. Sail supports the former (**`undefined`**), with our backends providing various semantics as needed for different purposes. ARM also contains unpredictable behaviour, but this is modelled directly in the specification using ordinary functions that specify how the unpredictable behaviour should be handled.

The language includes both loops and recursion, as these are needed in the examples, e.g. in the ARMv8-A address translation code and the `BigEndianReverse` function, which reverses the bytes of a 16/32/64/128-bit bitvector. The Sail code for each instruction should be terminating, but Sail itself does not check that; instead it is left to the theorem-prover targets. The termination arguments are usually very simple, e.g. the address translation table walk has at most 4 iterations, and manual termination proofs are rarely needed because most loops are inherently terminating `foreach` loops.

Translating ASL into Sail led us to make changes to the language, to better express the ARMv8-A ASL code. For example, we originally did not plan to include exceptions in Sail, but ASL includes exceptions and exception handling, and uses them to implement some key aspects of the architecture, so we needed to add these to Sail to generate clean definitions (we translate these away in the various targets, as appropriate). We also had to add support for arbitrary-precision rational numbers, as ASL specifies several floating point operations by converting the binary floating point values to rationals, performing arbitrary-precision rational arithmetic, and then rounding back to floating point values with the appropriate precision. ASL also assumes that various components in the model are configurable at run-time, so we had to add support for special 'configuration registers' to be set by command line flags when the model is used. Such command line flags had to be made compatible with ARM's tools, so we could run our model with the ARM-internal AVS test-suite.

The language includes pattern matching, used especially for bitvector-concatenation pattern matching in decode functions, and for tuples.

We support various convenience features tuned for ISA specifications. Such specifications are typically large and flat, so Sail supports splitting functions and type definitions into multiple clauses which can be *scattered* throughout the file, interleaved with other definitions. Fig. 3 shows those clauses for a load instruction from RISC-V formalised in Sail, grouped together in the way they would be in an ISA manual; they could be followed by the clauses for another instruction, perhaps in a different file. Some ISAs, including ARMv8-A, rely on syntactic sugar to define pseudoregisters that can be used either within l-values or expressions, with semantics defined by user-defined functions; we support this with an overloading mechanism, much as ASL and L3 do. We include mechanisms for specifying bi-directional mappings between binary opcodes and assembly syntax, discussed below.

Good concrete syntax design is important for accessibility. Initially we aimed to exactly match the various industry ISA pseudocode languages, idiosyncratic as they are, and to use a C-like syntax for types and type annotations (e.g. `int x = ...`). Experience showed that neither were sustainable, and so we redesigned the Sail syntax more cleanly, but in a way that should still be readable by a broad community of hardware, software, and tool developers.

Targeting multiple provers —currently Isabelle/HOL, HOL4, and Coq— forces us to be careful that all language features can be translated into usable definitions for each, taking their different logical foundations into account. In general, we want to make use of our type system to generate nicer prover definitions where possible. As detailed in §4, we are currently able to generate Coq that preserves most of the liquid types from the Sail specification, whereas for Isabelle and HOL4 we perform a specialised partial monomorphisation that retains useful typing information where possible and tries to avoid duplicating code (as a naive full-monomorphisation pass would do).

Any proof based upon an ISA specification is dependent on the specification being correct, but an executable ISA specification is a large and complex program in its own right, as is Sail itself. We prioritise clarity over emulation performance when expressing the specifications, and we have devoted considerable effort to testing Sail, e.g. to ensure that the libraries of bitvector operations do the same thing in all targets. We only provide arbitrary precision integers, integer ranges, and rationals in Sail—this costs us some performance but guarantees that the specification cannot contain the kind of integer overflow and underflow issues that commonly affect programs written in languages like C. We have implemented Sail so that every intermediate rewriting step from the original Sail source to our theorem prover definitions can be type-checked.

3.1 Dependent Types for Bitvector Lengths and Integer Ranges

Bitvector indexing and manipulation is ubiquitous throughout ISA specifications, including bitvector concatenation and taking sub-bitvector slices, as is indexing into arrays, e.g. indexing from a 5-bit opcode field into an array of 32 general-purpose registers. In a simple idealised ISA the sizes of these bitvectors might all be constants, but in more realistic cases, especially in ARMv8-A, they are very often parameterised or computed. For example, 'size' arguments in functions are often small powers of two, like 16, 32, or 64, and instructions often come in variants for multiple sizes. It is also extremely common for such arguments to be linked to others and the return type in dependent ways, such as one argument giving the length of another argument in bytes. Expressions used for indexing often involve nontrivial integer ranges. Sometimes the context determines a bitvector size, e.g. for the result type of a zero- or sign-extend operation.

ASL necessarily supports all this, but it does not statically check bitvector accesses. In contrast, Sail is designed to statically check these things wherever we can, without needing the specification to fall back onto bit-list representations. We do so for many reasons: to statically catch many specification errors; to enable specifications to more directly express their intent; to make it possible to generate theorem prover definitions in which the correctness of bitvector accesses and suchlike are guaranteed by the prover type system, rather than needing additional proof; and to simplify the generation of fast emulator code, using fixed-width bitvectors instead of bit-lists.

Accordingly, Sail supports a form of *lightweight dependent types* for statically checking vector bounds and integer ranges. We use a system inspired by Rondon et al's liquid types [Rondon et al. 2008], which uses the Z3 SMT solver to automatically solve vector bounds and integer range constraints. In our experience, liquid types are ideal for an ISA description language, as they easily express the often relatively simple numeric constraints that occur when bounds-checking vector accesses or the use of integer ranges, without imposing much burden on the user. Often we only need a type with appropriate constraints to be declared as a top-level type signature, and all types and constraints in the function body can be automatically inferred and discharged.

As mentioned, it is extremely common to want to represent an integer value that is either 16, 32, or 64. This would be represented in Rondon et al's notation as: $\{i : int | i = 16 \lor i = 32 \lor i = 64\}$. Our syntax differs slightly from this for historical reasons (previous versions of Sail had a type-system more similar to dependent ML [Xi 2007]), and in Sail such a type would be specified as `{'i. 'i in {16, 32 64}, int('i)}`. This allows us to write commonly used types succinctly, e.g. `bits(8 * 'n)` for a bitvector of n bytes, but such types can be converted into liquid types notation such as $\{m : bits | length(m) = 8 * n\}$ in this case, as described in §6.

Rondon et al's inference algorithm operates in steps: First it performs Hindley-Milner type inference, before using syntax directed liquid typing rules to generate *liquid constraints*, which are solved in a final third step. In Sail we use a syntax-directed bidirectional type-system (along the lines of [Dunfield and Krishnaswami 2013]), so we can generate and solve the constraints as part of the ordinary typing-rules in a single type checking pass. While this means we do not have full type inference, in practice we mostly require top-level type declarations, with types within function bodies being automatically inferred.

```
val LSL_C : forall ('N : Int), 'N >= 1.
  (bits('N), int) -> (bits('N), bits(1)) effect {escape}

function LSL_C (x, shift) = {
  assert(shift > 0);
  let shift as 'S : range(1, 'N) = if shift > 'N then 'N else shift;
  let extended_x : bits('S + 'N) = x @ Zeros(shift);
  let result : bits('N) = slice(extended_x, 0, 'N);
  let carry_out : bits(1) = [extended_x['N]];
  return (result, carry_out)
}
```

Fig. 6. Fully annotated left shift with carry function

Fig. 6 shows an example of how dependent types for bitvectors are often used in Sail. The assert guarantees that the `shift` variable is greater than zero, and the next **`let`** statement forces shift to be in the (inclusive) range `1` to `'N`, which the type checker will prove based on the assert and the type constraint `'N >= 1`. In order to refer to the value of shift in type signatures later in this function, we give it a name as a type variable `'S`. As in ML, identifiers starting with ticks are type variables. The next line extends the input bitvector `x` with a number of zeros equal to shift, resulting in a bitvector of length `'S + 'N`. Then we take a slice from index `0` of length `'N`. Here the type system will prove that `'N <= 'S + 'N` to show that the slice does not violate the bounds of `extended_x`. The next line accesses the carry bit. Here the type system relies on the fact that `'S` must be greater than `0` to establish that `'N` is a valid index into `extended_x`. In practice most of the manual type signatures in Fig. 6 are not required, and the body of the function can be written as below.

```
  let shift : range(1, 'N) = if shift > 'N then 'N else shift;
  let extended_x = x @ Zeros(shift);
  let result = slice(extended_x, 0, 'N);
  let carry_out = [extended_x['N]];
```

We return to Sail's type system in more detail in §6, where we formalise a core calculus of Sail. First, however, we report on our experience with translating dependent types from ASL to Sail, followed by a presentation of other features of Sail.

Translating from ASL to Sail: Dependent Types As mentioned in §2.3 there are differences in the type systems between ASL and Sail that make generating type-correct Sail a significantly

```
bits(N) FPThree(bit sign)
  assert N IN {16,32,64};
  constant integer E =
    (if N == 16 then 5
     elsif N == 32 then 8
     else 11);
  constant integer F = N - (E + 1);
  exp = '1':Zeros(E-1);
  frac = '1':Zeros(F-1);
  return sign : exp : frac;
```

Fig. 7. Original FPThree ASL

```
val FPThree : forall 'N. bits(1) -> bits('N)
  effect {escape}
function FPThree sign = {
  assert('N == 16 | 'N == 32 | 'N == 64);
  let E : {|5, 8, 11|} =
    if 'N == 16 then 5
    else if 'N == 32 then 8
    else 11;
  let F = 'N - (E + 1);
  let exp = 0b1 @ Zeros(E - 1);
  let frac = 0b1 @ Zeros(F - 1);
  sign @ exp @ frac
}
```

Fig. 8. FPThree function translated into Sail

harder task than just generating syntactically-correct Sail. ASL's typesystem is a compromise between expressivity and the ability to detect errors: like Sail, it provides dependent types for bitvector sizes and statically checks every function call but, unlike Sail, uses of bitvector indexing are not statically checked. During the conversion of ARM's pseudocode to ASL, ARM's architects requested a more flexible type system, and some form of flow-sensitive typing was considered but then rejected because it was not clear how to get good error messages, how to explain to users what could and could not be typechecked and how to avoid path explosion. Automatic translation of ASL to Sail is therefore not just practically useful to Sail users but also useful to ARM, since it demonstrates that ASL could also adopt flow-typing and gain the benefits of more expressive types and stronger checking. ARM's internal ASL steering committee is currently exploring this option.

To illustrate the translation of these dependent types, consider the Sail function FPThree in Fig. 8 translated from the ASL in Fig. 7. It constructs the floating point value 3.0, as either a 16, 32, or 64-bit vector. As can be seen, the length of both the exponent (`E`) and mantissa (`F`) are calculated based on the length of the returned bitvector, given by the type variable `'N`. Sail will check that the length of `sign @ exp @ frac` is equivalent to `'N`. In Fig. 8, the length of the exponent `E` has the integer set type `{|5,8,11|}`. Currently only this type signature must be present for this function to type check, while the other type signatures can be inferred automatically (in practice our translation tool will add type signatures wherever it can, but we have omitted them here for brevity).

Unlike ASL, Sail has flow-sensitive typing, so the assert statement will guarantee to the type checker that `'N` is either `16`, `32` or `64` in the body of the function. Typically in our hand-written Sail models, one would put such a constraint on `'N` in the type signature, as `val FPThree : forall 'N in {|16,32,64|}. bits(1)-> bits('N)`, rather than an assert statement, but in ASL such information is often encoded in runtime assertions. Rather than trying to lift this information into the type signatures, we have generally found that sticking closely with idioms found in ASL, and ensuring that such idioms also work well in Sail (e.g. by adjusting our rules for flow-sensitive typing) has been the best way to easily translate large amounts of ASL into Sail without a large amount of manual effort. Despite this we do make some stylistic improvements when translating ASL code where possible, such as turning some mutable variables in ASL into immutable let-bindings, e.g. `exp` and `frac` in Fig. 7. We also have to add an *escape* effect to the function in Sail, as the assert could fail and exit the function. Sail has a basic effect system that keeps track of whether functions read and write registers, and how they interact with memory, as well as other effects such as the aforementioned escape for non-local control flow. These effects are used in RMEM for the concurrency models, and also to decide if code needs to be monadic in the theorem prover backends. Sail can infer these effects automatically, but we find making them explicit in top-level type signatures helps with readability.

Currently we have slightly relaxed Sail's strict bounds checking behaviour for the translated ASL. Sail is able to fully check 2695 bounds checking problems encountered in the ARM specification, with 48 that are currently not automatically solvable. While we could resolve this by simply adding assertions in the specification where these problems occur using `asl_to_sail`'s patching mechanism, we instead plan to improve `asl_to_sail`'s ability to infer tight ranges on integer variables, which should help in these cases and also improve code generation.

3.2 Mappings and String Pattern-Matching

So far we have described the aspects of Sail needed to specify the decoding of binary instructions and their dynamic semantics. When working with an ISA specification, one often also needs the ability to define the assembly language syntax, and the pretty printing and parsing (disassembly and encoding/assembly) functions between it and binary instructions.

Sail *mappings* allow the definition of both sides of a bidirectional function at once, for example a parser and pretty-printer. This is similar to existing work on bidirectional programming, e.g. Boomerang [Bohannon et al. 2008], but much more lightweight. Mappings can be simply defined as a set of pattern-matching clauses, where the right-hand-side of the pattern-match is in itself a pattern, or as pairs of functions, allowing for more complex behaviour such as string conversion to and from integers. The type system allows mappings to be called as if they were functions, with the inferred result type determining in which direction the mapping runs. (This gives rise to the restriction that the types on either side of a mapping must be different.) In the implementation, mappings are expanded into two conventional pattern-matches. Mappings interact usefully with string pattern-matching. We allow string concatenation to be used as an operator in pattern-matches, and attempt a simple left-to-right exact matching (compiled into successively nested guarded pattern-matches). To date, we have handwritten mappings for RISC-V, as in Fig. 9; it should also be possible to generate mappings for ARMv8-A from ARM-supplied metadata.

4 GENERATION OF THEOREM PROVER DEFINITIONS

We now turn our attention to the backends of Sail, starting with theorem provers: One of our main goals is to provide models in various provers upon which verification projects that need detailed ISA specifications can build. For this purpose, we implement automatic translations from Sail code to definitions for different popular theorem provers; we target Isabelle/HOL, HOL4, and Coq (we currently have complete Coq translation only for MIPS). Most of the translation pipeline from Sail to those targets is shared, transforming features such as pattern guards and scattered definitions into forms supported by the targets. Some parts of this pipeline are also shared with generation of emulator code, in §5.

For Isabelle/HOL and HOL4, we first translate to Lem as an intermediate language, using Lem to generate the prover definitions [Mulligan et al. 2014]. Since the RMEM concurrency models are specified in Lem, this translation is also used for the integration of Sail ISA models into RMEM [Pulte

```
enum rop = { RISCV_ADD, RISCV_SUB, ... }
union clause ast = RTYPE : (regbits, regbits, regbits, rop)
mapping rtype_mnem : rop <-> string = { RISCV_ADD <-> "add", RISCV_SUB <-> "sub", ... }
mapping clause assembly = RTYPE(rs2, rs1, rd, op) <->
   rtype_mnem(op) ^ spc() ^ reg_name(rd) ^ sep() ^ reg_name(rs1) ^ sep() ^ reg_name(rs2)
```

Fig. 9. Parts of the Sail assembly syntax for RISC-V RTYPE binary operation instructions.

et al. 2018]. For Coq, we generate Coq definitions directly from Sail to make better use of Coq's type system, in particular to preserve dependent types for bitvector lengths, which are not supported by Lem or the other provers. We describe how we deal with those dependent types for Isabelle/HOL and HOL4 in §4.1. We explain further details of the translation, in particular the monadic treatment of effects, in §4.2. Our translations are generally intended to handle all of Sail, but there are areas where we currently require additional restrictions, which are all compatible with our models. For example, in monomorphisation we currently only support case splits on the types used in practice.

4.1 Bitvector Length Monomorphisation

As described above, Sail's type system can track the sizes of bitvectors with a reasonably rich suite of type-level arithmetic operations, backed by constraint solving. This is convenient for expressing data-dependent bitvector sizes, such as the data size used in the instruction shown in Fig. 5. However, Isabelle/HOL and HOL4 only permit very simple expressions at the type level; essentially just constants and variables. To translate into these, we have added a bitvector library to the intermediate Lem language, and perform a partial monomorphisation of models to fit them into these less expressive type systems.

The approach is similar to one previously used by ARM during translation to Verilog for model checking [Reid et al. 2016, §4], where additional case splits are added to ensure that all bitvector sizes will be constant, and constant propagation reveals exactly what those sizes are. Our goals are slightly different, however. We want to retain the original model structure as far as possible, in particular avoiding the duplication of functions due to specialisation. Fortunately, Isabelle and HOL4 support non-dependent size parametricity, representing sizes as type variables. For example, in the ARMv8-A model a case split for the data size can sometimes be introduced in the decoder, and the more complex execution function left parametric in the size.

The location of the case splits to be introduced is determined by an automated interprocedural dependency analysis. Case splits on bitvector and enumeration variables are simple to introduce, but for integer variables we consult the Sail typing to find the set of possible values. The constant propagation is also mildly interprocedural so that trivial helper functions can be eliminated. When a case split refines the type of an argument or a result, e.g., from `bits('n)` to `bits(8)`, etc. by a case split on `'n`, we introduce a cast using a primitive zero-extension operation, which will change the type but not the value.

For example, when applied to the definition of FPThree in Fig. 8 the analysis notices that the types of `exp` and `frac` depend on `E` and `F`, which ultimately depend on the parameter `'N`. It then introduces a case split on `'N`, using the three possibilities from the assertion, and adds casts for the returned values to `bits('N)`.

To reduce the amount of code duplication we perform a transformation on type signatures before monomorphisation. This lifts complex sizes out of types in function signatures, allowing them to be treated as type parameters. For example, a simple memory loading function might have the signature

```
val load : forall 'n, 'n >= 0. (bits(64), bits(8 * 'n)) -> bits(64)
```

suggesting that `8 * 'n` must be monomorphised in the body of `load` because it cannot be represented in Lem's type system. Instead, we rewrite it to the equivalent signature

```
val load : forall 'n 'm, 'n >= 0 & 'm = 8 * 'n. (bits(64), bits('m)) -> bits(64)
```

making the size a proper type parameter, which can be expressed in Lem.

For some combinations of variable-size bitvector operations it is preferable to rewrite them in terms of shifting and masking on a suitably large fixed-size bitvector. For example, comparing two slices of bitvectors `v[x .. y] == w[x .. y]` can be replaced by masking `v` and `w` and comparing,

```
fun execute_LOAD :: "12 word=>5 word=>5 word=>bool=>word_width=>bool=>bool=>bool M" where
"execute_LOAD imm rs1 rd is_unsigned width aq rl = (
 rX (regbits_to_regno rs1) >>= (λ w__0.
 let (vaddr :: xlenbits) = add_vec w__0 ((EXTS 64 imm)) in
 if check_misaligned vaddr width then
  handle_mem_exception vaddr E_Load_Addr_Align >> return False
 else
  translateAddr vaddr Read Data >>= (λ w__1 :: TR_Result.
  (case w__1 of
   TR_Failure (e) => handle_mem_exception vaddr e >> return False
  | TR_Address (addr) =>
    (case width of
     BYTE => mem_read addr 1 aq rl False >>= (λ w__2 :: 8 word MemoryOpResult.
           process_load rd vaddr w__2 is_unsigned)
    | HALF => mem_read addr 2 aq rl False >>= (λ w__4 :: 16 word MemoryOpResult.
           process_load rd vaddr w__4 is_unsigned)
    | WORD => ...
    | DOUBLE => ...))))"
```

Fig. 10. RISC-V load instruction translated into Isabelle

without needing to monomorphise `y-x`. We have a small library of combined operations like this, and a set of rewrites to use them.

4.2 Monadic Translation of Effects

The translation of imperative, effectful Sail code into monadic code for the generation of prover definitions is largely standard, rewriting into a sequence of monadic and let-bindings similar to A-normal form [Flanagan et al. 1993], but where the criterion is that arguments to functions must be pure. For example, the effectful first operand of `add_vec` in Fig. 10 has been pulled out into a monadic bind. Local mutable variable updates are translated to pure let-bindings, where local blocks that update variables, e.g. loop bodies and the branches of if-expressions, are rewritten to return the updated values so that they can be picked up by the surrounding context, while respecting their scoping. This avoids generating and handling per-function local state spaces, and the need for a polymorphic state that is difficult to support in the non-dependently typed backends. Early return statements in functions are translated in terms of the Sail exception mechanism, by throwing the return value and wrapping the function body in a try-catch-block, where early returns and proper exceptions are distinguished using a sum type. The translation assumes a left-to-right evaluation order of effectful function arguments. Boolean conjunction and disjunction are special-cased, however, to give them a short-circuiting semantics. This is required for the ARMv8-A specification, which includes expressions such as `UsingAArch32()&& AArch32.ExecutingLSMInstr()`, where an assertion fails in the right-hand function if the left-hand function does not return `true`.

Our translation targets two monads with different purposes. The first is a state monad with nondeterminism and exceptions. It is suitable for reasoning in a sequential setting, assuming that effectful expressions are executed without interruptions and with exclusive access to the state. Nondeterminism is needed for features such as load reserve/store conditional instructions that can succeed or fail, and it can be used to model behaviour that the architecture loosely specifies. For example, some variants of the LDR ("load register") instruction in ARMv8-A take two registers as parameters and write to both of them; however, if they refer to the same general-purpose register, the architecture allows different possible behaviours, including ignoring the instruction or raising an exception. The second monad can be used for a concurrent semantics, where a standard state monad interpretation of the Sail code is insufficient. In particular, in the relaxed memory models of ARMv8 and RISC-V, instructions observably execute out-of-order, speculatively, and non-atomically, and so the semantics needs to expose the instructions' effects at a finer granularity. For example, a store instruction waits until all program-order preceding memory accesses have resolved their address before it can propagate, and so it can observe intermediate states in the execution of those preceding instructions. To support integrating Sail with these concurrency models, we use a free monad over an effect datatype. It is implemented in terms of a monad type as below, parameterised by the return value type `'a`, the register value type `'r`, and the exception type `'e`:[1]

```
type monad 'r 'a 'e =
  | Done of 'a
  | Read_mem of read_kind * address * nat * (list mem_byte -> monad 'r 'a 'e)
  | Write_ea of write_kind * address * nat * monad 'r 'a 'e
  | Write_memv of list mem_byte * (bool -> monad 'r 'a 'e)
  ...
  | Barrier of barrier_kind * monad 'r 'a 'e
  | Read_reg of register_name * ('r -> monad 'r 'a 'e)
  | Write_reg of register_name * 'r * monad 'r 'a 'e
  | Undefined of (bool -> monad 'r 'a 'e)
  | Exception of 'e (* Exception thrown *)
```

A value of this type is either `Done a`, representing a finished computation with a pure value of type `'a`, or an *effect request*: each of the other constructors represents an effect, typically together with some parameters specifying the particular request, and a continuation. For example, `Read_reg "PC" k` is a request to the execution context to read the PC register and pass its value into the continuation `k`. Another example is `Undefined k`, which requests a Boolean value from the execution context, e.g. to make a nondeterministic choice or to resolve an undefined bit to a concrete value. The definition of the monad leaves the meaning of these instruction effects open —the monad's bind operator simply "nests" the requests— and the monad instead delegates handling the effects to an *effect interpreter* outside the instruction semantics definition. To support the integration with a concurrency model that executes these instruction definitions out-of-order, the monad type has effects for all concurrency-relevant events of the instruction's execution: for example, the `Barrier` effect announces memory barriers, register reads and writes are explicit requests (`Read_reg` and `Write_reg`) to enable handling the fine-grained memory ordering resulting from dataflow dependencies, and the writing of memory is split into the announcement of the write address, `Write_ea`, and the writing of the value, `Write_memv`, so program-order succeeding instructions can be informed about the address as early as it is known.

4.3 Target-Specific Differences in the Translation

Most of the translation pipeline is shared between the different provers, e.g. the rewriting of bitvector patterns to guarded patterns, and then the rewriting of those to a combination of if-expressions and unguarded pattern matches using an algorithm similar to that of [Spector-Zabusky et al. 2018, §3.4].[2] There are some differences, however, mainly due to the differences in the type systems of the provers.

[1]Such a monad is often implemented using a generic functor `Free`, e.g. in Haskell, but since this is not supported by the type system of Isabelle, we merged it with the concrete effects into a single type.

[2]The main differences are that we use a different grouping strategy for clauses (overlapping instead of mutually exclusive groups, since bitvector pattern rewriting can lead to many consecutive, overlapping patterns), and that we keep fall-through branches in place instead of pulling them out into let-bindings, since that could interfere with both effects and flow-typing.

```
Definition FPThree (N__tv : Z) (sign : mword 1) : M (mword N__tv) :=
  assert_exp' ((Z.eqb N__tv 16) \/ (Z.eqb N__tv 32) \/ (Z.eqb N__tv 64)) "" >>= fun _ =>
  let '(existT _ E _) :=
    (if sumbool_of_bool ((Z.eqb N__tv 16)) then build_ex 5
     else if sumbool_of_bool ((Z.eqb N__tv 32)) then build_ex 8
     else build_ex 11)
     : {n : Z & ArithFact (In n [5; 8; 11])} in
  let F := Z.sub N__tv (Z.add E 1) in
  let exp := concat_vec (vec_of_bits [B1] : mword 1) (Zeros__0 (Z.sub E 1)) in
  let frac := concat_vec (vec_of_bits [B1] : mword 1) (Zeros__0 (Z.sub F 1)) in
  returnm ((autocast (concat_vec sign (concat_vec exp frac))) : mword N__tv).
```

Fig. 11. FPThree function translated into Coq

Isabelle The prover definitions generated from a Sail model should ideally be parametric in the monad, but this is not supported by Isabelle's type system. Hence, when generating Isabelle definitions, we use the free monad, and provide a lifting to the state monad that enables reasoning in terms of the latter, if desired (cf. §8).

HOL4 When generating HOL4 definitions, we use only the state monad, since HOL4's datatype package does not currently support the free monad's type (it has a recursion on the right of a function arrow).

Coq The dependent type system in Coq enables us to give a much more direct translation of Sail's rich type information than would be possible with Lem's rudimentary Coq output support. The main difference in our Coq translation compared to our other backends is that the type-level sizes and constraints are fully retained in the generated Coq definitions. In particular, Sail's existential types are translated to dependent pairs in Coq. This can be seen in Fig. 11, the translation of Fig. 8, where a dependent pair is built for `E` to show that it is in the set $\{5, 8, 11\}$. However, it would be extremely challenging to reuse proofs about the constraints from the SMT solver used during Sail type checking, so instead we use a Coq typeclass wrapper to trigger a constraint solving tactic. In Fig. 11 this is done by the `build_ex` function. The core of the tactic is Coq's implementation of the Omega Presburger arithmetic decision procedure [Pugh 1991], with additional preprocessing to transform information from the context into a useful form and to evaluate constant powers of 2. The solver can also be extended by adding facts to a Coq hints database.

There is an important difference between the type checking in Sail and Coq: Sail uses the SMT solver to assist with type equivalence and subtyping checks automatically, whereas Coq only uses its built-in notion of reduction. This is often inadequate; for example, even `1 * z` does not reduce to `z` for Coq's integer type, `Z`, whereas Sail considers the types `bits(1 * z)` and `bits(z)` to be interchangeable. Our Coq backend detects differences like this and inserts a cast function. The cast function has a constraint that the two integer expressions are equal (which is automatically inferred from the context by Coq), and triggers the constraint solving tactic during type checking. For example, a proof of `1 + (1 + (E - 1) + (1 + (F - 1))) = N__tv` is used by the cast in the last line of `FPThree`.

The Coq backend is still under development. In particular, unbounded loops need to be manually adjusted to show that they terminate. Nonetheless, it is already sufficient to produce a full Coq translation of our MIPS model, and almost all of RISC-V. We also intend to experiment with a version without such rich typing and assess which is easier to reason with.

5 GENERATION OF EMULATORS

In addition to generating theorem prover definitions, it is also important to support emulation, with enough performance for validation purposes. We implement a simple direct mapping from Sail into OCaml, as well as more involved optimised compilation path to C. The simple OCaml translation is primarily used as a validation tool for the more involved C translation, and for prototyping.

Our C generation involves several steps. First we use the same type-preserving rewrites we use when generating theorem prover code, to eliminate some features and syntactic sugar found in the full Sail language, then we map into an A-normal form representation which is very similar to the MiniSail language described in §6. This is then translated to a lower-level intermediate representation, before we generate C code. Our intermediate representation is not particularly tied to C, so we could easily switch to e.g. LLVM IR if desired at some point in the future. There are three main optimisations that we perform during this compilation process that greatly speed up the resulting code.

First, bitvectors that are statically known to be 64-bits in length or less are mapped to 64-bit unsigned integers in C, whereas variable length bitvectors or those that are larger than 64 bits are mapped onto arbitrary length bitvectors implemented using GMP integers. Furthermore, some large bitvector types are mapped onto multiple 64-bit integers if necessary—this was key to get good performance for MIPS address translation, which features 117-bit wide bitfields for TLB entries.

Secondly, we use our liquid types and constraints to detect integer types that are bounded to fit within a 64-bit signed integer type. For example, the `int_of_accessLevel` function below is returning an integer constrained to be either 0, 1, or 2. Hence, we can use a fixed-width integer type rather than an arbitrary precision GMP integer. In general we can optimise any such integer types, provided Z3 can prove they fit within the bounds of a 64-bit signed integer. We could have tried to map statically smaller types to even smaller variables, such as mapping 32-bit bitvectors to 32-bit integers, but our profiling did not suggest that this would provide significant performance gains.

```
function int_of_AccessLevel(level : AccessLevel) -> {|0,1,2|} =
  match level { User => 0, Supervisor => 1, Kernel => 2 }
```

This optimisation turns out to be important, because small often-used functions like the above can be quite costly if they are forced to use arbitrary-precision integers. The `int_of_accesslevel` function accounted for nearly 5% of the time taken booting FreeBSD on our MIPS model before we implemented this optimisation.

Thirdly, we note that the vast majority of functions in ISA specifications are non-recursive, with the ARM specification containing only a single recursive function (for endianness reversal) and one small group of mutually recursive functions (for nested page table walks). For all non-recursive functions we are able to statically preallocate any space they need on the heap for arbitrary-precision bitvectors and integers to avoid calling malloc and free.

In total these optimisations gave us around a 13x performance increase in the performance of the generated emulator code. For ARM, we went from approximately 4000 instructions per second (IPS) to around 53 000 IPS on an Intel i7-7700 CPU with 3.6 GHz. For MIPS, which is significantly simpler, we were able to achieve performance of between 500 000 and 1.5 million IPS (this difference being caused by the number of memory accesses), with an average of about 850 000 IPS. By contrast, when compiling Sail into OCaml we can execute instructions at around 1800 IPS for ARM, and when using our interactive interpreter we can execute ARM instructions at about 30 IPS.

6 FORMAL TYPE SYSTEM

After having presented our models, the Sail language and its backends, we now return to Sail's type system in more formal terms. It is particularly important for an ISA specification language to have a solid foundation: ISA specifications are long-lived, and any formal work above them will involve substantial investment, so we want Sail to be robust and stable. Unfortunately, the initial version of Sail (like most architecture description languages) was implementation-defined: Sail was whatever the implementation would accept. This made evolving the system —even bug-fixing— a fraught process, and the fact that Sail's type inference relied on a complex custom constraint solver meant that there were many bugs to fix. Solving this problem required a degree of care, since we wanted to improve the language design "in place". To guide the evolution of Sail, we introduced a kernel calculus, MiniSail.

The two key properties we prove of MiniSail are (a) type safety, and (b) decidability of type checking. As a first-order language, the dynamic semantics of Sail are largely straightforward, but type safety is not entirely obvious, because Sail's support for type-dependency means that safety can rely upon control- and data-dependent properties. For software engineering reasons, we wanted to move away from a hand-rolled constraint solver to using an off-the-shelf SMT solver such as Z3. This both simplifies our implementation, and increases its reliability, as a widely-used solver like Z3 will be tested much more thoroughly than a hand-rolled solver. However, a danger is that we would merely replace one ad-hoc set of heuristics (embodied in our solver) with Z3's – a different set not even under own control. Luckily, SMT solvers come with a very clear guarantee: any query in the quantifier-free fragment is decidable, and in practice those we generate are efficient. So our decidability proof fundamentally exists to ensure that Sail's type system only generates pure SMT queries, ensuring that the specification of Sail is independent of the details of the solver. We include in MiniSail features from Sail that we judge to have an influence on the form of the SMT queries generated. These are a selection of value constructor and elimination forms, immutable and mutable variable binding statements, and imperative language statements. An example of a Sail feature that we do not include in MiniSail are records as these can be emulated using pairs.

Fig. 12 presents MiniSail's grammar and a table of judgements, and Fig. 13 presents a selection of the typing rules. The grammar in Fig. 12 defines a language in a slight variant of A-normal form. Programs are defined from expressions (which include arithmetic, variables, and function applications), and statements (which include let-binding, conditionals, and the assignment and declaration of mutable variables). Note that we distinguish bindings of immutable variables $\textbf{let}\ x = e\ \textbf{in}\ s$ from the declaration of mutable variables $\textbf{var}\ u\ : \tau = v\ \textbf{in}\ s$. Types τ are set-comprehension-style: they are the elements of a first-order base type, together with a boolean constraint restricting which elements of the base type are in τ.

As described in §3, MiniSail is a bidirectional type system, with a type synthesis mode $\Pi; \Gamma; \Delta \vdash e \Rightarrow \tau$ for expressions, and a type checking mode $\Pi; \Gamma; \Delta \vdash s \Leftarrow \tau$ for statements. The presence of three contexts arises from the fact that MiniSail is a first-order, imperative language. Function definitions are separated into a context Π, and Γ and Δ control the scoping of immutable bindings and mutable variables, respectively. Bindings in Γ are associated with a base type b and constraint ϕ, denoted $b[\phi]$, and variables in Δ are associated with a type τ.

The key technical idea ensuring decidability is as follows: whenever we have an expression in the SMT fragment, we record an exact constraint. Otherwise, we merely propagate any SMT constraints by attaching them to variables, using A-normal form to ensure that there is always a name to attach a constraint to. Rules 1-5 in Fig. 13 give typing rules for expressions. Since each of these terms is in the SMT fragment, we can generate an exact equality constraint for them. However, rules 6 and 7 (for function applications and mutable variables) are for terms not in the SMT fragment, and so we use the type as an approximation, with mutable variables just looking up the type in the environment Δ and function applications returning the result type with the argument value substituted in. Rules 8-12 play the same game with statements. The rules for variables (9 and 10) state no equalities between expressions and variables, since the expressions include forms (e.g.,

$$
\begin{array}{llcl}
\text{Value} & v & ::= & x \mid n \mid \mathbf{T} \mid \mathbf{F} \mid (v, v) \mid \mathrm{C}\ v \mid () \\
\text{Expression} & e & ::= & v \mid v + v \mid v \leq v \mid f\ v \mid u \mid \mathbf{fst}\ v \mid \mathbf{snd}\ v \mid \mathbf{len}\ v \mid \mathbf{concat}\ v\ v \\
\text{Statement} & s & ::= & v \mid \mathbf{let}\ x = e\ \mathbf{in}\ s \mid \mathbf{let}\ x : \tau = s\ \mathbf{in}\ s \mid \mathbf{if}\ v\ \mathbf{then}\ s\ \mathbf{else}\ s \mid \\
 & & & \mid \mathbf{match}\ v\ \{\ \mathrm{C}_1\ x_1 \Rightarrow s_1\ ,\ ..\ ,\ \mathrm{C}_n\ x_n \Rightarrow s_n \} \\
 & & & \mid \mathbf{var}\ u\ : \tau = v\ \mathbf{in}\ s \mid u := v \mid \mathbf{while}\ (s_1)\ \ \mathbf{do}\ \{s_2\} \\
\text{Base type} & b & ::= & \mathbf{int} \mid \mathbf{bool} \mid \mathbf{unit} \mid \mathbf{bitvec} \mid \mathrm{tid} \mid \mathrm{b} \times \mathrm{b} \\
\text{Constraint} & \phi & ::= & \mathbf{T} \mid \mathbf{F} \mid e^{-} = e^{-} \mid e^{-} \leq e^{-} \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \neg\phi \\
\text{Refinement Type} & \tau & ::= & \{ z : b \mid \phi\ \} \\
\text{Function Definition} & \mathrm{fd} & ::= & \mathbf{val}\ \ f : (x : b[\phi]) \to \tau \\
 & & & \mathbf{function}\ f(x) = s \\
\text{Datatype Definition} & \mathrm{td} & ::= & \mathbf{union}\ \mathrm{tid} = \{\mathrm{C}_1 : \tau_1, ..., \mathrm{C}_n : \tau_n\} \\
\text{Definition} & \mathrm{def} & ::= & \mathrm{td} \mid \mathrm{fd} \\
\text{Program} & \mathrm{p} & ::= & \mathrm{def}_1\ ; ..\ ; \mathrm{def}_n\ ; \mathrm{s}
\end{array}
$$

Π	Function definition context	Γ	Immutable variable context	Δ	Mutable variable context

$\Pi; \Gamma \vdash v \Rightarrow \tau$	Type synthesis values	$\Pi; \Gamma \vdash v \Leftarrow \tau$	Type checking values
$\Pi; \Gamma; \Delta \vdash e \Rightarrow \tau$	Type synthesis expressions	$\Pi; \Gamma; \Delta \vdash s \Leftarrow \tau$	Type checking statements
$\Pi; \Gamma \vdash \tau_1 \lesssim \tau_2$	Subtyping	$\Pi; \Gamma \models \phi$	Validity

Fig. 12. MiniSail Grammar Fragment and Judgements

$$
\frac{}{\Pi; \Gamma \vdash n \Rightarrow \{z : \mathbf{int} | z = n\}}\ \mathbf{1}
\qquad
\frac{}{\Pi; \Gamma \vdash \mathbf{T} \Rightarrow \{z : \mathbf{bool} | z = \mathbf{T}\}}\ \mathbf{2}
\qquad
\frac{}{\Pi; \Gamma \vdash \mathbf{F} \Rightarrow \{z : \mathbf{bool} | z = \mathbf{F}\}}\ \mathbf{3}
$$

$$
\frac{x : b[\phi] \in \Gamma}{\Pi; \Gamma \vdash x \Rightarrow \{z : b | z = x\}}\ \mathbf{4}
\qquad
\frac{\begin{array}{c}\Pi; \Gamma \vdash v_1 \Rightarrow \{z_1 : \mathbf{int} | \phi_1\} \\ \Pi; \Gamma \vdash v_2 \Rightarrow \{z_2 : \mathbf{int} | \phi_2\}\end{array}}{\Pi; \Gamma; \Delta \vdash v_1 + v_2 \Rightarrow \{z_3 : \mathbf{int} | z_3 = v_1 + v_2\}}\ \mathbf{5}
\qquad
\frac{\begin{array}{c}\mathbf{val}\, f : (x : b[\phi]) \to \tau\ \in\ \Pi \\ \Pi; \Gamma \vdash v \Leftarrow \{z : b | \phi\}\end{array}}{\Pi; \Gamma; \Delta \vdash f\ v \Rightarrow \tau[v/x]}\ \mathbf{6}
$$

$$
\frac{u : \tau \in \Delta}{\Pi; \Gamma; \Delta \vdash u \Rightarrow \tau}\ \mathbf{7}
\qquad
\frac{\Pi; \Gamma \vdash v \Leftarrow \tau}{\Pi; \Gamma; \Delta \vdash v \Leftarrow \tau}\ \mathbf{8}
\qquad
\frac{\Pi; \Gamma; \Delta \vdash e \Rightarrow \{z : b | \phi\} \quad \Pi; \Gamma, x : b[\phi[x/z]]; \Delta \vdash s \Leftarrow \tau}{\Pi; \Gamma; \Delta \vdash \mathbf{let}\, x = e\, \mathbf{in}\, s \Leftarrow \tau}\ \mathbf{9}
$$

$$
\frac{\begin{array}{c}\Pi; \Gamma \vdash v \Leftarrow \tau \\ \Pi; \Gamma; \Delta, u : \tau \vdash s \Leftarrow \tau_2\end{array}}{\Pi; \Gamma; \Delta \vdash \mathbf{var}\, u : \tau := v\, \mathbf{in}\, s \Leftarrow \tau_2}\ \mathbf{10}
\qquad
\frac{\begin{array}{l}\Pi; \Gamma \vdash v \Rightarrow \{x : \mathbf{bool} | \phi_1\} \\ \Pi; \Gamma; \Delta \vdash s_1 \Leftarrow \{z_1 : b | (v = \mathbf{T} \wedge (\phi_1[v/x])) \Longrightarrow (\phi[z_1/z])\} \\ \Pi; \Gamma; \Delta \vdash s_2 \Leftarrow \{z_2 : b | (v = \mathbf{F} \wedge (\phi_1[v/x])) \Longrightarrow (\phi[z_2/z])\}\end{array}}{\Pi; \Gamma; \Delta \vdash \mathbf{if}\, v\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2 \Leftarrow \{z : b | \phi\}}\ \mathbf{11}
$$

$$
\frac{\begin{array}{c}u : \tau \in \Delta \\ \Pi; \Gamma \vdash v \Leftarrow \tau\end{array}}{\Pi; \Gamma; \Delta \vdash u := v \Leftarrow \{z : \mathbf{unit} | \top\}}\ \mathbf{12}
\qquad
\frac{\begin{array}{c}\Pi; \Gamma \vdash v \Rightarrow \{z_2 : b | \phi_2\} \\ \Pi; \Gamma \vdash \{z_2 : b | \phi_2\} \lesssim \{z_1 : b | \phi_1\}\end{array}}{\Pi; \Gamma \vdash v \Leftarrow \{z_1 : b | \phi_1\}}\ \mathbf{13}
\qquad
\frac{\Pi; \Gamma, z_1 : b[\phi_1] \models \phi_2[z_1/z_2]}{\Pi; \Gamma \vdash \{z_1 : b | \phi_1\} \lesssim \{z_2 : b | \phi_2\}}\ \mathbf{14}
$$

Fig. 13. Selected MiniSail Typing Rules

function calls) outside of the SMT fragment. On the other hand, rule 11 for if's scrutinises a value and can flow the value into the branches.

Finally, the constraint discipline pays off in rules 13 and 14, where the subtyping relation is used. One type is a subtype of another just when they have a common base type and the first type's constraint implies the second, under the assumption of all the constraints in the context. Since no rule ever introduces a quantifier, we only generate entailments strictly in the SMT fragment.

MiniSail's design is heavily inspired by the observation in Liquid Types [Rondon et al. 2008] that if logical constraints are determined by the actual arguments to a function, there is no need to introduce existential constraint variables. However, we do not need a prepass deriving a simply-typed skeleton. Our bidirectional [Dunfield and Krishnaswami 2013] algorithm is completely syntax-directed, with subtyping checks (the only source of SMT queries) occurring at (syntactically evident) checking/synthesis boundaries.

The original Sail implementation had a Hindley-Milner-style typechecker, mated to a custom arithmetic constraint solver. This codebase was complicated and could not handle many of the constraints generated from the the ARM specification. Sail is now bidirectional, mostly replacing unification with constraint solving. The transition is ongoing: unification still plays a role in the implementation of function calls, and we still allow the declaration of non-argument-constrained quantifiers. Still, performance has dramatically improved: checking a fragment of the ARM spec has gone from 10-15 minutes to under 3 seconds.

The operational semantics of the full language (including tuples and sums) is standard, and can be found in an appendix available online via the Sail website (https://www.cl.cam.ac.uk/~pes20/sail/). The full type safety proof is also in this appendix: the proof is long (due to the presence of dependency) but not fundamentally difficult.

7 VALIDATION

In order to be generally useful, our ISA models need to be well-validated. For this purpose, we run test suites and boot various operating systems above our models by using the emulators generated from them. This also serves to validate the translation from Sail to the various backends. Ideally, one might want to have mechanised proofs about the correctness of the translation w.r.t. a deep embedding of Sail in each of the provers. However, the effort for this would have been prohibitive, especially while the Sail language itself was still evolving. Instead, we follow a testing approach here too. We have used Isabelle's code generation feature to extract an OCaml emulator from the Isabelle model of CHERI-MIPS, which successfully executes the CHERI test suite, albeit slowly. This gives us end-to-end validation for the nontrivial translation pipeline from Sail via Lem to Isabelle, including bitvector length monomorphisation and translation of effects (§4).

ARM We validated our ARM models first by booting Linux on the non-public v8.3 version with system register support (used for the timer, handling interrupts, and controlling the availability of architectural features). This does not directly validate the public version of our ARM model, but as the two are generated in much the same way from the same ultimate sources, it does provide significant confidence. We were able to boot older versions of the Linux kernel, in particular Linux 4.4 (2016). For more recent versions of the kernel, we observed issues with context switching when run above our model. Linux has changed how context switching is handled to unmap kernel memory, and it seems that a page fault that is supposed to happen at a certain point does not occur. This could be due to a bug in the address translation code of our model, in the implementation of a systems feature such as the interrupt controller, or in our tooling. However, the problem seems to be subtle enough to only be triggered in some versions of Linux, and we have not yet fully diagnosed it.

ARM's *Architecture Validation Suite* (AVS) is an extensive set of architectural compliance tests that are used as part of the signoff criteria for ARM-compatible processors. These tests are usually run on systems composed of processors, RAM and a verification device that can be used to monitor the processor's behaviour (e.g. memory accesses and their attributes) and to generate stimulus (e.g. patterns of interrupts). ARM currently runs these tests on an extension of the public ASL specification that adds a particular set of configuration choices for the implementation-defined behaviour, and an ASL specification of the verification device [Reid 2016]. ARM does not currently publicly release the tests, or the configuration or specification of the test device, but we were able to use them to test and debug our translation of the ASL specification.

The tests cover many aspects of the AArch64 architecture including all usermode behaviour (i.e., integer, float and SIMD instructions), and system behaviour (i.e., bigendian support, switching between 32-bit mode and 64-bit mode, memory protection, exceptions/interrupts, privilege levels, security and virtualisation). The tests for usermode behaviour make up 31% of the tests (this is roughly proportional to the fraction of ARM's specification documents and their ASL specification that describes usermode behaviour). Many of the tests for system behaviour explore obscure corners of the architecture, such as whether a memory access should be cacheable or non-cacheable if an operating system marks the page as cacheable but a hypervisor (in which the operating system is running) marks the same page as non-cacheable, or what exception should be signalled if a bus fault occurs during a page table walk. (These are just two of thousands of scenarios that are tested.) The tests consist of over 30 000 test programs and the tests run for billions of instructions.

Our current translation to Sail and our C model generation do not handle certain features of ARM's specification including the AArch32 instruction set, SIMD instructions, multiprocessor support and a small number of instructions added in the v8.3 model, and so we restricted our attention to 15 400 tests that do not rely on these features. Of those 15 400 tests, currently 24 (0.15%) pass on the ASL model but not on the Sail model. 12 of those are floating point failures, due to the square root primitive operation returning a rational number; 8 are exception handling failures due to a particular unallocated exception being misthrown; and the remaining 4 are memory management failures involving marking page table entries dirty. We are working on fixing these issues in the Sail model.

RISC-V We validated the RISC-V model with the seL4 and Linux boots and against the Spike reference simulator (the current platform model of our RISC-V OCaml emulator matches that of Spike). The OCaml emulator is run regularly against the tests in the `riscv-tests` test-suite repository, and passes all tests for integer and compressed instructions for the user, supervisor and machine modes (currently amounting to 181 tests). An official compliance test-suite is under construction by the RISC-V Compliance Working Group, but it has yet to create tests for the 64-bit architecture. We also compare the trace outputs of the Sail model and a version of Spike modified to provide additional execution traces, and to have a more regular I/O and timer interrupt dispatch schedule. Our comparison tool checks that the two simulators execute matching instructions, integer register writes, CSR reads and writes, LR/SC reservation state modifications, and outputs to certain device ports. We have ensured that these traces match on all but one of the above tests. The sole exception is the test for the `breakpoint` instruction, where the Sail model passes the test but the execution trace differs due to the absence of a debug module.

MIPS and CHERI-MIPS To validate these models we ran the CHERI test suite (which also tests MIPS ISA features) and booted FreeBSD-MIPS with a minimal system model consisting of just a write-only UART for console output. Using Sail's C backend and gcc 5.4 on an Intel Core i7-4770K desktop CPU clocked at 3.50GHz the boot reached a shell prompt after about 90 million instructions in less than 2 minutes, averaging about 850 000 instructions per second.

Coverage An executable-as-test-oracle architectural model makes it possible to assess the specification coverage of tests. We did this for the MIPS and CHERI-MIPS models, simply using the `gcov` coverage tool on the compiled C. Booting FreeBSD on the MIPS model touched 84.8% of the lines of generated C. Most unexecuted lines were due to instructions that were not used (e.g. debugging, cache management, fused multiply&add) and exception cases that were not hit, such as reserved instructions. The MIPS-only subset of the CHERI test suite covered 97.8% of the MIPS model, with the uncovered code due to missing tests for MIPS features such as unusual TLB page sizes and supervisor mode that are not used by FreeBSD. Coverage for the CHERI model was 94.8%.

This found a recently introduced instruction that had no tests and highlighted many exception paths that need more testing.

RMEM concurrency integration We integrated our RISC-V ISA model with the RMEM concurrency exploration tool [Pulte et al. 2018], allowing exploration of its relaxed-memory multi-threaded behaviour. For validation, we compared its behaviour on the library of 7251 litmus tests used to develop the RISC-V memory model [RIS 2017, App. A]. They concur on all except 4, due to a discrepancy between the RISC-V memory model and the Spike single-threaded reference simulator: the former allows store-conditionals to fail early before reading any registers, while the latter does not. We currently forbid this, to match traces with Spike. In addition, for emulator performance reasons, the sequential ISA model uses a definition of the JALR instruction that does not allow the write-before-read behaviour of the concurrent specification.

8 MECHANISED PROOF

To evaluate the usability of the generated theorem prover definitions, we proved a nontrivial property of the ARMv8-A specification in Isabelle/HOL. We focus on address translation from virtual to physical memory addresses. This is a critical part of the architecture specification; playing an important role in separating user-space processes from each other and from the operating system. ARMv8-A address translation is also an informative benchmark of the usability of our theorem prover definitions, as it is one of the most complex parts of the most detailed specification we have. The translation table walk function alone consists of over 500 lines of Sail code, not counting various helper functions. It includes a loop for the table walk, does the construction of the physical address from variable-length bitvector slices, reads and writes memory, and exhibits nondeterminism. The latter arises from underspecification that can be refined by implementations. For example, there is a validity check of page table entries that an implementation may choose to perform (potentially faulting) or to ignore. This is "implementation defined" behaviour in the ASL and translated to a nondeterministic choice in our model. Another source of nondeterminism is undefined values. Address translation returns a record containing the output address and other fields such as permission bits. If one of those fields does not make sense in a given situation, such as the device type field for non-device memory, the ASL code sets it to an "unknown" value or leaves it uninitialised. Again, this is translated to a nondeterministic choice of a value in Sail.

Details like these are typically abstracted away in verification projects involving an ISA semantics. This may be essential for reasoning about the ISA semantics in a scalable way, but the underlying assumptions should be made explicit. Proving soundness of an abstraction against our model allows —and requires— us to do this, in terms of the model. As our example, we therefore defined a purely functional characterisation of ARMv8-A address translation in a user-mode setting. Our function read_tables extracts from memory a snapshot of the translation tables (up to four hierarchical levels deep) starting at a given base address, while walk_tables is a partial function that takes a table snapshot and an input address and looks up the corresponding descriptor. The partial function translate_address calls those two, checks the permission bits, and, if all checks succeed, constructs a result record containing an address descriptor with the output address and its attributes, and potentially a descriptor update, if hardware updating of access and dirty bits is enabled. The function update_descriptor writes back the updated descriptor, if necessary.

This characterisation of address translation is quite detailed, but we do make some simplifying assumptions. We assume a setting in 64-bit user mode and not in a "secure" state, which is an isolation feature of the ARM architecture. We also assume that no virtualisation is active, so we have only one and not two stages of address translation. Moreover, we assume that hardware updating of descriptor flags is enabled (the Linux kernel uses this in its default configuration).

Without it, translating an address within a page or block without the access flag set results in a translation fault. Finally, we assume that the MMU is enabled and debug events are disabled. We formalise these assumptions as state predicates. For example, the predicate $\mathsf{HwUpdatesFlags}(s)$ requires that bits 39 and 40 of the TCR_EL1 system register are set. We omit the definitions of these predicates and functions here and refer the Isabelle proof scripts, which can be found online via the Sail website (https://www.cl.cam.ac.uk/~pes20/sail/).

We have proved the following soundness result about our characterisation w.r.t. the original function AArch64_TranslateAddress defined in the model, where $[\![\cdot]\!]$ denotes the lifting from free to state monad mentioned in §4.2, the relation $\approx_{det}$ denotes equivalence of the deterministic parts of address descriptors, ignoring undefined parts, and Value indicates a successful outcome of an expression in the state monad, as opposed to an exception denoted using Ex (where in this case, the preconditions guarantee that there is no exception).

Theorem 8.1. *If*

- $\mathsf{InUserMode}(s) \wedge \mathsf{NonSecure}(s) \wedge \mathsf{MMUEnabled\_EL01}(s) \wedge \mathsf{VirtDisabled}(s) \wedge \mathsf{HwUpdatesFlags}(s) \wedge \mathsf{UsingAArch64}(s) \wedge \mathsf{DebugDisabled}(s)$ *and*
- $\mathsf{translate\_address}(vaddr, acctype, iswrite, aligned, size, s) = r$

then

$$\forall(\mathsf{Value}(r'), s') \in [\![\mathsf{AArch64\_TranslateAddress}(vaddr, acctype, iswrite, aligned, size)]\!](s).$$
$$r' \approx_{det} \mathsf{addrdesc}(r) \wedge s' = \mathsf{update\_descriptor}(r, acctype, iswrite, s)$$

The assumption that the partial function translate_address successfully returns a value implies that all checks have passed and all table entries related to the input address are valid. If one of those checks fails, then the original address translation function returns a record detailing which kind of fault occurred; we do not currently model faulting behaviour in our characterisation.

This means that Theorem 8.1 may not shed light on any potential address translation bug related to the Linux booting issue of §7, as that would involve a page fault. However, our proof did uncover a missing endianness reversal and several potential uses of uninitialised variables in the original ASL code, which have been reported to and confirmed by ARM.

Our Isabelle proof is with respect to the sequential Sail model in the state-nondeterminism-exception monad. We manually stated and proved a loop invariant for the translation table walk, and Hoare triples about various helper functions. This helps reduce the complexity of the main proof, which uses an automatic proof method that iteratively applies the basic proof rules of the Hoare logic and the helper lemmas to derive a precondition for a given postcondition.

9 RELATED WORK

There is extensive work on low-level verification using ISA specifications, as well as language design for ISA description languages, e.g. [Misra and Dutt 2008]. There are also hardware specification languages, e.g. [Choi et al. 2017], optimised for designing and verifying hardware implementations, which addresses a different problem than we do. We focus on ISAs, which specify an envelope of allowed behaviours for processor implementations. As mentioned in the introduction, there exist many smaller partial formal ISA models, usually created for very specific purposes, e.g. capturing the ISA fragments needed for compiler implementation [Dias and Ramsey 2010]. Here we mostly focus on work that involves larger ISA specifications that include some system-level features.

[Degenbaev 2012] presents a model of x86 that includes features such as virtual memory, interrupts, and virtualisation; it does not report validation results of the model, however. The ACL2 X86isa model [Goel et al. 2017] is a hand-written specification of the (64-bit) IA-32e mode of the x86 architecture. It contains a very comprehensive specification of user-mode parts of the architecture, as well as system-level features including paging, segmentation, and a system call interface. Their model has been extensively validated via co-simulation with actual x86 processors. This work represents the most complete public x86 specification to date. Our work differs mainly in targeting different architectures, providing for multiple LCF-family theorem provers (Isabelle, HOL4, and Coq) rather than ACL2, using a dependently typed metalanguage and (validated but not proved) translations from it rather than working entirely within ACL2, and in translating from the vendor-supplied ARMv8-A specification. Our models have sufficient system-feature coverage to boot operating systems, though that is particularly challenging for x86.

L3 [Fox 2012, 2015; Fox et al. 2017] is a well-developed ISA specification language, which like Sail, supports multiple prover targets (HOL4 and Isabelle/HOL), and has existing models for numerous architectures. L3 was a key inspiration in the design of Sail, which differs principally in its more sophisticated type-system (better able to express and check the dependent features found in ASL), its integration with concurrency models, and features to better support direct translation of ASL pseudocode, such as exception handling.

seL4 [Klein et al. 2014] uses a specification of the ARMv7 architecture [Fox and Myreen 2010] to verify binary correctness of all seL4 functions. However, this binary verification is not done for certain machine-interface functions that interact with system-level parts of the architecture, which were originally assumed correct as part of the main seL4 proof. The CertiKOS project [Gu et al. 2016] presents another verified operating system, which defines a machine-model for x86 [Gu et al. 2015] in Coq extended with support for devices and interrupts [Chen et al. 2016]. This machine model is based on the 32-bit x86 subset specified in CompCert [Leroy et al. 2017].

Syeda and Klein [Syeda and Klein 2018] formalise an ARMv7 style memory management unit (MMU) in Isabelle/HOL, with a translation lookaside buffer and multiple levels of page tables. They are able to reason about system-level code in the presence of a TLB, including operating system context-switching. Joloboff et al [Joloboff et al. 2015; Shi 2013] develop a verified instruction set simulator using Coq for the ARMv6 architecture. They compile C code implementing each instruction using CompCert, before proving equivalence between the CompCert instruction set semantics and a model of ARMv6 extracted to Coq from the ARM architecture reference manual PDF. With ARM's release of a machine readable specification [Reid 2017], which we have used, such an extraction process is no longer necessary.

The PROSPER project [Baumann et al. 2016; Guanciale et al. 2016] has extended L3 models of ARMv8 [Fox 2015] with system features sufficient to verify a virtualisation platform including secure boot and a hypervisor. This specification is based on hand-translating the required parts from the ARM architecture reference manuals. In contrast, by basing our ARMv8 model on ASL, we are able to more easily keep track of the constant revisions to the architecture, as well as cover more obscure corner cases in the architecture with improved confidence.

ACKNOWLEDGMENTS

The ARMv8-A modelling work would not have been possible without generous technical assistance from ARM. We thank Kyndylan Nienhuis for proving useful helper lemmas for the Isabelle proof presented in §8. This work was partially supported by EPSRC grant EP/K008528/1 (REMS), ERC Advanced Grant 789108 (ELVER), an ARM iCASE award, EPSRC IAA KTF funding, and ARM donation funding. This work is part of the CTSRD, ECATS, and CIFV projects sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237, HR0011-18-C-0016, and FA8650-18-C-7809. The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

REFERENCES

2017. The gem5 Simulator. http://gem5.org.

2017. QEMU: the FAST! processor emulator. https://www.qemu.org/.

2017. The RISC-V Instruction Set Manual. Volume I: User-Level ISA; Volume II: Privileged Architecture. https://riscv.org/specifications/. 236 pages.

Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in Weak Memory Models. In *Proceedings of CAV 2010: the 22nd International Conference on Computer Aided Verification, LNCS 6174*. https://doi.org/10.1007/978-3-642-14295-6_25

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM TOPLAS* 36, 2, Article 7 (July 2014), 74 pages. https://doi.org/10.1145/2627752

Roberto M. Amadio, Nicholas Ayache, François Bobot, Jaap Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, Randy Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, Ian Stark, and Paolo Tranquilli. 2013. Certified Complexity (CerCo). In *Foundational and Practical Aspects of Resource Analysis - Third International Workshop, FOPARA 2013, Bertinoro, Italy, August 29-31, 2013, Revised Selected Papers*. 1–18. https://doi.org/10.1007/978-3-319-12466-7_1

Andrew W. Appel, Lennart Beringer, Robert Dockins, Josiah Dodds, Aquinas Hobor, Gordon Stewart, and Qinxiang Cao. 2017. Verified Software Toolchain. http://vst.cs.princeton.edu/download/.

ARM. 2017. ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile. v8.2 Beta. 6354 pages.

Christoph Baumann, Mats Näslund, Christian Gehrmann, Oliver Schwarz, and Hans Thorsen. 2016. A high assurance virtualization platform for ARMv8. In *European Conference on Networks and Communications, EuCNC 2016, Athens, Greece, June 27-30, 2016*. 210–214. https://doi.org/10.1109/EuCNC.2016.7561034

Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. 2008. Boomerang: Resourceful Lenses for String Data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, New York, NY, USA, 407–419. https://doi.org/10.1145/1328438.1328487

David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.), Vol. 6806. Springer, 463–469. https://doi.org/10.1007/978-3-642-22110-1_37

Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. 2016. Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 431–447. https://doi.org/10.1145/2908080.2908101

Adam Chlipala. 2013. The Bedrock structured programming system: combining generative metaprogramming and Hoare logic in an extensible program verifier. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. 391–402. https://doi.org/10.1145/2500365.2500592

Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: a platform for high-level parametric hardware specification and its modular verification. *PACMPL* 1, ICFP, 24:1–24:30. https://doi.org/10.1145/3110268

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

Ulan Degenbaev. 2012. *Formal Specification of the x86 Instruction Set Architecture*. Ph.D. Dissertation. Universität des Saarlandes. https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/26394.

João Dias and Norman Ramsey. 2010. Automatically Generating Instruction Selectors Using Declarative Machine Descriptions. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, New York, NY, USA, 403–416. https://doi.org/10.1145/1706299.1706346

Joshua Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 429–442. https://doi.org/10.1145/2500365.2500582

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, Robert Cartwright (Ed.). ACM, 237–247. https://doi.org/10.1145/155090.155113

Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA. In *Proceedings of POPL: the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. https://doi.org/10.1145/2837614.2837615

Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. 2017. Mixed-size Concurrency: ARM, POWER, C/C++11, and SC. In *The 44st Annual ACM SIGPLAN-SIGACT*

Symposium on Principles of Programming Languages, Paris, France. 429–442. https://doi.org/10.1145/3009837.3009839

Anthony C. J. Fox. 2012. Directions in ISA Specification. In *Interactive Theorem Proving – Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings.* 338–344. https://doi.org/10.1007/978-3-642-32347-8_23

Anthony C. J. Fox. 2015. Improved Tool Support for Machine-Code Decompilation in HOL4. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, Christian Urban and Xingyuan Zhang (Eds.), Vol. 9236. Springer, 187–202. https://doi.org/10.1007/978-3-319-22102-1_12

Anthony C. J. Fox and Magnus O. Myreen. 2010. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings.* 243–258. https://doi.org/10.1007/978-3-642-14052-5_18

Anthony C. J. 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, Paris, France, January 16-17, 2017.* 125–137. https://doi.org/10.1145/3018610.3018621

Shilpi Goel, Warren A. Hunt Jr., and Matt Kaufmann. 2017. Engineering a Formal, Executable x86 ISA Simulator for Software Verification. In *Provably Correct Systems.* 173–209. https://doi.org/10.1007/978-3-319-48628-4_8

Kathryn E. Gray, Gabriel Kerneis, Dominic Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. 2015. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proc. MICRO-48, the 48th Annual IEEE/ACM International Symposium on Microarchitecture.* https://doi.org/10.1145/2830772.2830775

Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. 2015. Deep Specifications and Certified Abstraction Layers. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15).* ACM, New York, NY, USA, 595–608. https://doi.org/10.1145/2676726.2676975

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.* 653–669. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu

Roberto Guanciale, Hamed Nemati, Mads Dam, and Christoph Baumann. 2016. Provably secure memory isolation for Linux on ARM. *Journal of Computer Security* 24, 6 (2016), 793–837. https://doi.org/10.3233/JCS-160558

Intel Corporation. 2017. Intel 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4. https://software.intel.com/en-us/articles/intel-sdm. 325462-063US. 4744 pages.

Jonas B. Jensen, Nick Benton, and Andrew Kennedy. 2013. High-level Separation Logic for Low-level Code. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13).* ACM, New York, NY, USA, 301–314. https://doi.org/10.1145/2429069.2429105

Vania Joloboff, Jean-François Monin, and Xiaomu Shi. 2015. Towards Verified Faithful Simulation. In *Dependable Software Engineering: Theories, Tools, and Applications - First International Symposium, SETTA 2015, Nanjing, China, November 4-6, 2015, Proceedings (LNCS)*, Xuandong Li, Zhiming Liu, and Wang Yi (Eds.). Springer, 105–119. https://doi.org/10.1007/978-3-319-25942-0_7

Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A Formally-Verified C Static Analyzer. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015.* 247–259. https://doi.org/10.1145/2676726.2676966

Andrew Kennedy, Nick Benton, Jonas Braband Jensen, and Pierre-Évariste Dagand. 2013. Coq: the world's best macro assembler?. In *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013.* 13–24. https://doi.org/10.1145/2505879.2505897

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM TOCS* 32, 1 (Feb. 2014), 2:1–2:70. https://doi.org/10.1145/2560537

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14).* ACM, New York, NY, USA, 179–191. https://doi.org/10.1145/2535838.2535841

Dirk Leinenbach and Thomas Santen. 2009. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings.* 806–809. https://doi.org/10.1007/978-3-642-05089-3_51

Xavier Leroy. 2009. A formally verified compiler back-end. *J. Automated Reasoning* 43, 4 (2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

Xavier Leroy et al. 2017. CompCert 3.1. http://compcert.inria.fr/.

Junghee Lim and Thomas W. Reps. 2013. TSL: A System for Generating Abstract Interpreters and its Application to Machine-Code Analysis. *ACM TOPLAS* 35, 1 (2013), 4:1–4:59. https://doi.org/10.1145/2450136.2450139

Prabhat Misra and Nikil Dutt (Eds.). 2008. *Processor Description Languages.* Morgan Kaufmann.

Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: better, faster, stronger SFI for the x86. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012.* 395–404. https://doi.org/10.1145/2254064.2254111

Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. 2014. Lem: reusable engineering of real-world semantics. In *Proceedings of ICFP 2014: the 19th ACM SIGPLAN International Conference on Functional Programming.* 175–188. https://doi.org/10.1145/2628136.2628143

Magnus Oskar Myreen. 2009. *Formal verification of machine-code programs.* Ph.D. Dissertation. University of Cambridge, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.611450

Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07).* ACM, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

William Pugh. 1991. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings Supercomputing '91, Albuquerque, NM, USA, November 18-22, 1991*, Joanne L. Martin (Ed.). ACM, 4–13. https://doi.org/10.1145/125826.125848

Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. In *POPL 2018.* https://doi.org/10.1145/3158107

Alastair Reid. 2016. Trustworthy Specifications of ARM v8-A and v8-M System Level Architecture. In *FMCAD 2016.* 161–168. https://alastairreid.github.io/papers/fmcad2016-trustworthy.pdf

Alastair Reid. 2017. ARM Releases Machine Readable Architecture Specification. https://alastairreid.github.io/ARM-v8a-xml-release/.

Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. 2016. End-to-End Verification of Processors with ISA-Formal. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science)*, Swarat Chaudhuri and Azadeh Farzan (Eds.), Vol. 9780. Springer, 42–58. https://doi.org/10.1007/978-3-319-41540-6_3

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08).* ACM, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER Multiprocessors. In *Proceedings of PLDI 2011: the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation.* 175–186. https://doi.org/10.1145/1993498.1993520

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Comm. of the ACM* 53, 7 (July 2010), 89–97. https://doi.org/10.1145/1785414.1785443

Xiaomu Shi. 2013. *Certification of an Instruction Set Simulator.* Theses. Université de Grenoble. https://tel.archives-ouvertes.fr/tel-00937524

Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016.* IEEE Computer Society, 138–157. https://doi.org/10.1109/SP.2016.17

Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is Reasonable Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018).* ACM, New York, NY, USA, 14–27. https://doi.org/10.1145/3167092

Hira Taqdees Syeda and Gerwin Klein. 2018. Program Verification in the Presence of Cached Address Translation. In *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings (Lecture Notes in Computer Science)*, Jeremy Avigad and Assia Mahboubi (Eds.), Vol. 10895. Springer, 542–559. https://doi.org/10.1007/978-3-319-94821-8_32

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. 2016. A new verified compiler backend for CakeML. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016.* 60–73. https://doi.org/10.1145/2951913.2951924

Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2013. CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency. *J. ACM* 60, 3, Article 22 (June 2013), 50 pages. https://doi.org/10.1145/2487241.2487248

Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Nathaniel Wesley Filardo, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Sewell, Stacey Son, and Hongyan Xia.

2018. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7).* Technical Report UCAM-CL-TR-927. University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom, phone +44 1223 763500. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.html

Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015.* 20–37. https://doi.org/10.1109/SP.2015.9

Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: revisiting RISC in an age of risk. In *ISCA '14: Proceeding of the 41st annual international symposium on Computer architecture.* IEEE Press, Piscataway, NJ, USA, 457–468. https://doi.org/10.1145/2678373.2665740

Hongwei Xi. 2007. Dependent ML An approach to practical programming with dependent types. *J. Funct. Program.* 17, 2 (2007), 215–286. https://doi.org/10.1017/S0956796806006216