UNIVERSITY OF CAMBRIDGE

Blind Grading Number: 4485B

ArchSem in Lean: Integrating ISA and Concurrency Semantics

Christopher Lang Queens' College

May 2026

This Dissertation is submitted for
Part II of the Computer Science Tripos

Declaration of Originality

I, the candidate for Part II of the Computer Science Tripos with Blind Grading Number 4485B, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this report, I adhered to the Department of Computer Science and Technology AI Policy. **I am content for my report to be made available to the students and staff of the University.**

Date: 13th May 2026 Signed: Christopher Lang

Proforma

Candidate number (BGN)	`4485B`
Project Title	ArchSem in Lean: Integrating ISA and Concurrency Semantics
Examination	Computer Science Tripos - Part II, 2026
Word-count	11060[1]
Code line count	3581[2]
Project Originator	Thibaut Pérami and Peter Sewell
Day-to-Day Supervisor	Thibaut Pérami (and Léo Stefanesco)
Marking Supervisor	Peter Sewell
Ethical Approval	None

Original Aims of the Project

At the frontier of research into formalising multi-core CPU architecture specifications is *ArchSem*, a framework for writing machine-verified proofs against an architecture. Since ArchSem's inception in 2021, it has been developed in the Rocq theorem proving language, but limitations in Rocq's ecosystem, language design and performance have seriously hindered ArchSem's progress. **My project aims to port ArchSem to Lean**, a newer theorem proving language that appears to solve many of the issues faced in ArchSem's Rocq implementation.

Work Completed

All success criteria were met and exceeded. ArchSem's interface between ISA and concurrency semantics was ported to Lean, along with an Arm-A relaxed memory model. A CLI for parsing and running litmus tests empirically demonstrated correctness against Herd's [1] architecture specification and an original and substantial proof was written for a correctness property of the memory model. My Lean implementation of ArchSem requires many thousands fewer lines of boilerplate, makes development easier with human-readable error messages and boasts a significant performance improvement over the existing Rocq implementation.

Special Difficulties

None.

[1] Using the Typst *wordometer* package `v0.1.5`.

[2] Since my project involves modifying other existing repositories, it is not easy to get an exact line count. I arrived at this number by summing 2996 lines for the ArchSem-Lean main repository, 415 lines modified in lean-sail, and 170 lines of modifications to ArchSem-Rocq.

Contents

1. Introduction

ArchSem [2] is a framework for formally defining an *architecture specification* in the Rocq theorem proving language, suitable for formal verification of architecture properties or behaviour of specific programs. My project aims to port ArchSem to the Lean theorem proving language in order to benefit from Lean's modern language features and provide a version of ArchSem for the growing Lean community.

1.1. Previous Related Work

Historically, architecture specifications have been written in large prose documents, occasionally accompanied by pseudocode. Recent Arm reference manuals exceed 16,000 pages [3] and are a challenge for many PDF readers, let alone as a foundation for formal verification. Most previous work on formalising architecture has focused on one of two components: the *instruction set architecture* (ISA), defining the behaviour of instructions in the context of a single thread, or the *memory concurrency semantics*, which defines how instruction effects are propagated between threads.

ASL [4] and more recently *Sail* [5] have made considerable progress on **formalising ISA semantics** by expressing the fetch-decode-execute pipeline in imperative domain-specific languages. Sail is equipped with various *backends* which convert from the DSL into general-purpose programming languages for execution or formal verification.

The **memory concurrency semantics** of mainstream architectures is also well understood. There has been particular success in the creation of *executable test oracles* [1, 6, 7]: software which, given an initial multiple-processor architecture state, can compute all architecturally allowed executions until some termination condition is reached. These oracles have been used to test correctness of hardware and have found bugs in real processors [8].

ArchSem [2] integrates both ISA and concurrency semantics in the Rocq [9] theorem proving language, thus building a full architecture specification suitable both as an executable test oracle and as the foundation for machine-checked proofs. Unlike previous formalisations [10, 11], ArchSem can ingest Sail ISA semantics and supports multiple memory models thanks to the generic *ArchSem interface* between ISA and memory semantics.

1.2. Success Criteria

To evaluate the suitability of Lean for ArchSem, it is important that I implement both ISA and concurrency semantics. Then I will use the resulting architecture specification to evaluate all architecturally allowed behaviour for some small, multithreaded *litmus tests* and compare the results against the existing ArchSem-Rocq. This will demonstrate correctness of my implementation at least in common cases.

Success Criteria:

1. Model CPU architecture semantics in Lean by combining one ISA semantics with at least one relaxed concurrency model.
2. Demonstrate that at least three small litmus tests have equivalent behaviour in both the existing Rocq ArchSem implementation and the new Lean implementation.

1.3. Motivation

Reduced boilerplate. Of the 22,054 lines of Rocq code in ArchSem, 7,933 are implementing functionality that exists in Lean's standard library[3], including bit vector types, basic simplification tactics and a syntax for monadic programming. There are also 784 lines of OCaml boilerplate required for extracting Rocq into OCaml for the ArchSem command-line interface, because Rocq is not equipped for general-purpose functional programming. Lean on the other hand has good support for IO operations and is suitable for both implementing a CLI and defining a formal architecture specification.

Lean's centralised ecosystem of mathematics. When writing a machine-checked proof, one often wants to make use of well-known theorems without re-implementing them from scratch. Lean's ecosystem of mathematical formalisations, centred around Mathlib [12], is significantly less fragmented than Rocq's. Figure 1 shows how Lean's proof ecosystem has quickly caught up with Rocq's, especially since the release of Lean v4.0.0 in 2023. A Lean implementation of ArchSem would enable proofs about architecture specifications to be written in this rapidly growing ecosystem.

The Lean community. I aim to provide the ArchSem framework to the rapidly growing Lean community so that writing architecture proofs is accessible to a wider range of developers. Figure 2 shows the recent surge in development activity on Lean mathematical libraries compared to Rocq.

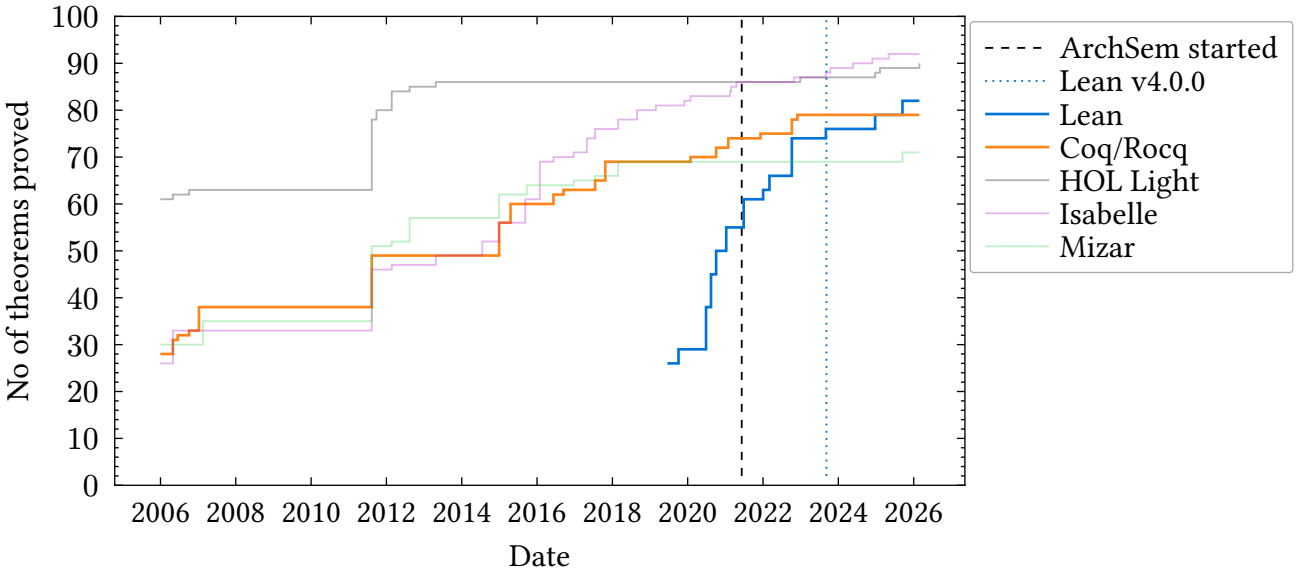


Figure 1: A plot of historical data scraped from a page maintained by Freek Wiedijk at https://www.cs.ru.nl/~freek/100 [13, 14] using Internet Archive's Wayback Machine. It shows the number of theorems proved in different systems from a fixed list of 100 important theorems.

[3]These files are `CArith.v`, `CBase.v`, `CBitvector.v`, `CBool.v`, `CDestruct.v`, `CExtraction.v`, `CInduction.v`, `CList.v`, `CMaps.v`, `CMonads.v`, `Common.v`, `COption.v`, `CResult.v`, `CSets.v`, `CSimp.v`, `CVec.v`, `dune`, `GRel.v`, `HVec.v`, `Options.v`, `StateT.v` at `cb186e471017cce0`.

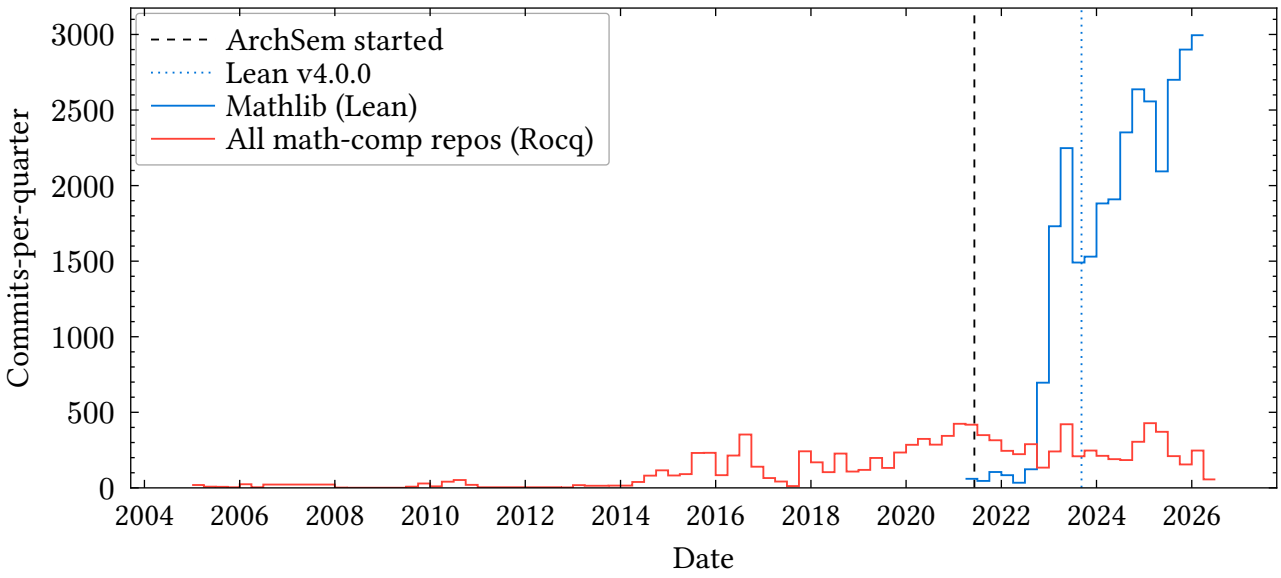


Figure 2: Quarterly git commit frequencies for Lean4′s Mathlib against the sum of 16 math repositories hosted under the Rocq "Mathematical Components" GitHub organisation https://github.com/math-comp: `Abel`, `Coq-Combi`, `algebra-tactics`, `analysis`, `bigenough`, `cad`, `dioid`, `finmap`, `hierarchy-builder`, `math-comp`, `mczify`, `multinomials`, `newtonsums`, `odd-order`, `real-closed`, `trajectories`.

2. Preparation

This chapter presents background information relevant to my project, conducts a requirements analysis, and justifies the tools used and software engineering approach taken. Many aspects of the project go beyond Part IB content, including the relaxed memory models of Section 2.2, the use of theorem proving languages in Section 2.7, and the free monad discussed in Section 2.6 along with its categorical justification in Appendix A.

2.1. Relaxed-Memory Concurrency

The *memory consistency model* of a processor architecture defines how memory updates are propagated between hardware threads. This section introduces *litmus tests* and discusses a small multithreaded program that exhibits different behaviour depending on the memory model.

2.1.1. A Small Example

The need for a memory consistency model arises when multiple threads simultaneously access the same region of memory. Consider the multithreaded program shown in Figure 3 (left). Assume that `x` and `y` are unique, non-overlapping memory addresses. Thread 0 writes an immediate `1` to both `x` and `y` while Thread 1 reads from the same two locations in reverse order. The race condition means that the values read by Thread 1 depend on how the CPU schedules the instructions and how the memory interconnect propagates Thread 0's writes to Thread 1. But what values is Thread 1 architecturally allowed to read from `[x]` and `[y]`?

MP — Arm

Initially all memory and registers 0	
`Thread 0`	`Thread 1`
`MOV [x], #1; (a)`	`LDR R0, [y]; (c)`
`MOV [y], #1; (b)`	`LDR R1, [x]; (d)`
Final: `1:R0=1; 1:R1=0;`	

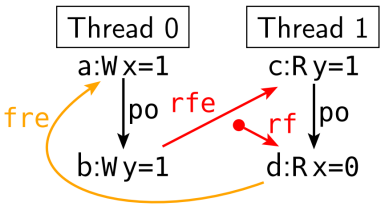


Figure 3: The *MP* litmus test (left) and its candidate execution graph (right)[4].

Clearly `(R0=0, R1=0)` and `(R0=1, R1=1)` are possible: the CPU could choose to schedule all instructions of one thread before the other. And `(R0=0, R1=1)` is allowed if the instructions are interleaved in the order `(a),(c),(d),(b)`. But is there an architecturally allowed execution in which Thread 1 reads `(R0=1, R1=0)`? The reader may wish to pause and consider this question before reading on.

The answer: it depends on what *memory consistency model* our processor conforms to. On Arm and RISC-V, speculative execution could reorder when instructions from a thread reach memory, making the `(R0=1, R1=0)` outcome possible. But the stricter x86 memory model forbids this outcome.

2.1.2. Litmus Tests

The scenario we saw in Figure 3 is an example of a *litmus test*: a small piece of multi-threaded code with a final condition that may or may not be reachable depending on the memory model. A *candidate execution* is a description of how a litmus test might be executed. Candidate executions are often represented as directed graphs of various ordering relations between memory operations.

[4]Candidate execution graph (right) from Sewell et al. [15].

For example, Figure 3 (right) shows the candidate execution graph that depicts how Thread 1 could read `(R0=0, R1=1)` in Arm's memory model. It contains three types of edges:

- The *program order* (`po`) relations in black denoting the order of instructions as they appear in a single thread.
- The *read-from* (`rf`) relations in red denoting the source of a value obtained from a memory read.
- The *from-read* (`fr`) relation in orange which in this case indicates that write *(a)* occurs after the (initial) write which *(d)* read from.

The "`e`" *external* suffix as in `fre` and `rfe` denotes that the relation is between two different threads. For a complete description of the relations in candidate execution graphs, see Alglave et al. [1]. The reader does not need to understand the precise meaning of all ordering relations, only that there exist various types which describe how a litmus test might be executed.

2.2. Formal Models of Memory Concurrency Semantics

Formal models of memory concurrency semantics can largely be divided into *axiomatic* models, which define allowed executions as a predicate on candidate execution graphs, and *operational* models, which use the techniques of operational semantics to formalise the behaviour of memory.

2.2.1. Axiomatic Models

In discussing litmus tests, we saw that candidate executions can be expressed as a graph of memory operations and edges labelled with ordering relations. It is from this representation that the *axiomatic* model naturally arises. Such a model defines the allowed behaviours of the memory model by a predicate on candidate execution graphs. For example, the x86-*TSO* model [16] forbids candidate executions containing a cycle consisting of read-from and from-read edges (such as the one in Figure 3). Arm's relaxed model allows a cycle of these edge types, but forbids a cycle of program order and read-from edges.

An axiomatic model can be expressed using the Cat language [1]. In this language, terms are binary relations on the candidate execution graph and set-theoretic operators are provided such as intersection (`&`) and union (`|`). A cycle of a specific binary relation can be forbidden with the `acyclic` command. A simplified x86 TSO model is shown in Listing 1 for flavour, though it is not important to understand in detail.

```
let pos = po & loc (* program order memory accesses to the same address *)
acyclic pos | rf | co | fr (* enforce coherence *)

let obs = rfe | coe | fre (* observed-by relation *)
let lob = po \ ([W];po;[R]) (* locally-ordered-before relation *)
let ob = obs | lob (* ordered-before relation *)
acyclic ob (* enforce no cycle in the ordered-before relation *)
```

Listing 1: Simplified x86 TSO CAT model without fences or atomics by Sewell et al. [15].

Axiomatic models can concisely describe an architecture's memory model but because the rules are so far abstracted away from an executable machine, it is not easy to determine whether a microarchitectural implementation conforms to an axiomatic model.

2.2.2. Operational Models

We can equivalently specify the memory model with an operational semantics designed in a way that mimics the microarchitecture of a real implementation. The x86 *total store ordering* (TSO) model is particularly suited to an operational semantics. By placing each write in a per-thread, order-preserving store buffer and letting the contents of these buffers propagate to memory at an undetermined later time, we can capture the semantics of x86-TSO relatively simply [17].

But designing an operational semantics for Arm and RISC-V is significantly more challenging because these architectures allow reads and writes to be speculated out-of-order. I will discuss two successful approaches: the *Flat model* [18] and the *promising model* [19]. In Section 3.4, I will go into more detail about my Lean implementation of the promising model.

The Flat model expresses out-of-order and speculative execution explicitly by maintaining a tree of partially completed instructions for each thread [18]. The root of the tree is the thread's first instruction and children are successors in the control flow graph. On each operational step, we non-deterministically choose one of the instructions to make progress on (subject to ordering requirements) or to fetch a new instruction. By modelling speculative execution explicitly, the Flat model can closely resemble the microarchitecture but at the cost of greater complexity compared to the more abstract *promising model*.

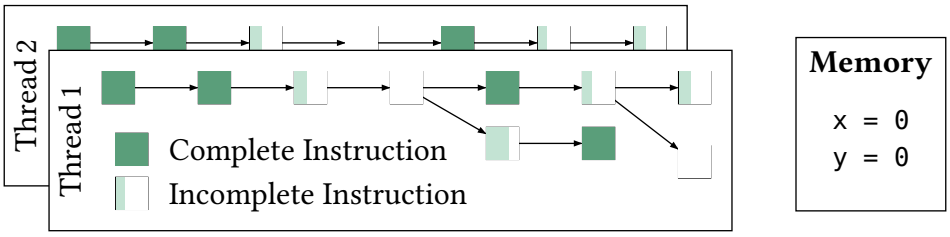


Figure 4: The *Flat* operational relaxed memory model.

The promising model [19] instead chooses to execute instructions in order. It captures speculative executions of reads by recording a history of all writes and allowing reads to access older points in history. Speculative execution of writes is modelled by *promising*: a thread is allowed to *promise* a write if that write could be made by some sequential execution of the thread's instructions starting at the current processor state. When a thread makes a promise, it must later be fulfilled (i.e. the write must actually take place) otherwise the non-deterministic execution is discarded. On each operational step, the *naive promising model* non-deterministically promises a write or executes an instruction. Pulte et al. prove that all naive model executions are equivalent to some *promise-first execution* in which first all promises are made and then all instructions are executed [19]. Hence, we can write an equivalent *promise-first* model which reduces unnecessary non-determinism by making all promises before executing instructions.

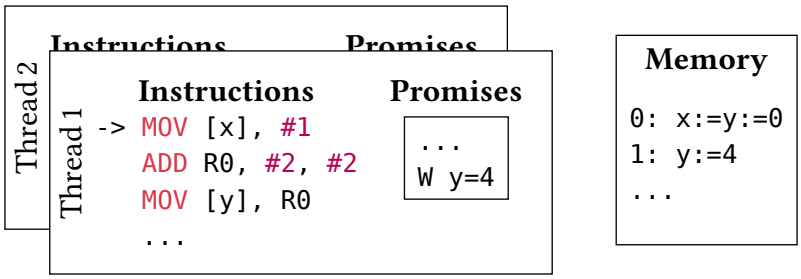


Figure 5: The *promising* operational relaxed memory model.

2.3. Sail ISA Specification

To complete an architecture specification, we need a model of the sequential execution of a single hardware thread: the *ISA specification*. Sail [5] is a language designed for the purpose of formally defining an ISA specification by expressing the fetch-decode-execute lifecycle of an instruction in a custom imperative language equipped with a lightweight dependent type system. Sail *backends* convert Sail into other languages for sequential execution (e.g. C or OCaml), for documentation or as a theorem prover definition.

Léo Stefanesco has recently been developing a Sail-to-Lean backend which converts Sail ISA semantics into a state monad over sequential architecture state in Lean. This makes it unsuitable for reasoning about multithreaded programs. It also does not support larger architecture specifications such as full versions of Arm-A because of a limitation in the current Lean runtime that prevents inductive types with more than 244 constructors[5].

Sail defines how a sequential processor executes a single instruction. However, this does not capture intra-instruction parallelism that appears in some architectures. Arm-A's load pair `LDP` instruction issues two loads in parallel, but its Sail implementation is forced to put one load before the other. As a result of Sail's lack of support, current ArchSem also does not support intra-instruction parallelism which significantly simplifies the ArchSem interface.

2.4. Executable Test Oracles

Suppose we wish to empirically validate that a processor conforms to a memory concurrency model. We could generate a large number of small litmus tests demonstrating a wide range of architectural behaviour, run these tests on hardware for a large number of iterations and record the final state of registers and memory after each run. If we observe any results that are not explainable by some architecturally allowed execution according to the memory model then we can conclude our hardware is not sound with respect to the model. An *executable test oracle* is a class of software predating ArchSem, that for a given litmus test and architecture specification will produce a set of allowed final states.

[5]See https://github.com/leanprover/lean4/issues/8930.

This section gives an overview of three existing executable test oracles, setting the stage for introducing ArchSem which, in addition to providing a test oracle, serves as a foundation for mathematical proof of architectural properties.

Herd7 is an OCaml command line tool that can produce a set of allowed behaviour for a litmus test under a specific axiomatic memory concurrency model provided in the Cat language [1]. It is mature and well-trusted: the Isla paper [6] argues its own correctness by showing it behaves the same as Herd on a large corpus of litmus tests. Herd primarily uses custom hard-coded instruction semantics for subsets of Power, Arm, x86 and x86-64, but there is ongoing work to support interpreting ASL ISA semantics [20]. Hard-coded ISA semantics are not easy to validate and increase the effort required to maintain existing architectures or add support for more.

Isla-axiomatic [6] is an axiomatic test oracle using Sail ISA semantics. The Z3 SMT solver [21] is used to find candidate executions consistent with both the Cat axiomatic memory model and a symbolic execution of Sail ISA semantics.

Rmem [7] implements a number of operational concurrency models over Sail ISA semantics in the Lem specification language [22]. In addition to computing a set of allowed final outcomes, we can step through a litmus test interactively choosing which non-deterministic path to take. This feature, unique to operational models, is useful for software developers in trying to understand a concurrency model. But implementing an operational concurrency model for a new architecture is a time-consuming and often error-prone task (we will discover in Section 3.4 that ArchSem-Rocq's promising model had bugs).

2.5. ArchSem

The executable test oracles that we have seen integrate ISA semantics with relaxed memory models. Given a small litmus test, they produce the set of architecturally allowed behaviours. ArchSem, by Pérami et al. [2], provides this same capability, but it is more ambitious: by implementing the executable model in a theorem proving language, it provides a mathematical model of architecture semantics suitable for machine-checked proofs about both architectural properties and specific programs. For example, the soundness of the Arm virtual memory abstraction has been proved with help from ArchSem [2]. ArchSem supports both operational and axiomatic models within the same framework which in theory could be used to prove the equivalence of axiomatic and operational models right down to their implementation.

At the core of ArchSem is the *ArchSem interface* that sits between ISA and concurrency semantics. The interface consists of an *architecture typeclass*, which fixes types and constants shared between the ISA and memory model such as address sizes and register types, and an *instruction monad*, which encodes instruction semantics generated from Sail. In the next section we shall see how the instruction monad is defined as a free monad over instruction effects.

2.6. Monads

A monad is a construction that can be used to represent effectful operations in a functionally pure programming language. We are interested in using monads to express ISA semantics because the execution of an instruction can be thought of as an operation having side-effects on the thread state and memory model.

2.6.1. State Monad

The *state monad* over some state type σ, represents computations that have a side-effect on some global state with type σ. A side-effect on σ is modeled as a function from the current state to the updated state and some return value $\sigma \to (\alpha \times \sigma)$. This state monad could be used to express ISA semantics as a function from an architecture state (including memory and register files) to the architecture state after an instruction is executed. Such a function would implicitly contain the memory-model of our architecture.

2.6.2. Free Monad

The idea behind ArchSem's use of the *free monad* is to define ISA semantics without fixing the memory model, thus postponing implementation of the memory model. We first define a set of *instruction effects* that the ISA is allowed to produce as side-effects. Instruction effects might include a memory read or memory write effect, a register access, or issuing of memory barriers. The free monad over a set of instruction effects can be thought of as a "continuation tree", where the root node is the first effect produced, and for each value that the effect could return, there is a branch to the next effect that will be produced.

Once ISA semantics are expressed as a free monad, we can interpret the instruction effects into a monad specific to our memory model (e.g. a state monad over our memory model's architecture state). Factorising instruction and memory semantics through the interface of the free monad makes it possible to write proofs about the ISA or memory model independently of each other. Decoupling the two also makes development easier, as one can think about the ISA semantics or memory model in isolation from the other.

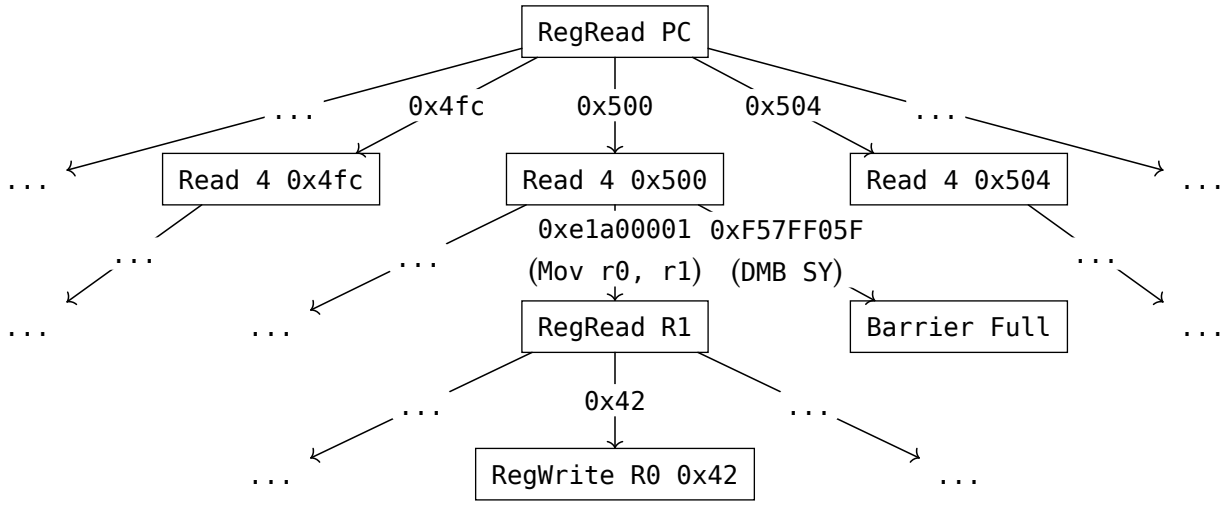


Figure 6: A portion of the *continuation tree* for the free monad representing the ISA of a simplified Arm-like instruction set.

In Appendix A, I have written a justification for ArchSem's use of the free monad from the perspective of category theory. It also explains what is "free" about the free monad.

2.7. The Lean Theorem Proving Language

Lean4 [23] is a dependently typed functional programming language and theorem proving language. It is based upon the same *calculus of constructions* [24] logical system as Rocq. In this section, I briefly introduce some fundamental concepts of Lean.

2.7.1. First Class Types and Type Universes

In Lean, types are first class objects which can be passed to functions and stored in datatypes.

```
-- `#check` is used to infer and display the type of an expression.
#check 42 -- 42 : Nat
#check true -- true : Bool
-- A function that takes and returns `Type` arguments.
def pickType (x : Bool) (α β : Type) : Type := if x then α else β
```

Types are even themselves typed. `Type` is a first class object of type `Type 1`, and `Type 1` is a first class object of type `Type 2`, etc. From this arises an infinite hierarchy of *type universes* which are necessary to avoid logical contradictions in the same class as Russell's paradox[6].

```
#check Nat    -- Nat : Type
#check Type   -- Type : Type 1
#check Type 1 -- Type 1 : Type 2
```

2.7.2. Propositions and Proofs

Both Lean and Rocq support machine-checked proof by exploiting the Curry-Howard correspondence in a logical system called the *calculus of constructions* [24]. Lean has a type of all propositions, `Prop` with members such as `True`, `False` and `∀ (x : Nat). x ≤ 0`. Propositions can be combined with functions such as `And` or `Or` to build larger propositions.

```
#check True   -- True : Prop
#check False  -- False : Prop
#check And    -- And (a b : Prop) : Prop
#check Or     -- Or (a b : Prop) : Prop
#check Nat.le -- Nat.le (n : Nat) : Nat → Prop
#check 1 ≤ 0  -- 1 ≤ 0 : Prop
```

Propositions in `Prop` are themselves types inhabited by proofs of the respective proposition. By combining axioms such as `True.intro` with rules for logical deduction such as `Or.intro_right`, we can constructively build complex proofs. The following example shows constructive proofs for $\text{False} \vee \text{True}$.

```
-- `True.intro` is the (axiom) proof for `True : Prop`.
#check True.intro -- True.intro : True
-- `Or.intro_right` lets us deduce `a ∨ b : Prop` from a proof `h` of `b : Prop`.
#check Or.intro_right -- Or.intro_right {b : Prop} (a : Prop) (h : b) : a ∨ b
-- By combining the two, we can prove `False ∨ True`.
theorem : False ∨ True := Or.intro_right False True.intro
```

By the Curry-Howard correspondence, we see that logical implication can be encoded as a function between propositions. A proof of the implication $P \Rightarrow Q$ is nothing but a function that takes a proof for P and produces a proof for Q.

```
#check False → True -- False → True : Prop
```

[6] When Martin-Löf was developing dependent type theory in 1971, he initially proposed a single type universe level with `Type : Type` [25]. But in the following year Jean-Yves Girard proved it led to a mathematical inconsistency by using an extension of Burali-Forti paradox [26–28]. Russell's paradox alone is not technically enough to draw a contradiction from `Type : Type` due to Lean's proof normalisation [28].

Universal quantification can similarly be expressed as a function. The statement $\forall x \in T.P(x)$ is nothing but a function that takes arbitrary `x : T` and for each produces a proof for `P(x)`.

```
#check ∀ (x : Nat), 0 ≤ x -- ∀ (x : Nat), 0 ≤ x : Prop
#check Nat.zero_le        -- Nat.zero_le (n : Nat) : 0 ≤ n
theorem : ∀ (x : Nat), 0 ≤ x := fun (x : Nat) => Nat.zero_le x
```

2.7.3. Tactics

Lean offers *tactics*, a syntax for incrementally building proofs. While the user is writing a proof in tactic mode, the LSP server shows the user the remaining proof goals and premises available at the current stage of the proof under the cursor in the form of a Horn clause. An example proof by induction is shown below for flavour, though it is not necessary for the reader to understand in detail. The cursor is placed just before the inductive case concludes, so the goal view contains information about this stage of the proof. In the goal view, the proof assumptions and remaining proof goal are shown, separated by the turnstile ⊢. The final line of the proof, `rw [ih]`, concludes by trivially applying the inductive assumption `ih`.

Lean

```
/--
The sum of the lengths is equal to
the length of the concatenation.
-/
theorem List.append_length (a b : List α)
    : a.length + b.length = (a ++ b).length
  := by
  -- Proceed by induction on `a`.
  induction a with
  | nil => -- base case
    rw [length_nil]
    -- ⊢ 0 + b.length = ([] ++ b).length
    rw [Nat.zero_add]
    -- ⊢ b.length = ([] ++ b).length
    rw [nil_append]
  | cons head tail ih => -- inductive case
    -- ⊢ (head :: tail).length + b.length
    --     = (head :: tail ++ b).length
    simp only [length, cons_append]
    rw [Nat.add_assoc]
    rw [Nat.add_comm 1, ←Nat.add_assoc]
    -- CURSOR HERE
    rw [ih]
```

Goal view

```
case cons
α : Type u_1
b : List α
head : α
tail : List α
ih : tail.length + b.length = (tail ++ b).length
⊢ tail.length + b.length + 1 = (tail ++ b).length + 1
```

2.8. Requirements Analysis

The purpose of ArchSem is to provide a framework suitable for formal verification of concurrent architecture properties. In porting ArchSem, I must demonstrate this same capability for at least one architecture in Lean so that I can meaningfully evaluate the suitability of Lean for ArchSem. The ISA semantics I have chosen to use is Sail's *Tiny-Arm* specification[7]. This simplified Arm-like ISA was written to be used for developing new Sail features without re-compiling a full Arm version. The limitations of the Sail-to-Lean backend discussed in Section 2.3 prevent us from using the full Arm ISA semantics. I will implement the promising memory model [19] for Tiny-Arm to complete

[7] sail-tiny-arm https://github.com/rems-project/sail-tiny-arm

the Lean architecture specification. This will enable a direct comparison to be made with ArchSem-Rocq's promising model for Tiny-Arm.

To arrive at our final design requirements, I consider the needs of four key stakeholders:

- The Lean Community to whom we are providing a **proof-ready** version of ArchSem with some assurance of **correctness**.
- Hardware engineers may use ArchSem-Lean as an executable test oracle so it must be **executable** on litmus tests.
- ArchSem-Rocq's main developer, Thibaut Pérami, and myself may wish to add additional memory models or ISA semantics to ArchSem-Lean, so we require that ISA semantics be **memory model generic** and generated from **Sail ISA semantics**. To enable a direct comparison to be made with ArchSem-Rocq, I will implement a **promising memory model**.
- Léo Stefanesco, who maintains the Sail-to-Lean backend wishes to avoid large dependencies so that recent Lean-nightly builds can be used without waiting for dependencies to update (**portability**).

Hence, our design requirements:

Functional Requirements:

1. **Sail ISA semantics**. ArchSem-Lean should use ISA semantics generated from Sail's *Tiny-Arm* specification.
2. **Promising memory model**. The promising memory-model [19] should be implemented to complete the *Tiny-Arm* architecture specification.
3. **Executable**. The ArchSem-Lean Tiny-Arm architecture specification should be executable on small litmus tests. Given an initial multiprocessor state and termination condition, it should be able to evaluate all architecturally allowed final states.
4. **Correctness**. The executable Tiny-Arm architecture specification must show equivalent behaviour to the Rocq implementation for at least three small litmus tests.

Non-Functional Requirements:

5. **Proof-ready**. ArchSem-Lean should facilitate writing machine-checked proofs about the Tiny-Arm architecture specification.
6. **Portability**. ArchSem-Lean should not require large dependencies such as Mathlib so that we can use the latest Lean-nightly build without waiting for dependencies to update.
7. **Memory model generic**. The Sail-to-Lean backend should generate the Tiny-Arm ISA semantics for Lean without enforcing a specific memory model.

2.9. Tools Used

Development environment. The project requires me to work with various specific versions of Lean4, Sail, OCaml, Rocq, Herd7, Rmem, Isla and other related software. To isolate this from my host OS, all development happens on an *Incus* container running Debian 12. Later in the project, I found myself exhausting the 16GiB of RAM on my personal machine by having multiple Lean LSP instances running simultaneously. Therefore, I moved the container to a Google Cloud instance equipped with 64GiB RAM.

Backup and version control. A cron job running every 15 minutes backs up all development files to *Backblaze B2* using *Restic*, an encrypted differential backup software. I frequently backup to my second replica at *rsync.net* using *Unison*, chosen to have a different software stack and physical location to the first replica. I used my personal GitHub account and the REMS-project GitHub organisation as Git remotes. In addition to my laptop as the primary replica, this is 4 replicas managed by 4 organisations in at least 3 physical locations on 2 media types. I conducted one recovery drill in which I restored all files from the Backblaze replica onto a second laptop.

Editor for interactive theorem proving. My preferred text editor, *Kakoune*, did not support the goal view and Unicode entry mechanism necessary for writing Lean code. I wrote a text-editor plugin to enable user-friendly unicode entry and upstreamed my changes to Kakoune-lsp server to support reading the goal view information from the Lean4 LSP server. A screenshot of my text editor plugin is shown in Figure 7.

Figure 7: A screenshot showing the Kakoune text editor editing a Lean proof (left) with help from the goal view (right) which is served to the user thanks to my Kakoune-lsp PR.

2.10. Software Engineering Approach

Development methodology. Development can roughly be divided into two phases. First I focused on learning the prerequisite theoretical concepts covered in this chapter. Then after approximately 5 weeks, I transitioned to an Agile development workflow having frequent in-person meetings with subject experts. To ensure the project completed on time I prioritised work that required collaboration, for example the work on the Sail-to-Lean backend with Léo Stefanesco.

Testing and continuous integration. I used GitHub's continuous integration for both unit and regression tests. These empirical tests are supplemented with the machine-verified proofs, written in Section 3.6, which are also checked in continuous integration.

Code style and documentation. I use the Mathlib conventions [29] except for in the Lean-Sail backend where I respect Sail naming conventions such as using snake_case for term-level attributes. Over 175 documentation comments were written atop function or type definitions. In many places I made identifiers more descriptive than their ArchSem-Rocq counterparts.

Discussion with the Lean community. Throughout the project, I communicated with the wider Lean community via GitHub and the Lean Zulipchat to discuss topics such as non-standard behaviour of the Lean4 LSP server, Lean4 UX issues and limitations of the standard library's intensional hash map.

Software license. Pérami et al.'s ArchSem-Rocq and most other projects released under the rems-project, such as Sail and Rmem use the BSD 2-clause License. However, Lean4 libraries tend to use the Apache 2.0 license. To be maximally compatible with both ecosystems, I dual license ArchSem-Lean under the user's choice of BSD 2-clause or Apache 2.0. Both licenses have been approved by the Free Software Foundation [30] and Open Source Initiative [31, 32] as open-source and GPL-compatible licenses. I use the Free Software Foundation's *reuse* [33] command line program to automatically manage machine readable license comments.

2.11. Starting Point

Theorem provers. Apart from briefly reading the first few sections of a HOL-lite tutorial and a Lean4 tutorial, I had no experience in any theorem proving languages or in writing machine-checked proofs. Learning to write Lean and use its proof automation was a particularly challenging aspect of the dissertation.

ArchSem. ArchSem-Rocq is an existing software which I port to Lean.

Sail. My project makes use of the existing Sail Tiny-Arm ISA specification and modifies an existing Sail-to-Lean backend (details in Section 3.2).

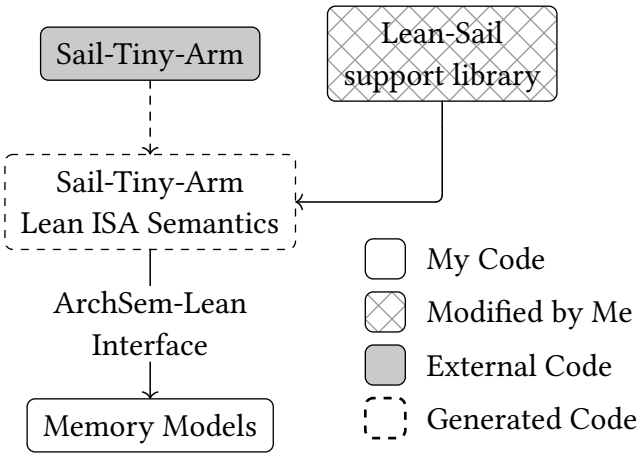
3. Implementation

This chapter discusses my implementation of ArchSem-Lean, which enables full architecture proofs to be written in Lean for the first time. The automatic population of ArchExtra fields described in Section 3.1.4 and the fuel monotonicity proof of Section 3.6 represent original developments which do not exist in ArchSem-Rocq. The chapter ends with a repository overview in Section 3.7.

3.1. The ArchSem Interface

The ArchSem interface consists of the architecture typeclass and a free monad of instruction effects expressing the ISA semantics. This section describes how I modified the Lean-Sail support library to implement the ArchSem interface. In the next section we will then see how the Sail-to-Lean backend was updated to conform to this updated interface.

The existing Lean-Sail support library defines the instruction monad as a state monad over sequential architecture state because it is concerned only with reasoning about single-threaded architectures. The support library also defines an architecture typeclass that is similar to the ArchSem typeclass as it is implemented in Rocq. The Sail-to-Lean backend generates code that instantiates the architecture typeclass and instruction monad with the architectural constants and ISA semantics defined in Sail.



To build the ArchSem interface in Lean, I must change the instruction monad to a free monad over instruction effects. This makes the generated ISA semantics memory-model-agnostic, and we can later implement a relaxed memory model for a multithreaded architecture using the instruction monad.

7. **Memory model generic**. The Sail-to-Lean backend should generate the Tiny-Arm ISA semantics for Lean without enforcing a specific memory model.

3.1.1. The Free Monad

This subsection contrasts two implementations of the free monad in Lean: the mathematically elegant *freer monad* [34] and the ArchSem-Rocq version, which avoids increasing the universe level at the cost of some expressiveness.

The freer monad uses effects defined under signature `Type u → Type v`, a higher order function which takes a type to the type of effects which return that type. This results in a free monad definition that must live in a universe level at least one higher than `Type u`, the universe level of the effect return types. ArchSem-Rocq was initially written using this style of free monad, but it resulted in many universe inconsistency errors thrown by the compiler that were difficult to debug. These errors were a result of Rocq's default use of *monomorphic* definitions for which the inferred universe levels are fixed and may not change per-invocation like a universe polymorphic definition

would. A Lean implementation of the freer monad and an example definition of effects it might use are below:

```
-- The freer monad, annotated with universe levels.
inductive FreeM.{u, v, w} (eff : Type u → Type v) (α : Type w)
    : Type (max (u+1) (max v w))
  | pure (a : α) : FreeM eff α
  | impure {ι} (a : eff ι) (f : ι → FreeM eff α) : FreeM eff α

-- An effect definition that might be used in the freer monad.
inductive Eff : Type → Type where
  | readMem (addr : Addr) : Eff Value
  | writeMem (addr : Addr) (value : Value) : Eff Unit
```

To avoid the universe inconsistency errors, ArchSem-Rocq switched to a definition of free monads which can live at the same universe level as its constituent types [2]. This relies on an alternative definition of effects: an effect is a `Eff : Type v` paired with a function from effects to their return type `Eff → Type u`. This slightly restricts the kinds of effects we can define. For example, an "identity" effect that takes a value of arbitrary `Type u` and returns it back with no change is impossible to define using this effect specification (proof in Appendix B). But it turns out that for the purpose of defining architecture instruction effects, the extra expressiveness is not needed. My Lean implementation of the ArchSem style free monad and an example effect definition is included below:

```
-- ArchSem style free monad, annotated with universe levels.
inductive FreeM.{u, v, w} (eff : Type v) (effRet : eff → Type u) (α : Type w)
    : Type (max u (max v w)) where
  | pure (a : α) : FreeM eff effRet α
  | impure (call : eff) (cont : effRet call → FreeM eff effRet α)
      : FreeM eff effRet α

-- An ArchSem style effect definition.
inductive Eff : Type where
  | readMem (addr : Addr) : Eff
  | writeMem (addr : Addr) (value : Value) : Eff
def Eff.ret : InstructionEffect → Type
  | readMem _ => Value
  | writeMem _ => Unit
```

Lean's pervasive use of universe polymorphism means that escalating universe levels, as the freer monad does, would not be as much of an issue as it was in Rocq. However, after discussing with Thibaut Pérami, I decided it was still best to use the second free monad implementation that can live at the same universe level as its effect return types. This makes it possible to, for example, return a free monad from a free monad at the same universe level.

3.1.2. Instruction Effects

The instruction effects I implemented for ArchSem-Lean are largely the same as those defined in ArchSem-Rocq, except I add three new effects for tracking the number of clock cycles and printing debug messages from the ISA semantics. These features were offered by the existing Sail-to-Lean backend and I don't want to regress. In this sub-section, I will discuss an instructive subset of

the sixteen instruction effects available in ArchSem-Lean. A complete listing of the ArchSem-Lean instruction effects can be found in Appendix C.

- The memory read effect, `memRead (memReq : MemRequest)`, returns an `Except Arch.abort (BitVec (8 * memReq.size) × BitVec (memReq.numTag))`. The `MemRequest` structure contains the address and size of the request along with architecture-specific memory access parameters. The read may fail by returning an architecture-specific error type `Arch.abort`, which might be caused by an address decode error or page-fault for example. To support CHERI architectures, the return value may contain a bitvector of capability tags.
- The register read effect, `regRead (reg : Arch.register) (accessType : Option Arch.sys_reg_id)`, returns an `Arch.register_type reg`. The `Arch.register` is an architecture-specific enumeration of registers. To support architectures having registers of different sizes or types, we use Lean's dependent type system to make the return type depend on the register accessed, `reg`. The register effects are optionally equipped with an architecture specific *access type* which is used by the Arm system registers whose synchronisation depends on whether the accesses are *explicit* or *implicit*[8].
- The ISA specification can non-deterministically branch into `n` continuations by invoking the `choice (n : Nat)` effect. Each of the `n` non-deterministic continuations will receive a unique integer from `0` to `n-1` inclusive. Notice how this restricts the ISA to finite non-determinism. The `choice 0` effect can be used to discard the current non-deterministic execution path. Real architectures frequently have non-determinism in their specification, for example Arm-A's store exclusive `STXR` instruction, for which it is "implementation defined" whether or not the store takes place in certain circumstances[9].

3.1.3. The Architecture Typeclass

We have seen that many of the types used to define the instruction effects are architecture-specific such as the register types `Arch.register` and the physical memory access error type `Arch.abort`. The architecture typeclass `Arch` is defined in the Lean-Sail support library and instantiated for each architecture by code generated with the Sail-to-Lean backend. Memory models can then be written against an arbitrary `Arch`. I refactored the support library to make all architecture-specific parameters live in the `Arch` typeclass.

3.1.4. ArchExtra

When writing memory models, we frequently need more architecture-specific functionality than is exposed through the `Arch` typeclass. For example, in Section 3.3.1 we will require a hash function for register types and values, but the `Arch` typeclass does not provide this feature on its `Arch.register` type. Updating the Sail-to-Lean backend every time such a feature is required would significantly hinder speed of development, so in ArchSem-Lean I define an `ArchExtra` typeclass containing the additional functionality I require on `Arch` and then write memory models in terms of `ArchExtra`. For each architecture supported in ArchSem, I write a small piece of Lean code to instantiate `ArchExtra` using the automatically generated `Arch`.

[8] See section D24.1.2.2 of Armv9.6 A-Profile EAC Architecture Reference Manual DDI0487_M.a.a [3]
[9] See section B2.12 of Armv9.6 A-Profile EAC Architecture Reference Manual DDI0487_M.a.a [3]

In ArchSem-Rocq, all of the `ArchExtra` typeclass fields are manually instantiated for each architecture. I present an original technique to automatically populate the `ArchExtra` fields by treating the task of populating the `ArchExtra` fields as if I were populating the sub-goals of a proof, which allows me to use Lean's proof automation to automatically fill the fields using type inference. The following example code compares a manual `ArchExtra` instantiation with an automated one for a simplified `ArchExtra` typeclass. For an `ArchExtra` this small the automation is overkill, but in the real `ArchExtra` we are able to populate 23 fields in only 8 lines of proof.

```
-- Manually populate ArchExtra fields.
instance ManualArchExtra : ArchExtra MyArch where
  register_to_str.toString := fun n : Fin 32 => toString n
  barrier_to_str.toString := fun b : Unit => toString b

-- Use Lean's proof automation to infer the fields.
instance AutoArchExtra : ArchExtra MyArch := by -- Begin tactic block.
  refine' {..} -- Split fields into sub-goals.
  all_goals (conv => rhs ; whnf) ; infer_instance -- Expand definition and infer.
```

3.2. Generating ISA Semantics for Lean

Now that we have ported the ArchSem interface to Lean, we must instantiate the ISA and memory model on either side. In this section, we begin with the ISA semantics as-per the relevant design requirement:

1. **Sail ISA semantics**. ArchSem-Lean should use ISA semantics generated from Sail's *Tiny-Arm* specification.

The existing Sail-to-Lean backend converts Tiny-Arm to Lean code against the old Lean-Sail support library. Now that we have updated the support library, the backend must change to use the new ArchSem-Lean interface. This work was done in collaboration with Léo Stefanesco, the main developer of the Sail-to-Lean backend. I manually modified the generated Lean code to fit my requirements, then I gave an annotated diff to Léo who lifted these changes into the Sail-to-Lean backend after some back-and-forth discussion.

A significant change compared to the old Sail-to-Lean backend is that infinite non-determinism is no longer supported. The support library previously provided a mechanism for the ISA to request a natural number be non-deterministically generated. Since there are infinite natural numbers, this gave the ISA access to infinite non-determinism which meant that the problem of determining if a final state is reachable for some litmus test is undecidable. In the new interface, the ISA only has access to finite non-determinism through the `choice` effect which can generate a finite natural number under some specified maximum.

I also wrote some code to hide the nature of Sail's type system from the Lean interface. Sail uses *liquid types* [5, 35], a restricted form of dependent typing. Structures in Sail which have semantically meaningful type-level parameters often need to duplicate these parameters at the term-level so they are not lost after converting to non-dependently typed languages like C. Consider the Sail `Mem_request` type (below), the size of the memory request `'n` is a type-level argument so that the

typechecker has visibility into its value. We also have a `size` term-level attribute of type `int('n)` (integer that is equal to `'n`).

```
struct Mem_request('n : Int, 'nt : Int, 'addr_size : Int, 'addr_space : Type,
    'mem_acc : Type), 'n >= 0 & 'nt >= 0 & 'addr_size > 0 = {
  access_kind : 'mem_acc,
  address : bits('addr_size),
  address_space : 'addr_space,
  size : int('n),
  num_tag : int('nt), // This is CHERI tags, MTE-like tags are not supported
}
```

The Lean-Sail backend naively generalises term-level `size` attribute to `Nat` and keeps the type-level parameter, resulting in a Lean `Mem_request` type containing some redundancy. This is hidden from the ArchSem-Lean interface by conversions back and forth in the Lean-Sail support library. Below, I show the `Mem_request` type automatically generated by the Lean-Sail backend and the `MemRequest` exposed to the ArchSem interface.

```
/- Sail's memory request, automatically converted to Lean. -/
structure Mem_request (_size : Nat) (_numTags : Nat)
    (addr_size : Nat) (addr_space : Type) (mem_acc : Type) where
  access_kind : mem_acc
  address : BitVec addr_size
  address_space : addr_space
  size : Nat
  num_tag : Nat

/-- ArchSem's external memory request type used in ISA effects. -/
structure MemRequest where
  accessKind : Arch.mem_acc
  address : BitVec Arch.addr_size
  addressSpace : Arch.addr_space
  size : Nat
  numTag : Nat
```

3.3. Sequential Memory Model

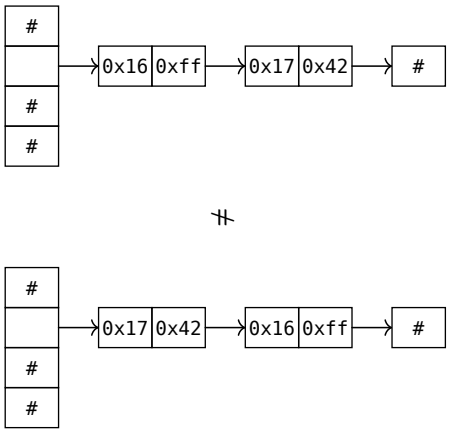
Our design requirements state I must implement the promising relaxed concurrency model. As a warm-up, I start by implementing the simplest possible memory model: the sequential and single-threaded model, where instructions execute in program order and their effects immediately propagate to memory. In such a model we only need to maintain registers for the one thread and memory can be treated as a simple mapping from addresses to values:

```
def MemoryMap := Std.ExtTreeMap Address (BitVec 8)
def RegisterMap := Std.ExtDTreeMap Arch.register Arch.register_type
structure SequentialState where
  regs : RegisterMap
  mem : MemoryMap
  ...
```

3.3.1. Representing Memory

Before I arrived at using the `ExtTreeMap` to represent the contents of memory, I tried two other options as described:

- `HashMap`. We expect memory to be sparse, so a hash map keyed by memory addresses is a reasonable first idea. The `HashMap` Lean standard library type is implemented as a separate chaining hash table so address-value pairs with a hash collision are stored in a simple linked list. Now suppose we construct two hash maps containing the same address-value pairs but by inserting them in a different order we end up with the linked lists for colliding pairs in a different order. Clearly these hash maps should be considered semantically equal, but enumerating their contents will output pairs in a different order revealing their difference. In Lean, equal terms are required to behave equivalently when passed to any function, so we are unable to define a lawful equality on *internal* hash maps.



Hash maps which enumerate differently may not be equal.

- `ExtHashMap`. In contrast to the *internal* hash maps, Lean provides an *external* hash map. By removing the ability to enumerate their contents, it becomes impossible to distinguish external hash maps by their internal orderings, hence equality can be defined between them. The external hash maps are not suitable for ArchSem, as we frequently need to enumerate the contents of memory to convert from architecture-specific to architecture-generic memory formats or to hash the contents of a memory map.
- `ExtTreeMap`. The Lean standard library provides an extensional self-balancing binary search tree which supports both equality and enumeration. This is possible because the total ordering of elements in the search tree can be used to enumerate the contents in the same way for all maps with equal contents. The lookup and insert operations are asymptotically slower than the hash maps, but in practice we find this is not an issue because most litmus tests only access a small and bounded region of memory.

We could achieve both fast lookup operations and well-behaved equality with a custom hash map carefully designed to guarantee that maps with the same content enumerate in the same order. However, building such a map would be a significant undertaking as one would need to prove many lemmas before it becomes suitable for theorem proving. The Lean standard library implements over 650 lemmas about the simple `HashMap` type[10].

The register map is an `ExtDTreeMap`, the *dependent* generalisation of `ExtTreeMap` in which the type of the values is a function of the corresponding key. This is to support architectures with heterogeneous register types.

[10] `HashMap` lemmas from the Lean standard library https://github.com/leanprover/lean4/blob/3dc1a088b6d2d8eafe25a7cd7ec7b58d731bd7cc/src/Std/Data/HashMap/Lemmas.lean

3.3.2. The Non-deterministic State Monad

The ISA semantics are encapsulated within a free monad so to recover the sequential model we can interpret the free monad down into a *non-deterministic error state monad* over the sequential architecture state. This monad is similar to the regular state monad, except that the effects are transitions in a **non**-deterministic state machine and there is support for effects throwing exceptions. Transitions are defined as a function from initial state to a **list of** updated states and return values and to a list of exceptions thrown. By implementing a function from each of the instruction effects to their corresponding non-deterministic state transitions between sequential architecture states, I can interpret the Tiny-Arm instruction semantics free monad into an executable function.

3.4. Promising Memory Model

This section discusses the promising memory model in more detail and covers some challenges I faced while implementing it in Lean.

2. **Promising memory model**. The promising memory-model [19] should be implemented to complete the *Tiny-Arm* architecture specification.

In contrast to the Flat model of Arm relaxed memory semantics which models speculative execution explicitly, the promising model evaluates instructions in order. How then, is it able to model out-of-order reads and out-of-order writes speculated by the processor? We shall see by means of example below:

Out-of-order reads. Consider the candidate execution of Figure 8. It is allowed in Arm's relaxed memory model because the reads can be speculatively executed before the stores. The promising model allows this test by maintaining a history of writes to memory and allowing reads to access old values. Our history begins in the initial state $[0 : \langle x := y := 0\rangle]$. Suppose events are evaluated in the order *(a)*, *(b)*, *(c)*, *(d)*. Event *(a)* appends a write `x=1` to memory, *(b)* reads a zero from `y`, and *(c)* appends a write `y=1` to memory leaving it in the state $[0 : \langle x := y := 0\rangle; 1 : \langle x := 1\rangle; 2 : \langle y := 1\rangle]$. When *(d)* reads `y`, it is allowed to do so either from $\langle y := 1\rangle$ at timestamp 2 or initial memory $\langle x := y := 0\rangle$ at timestamp 0 so the test is allowed.

SB — Arm

Initially all memory and registers 0	
Thread 0	Thread 1
MOV [x], #1; (a)	MOV [y], #1; (c)
LDR R0, [y]; (b)	LDR R0, [x]; (d)
Final: 0:R0=0; 1:R0=0;	

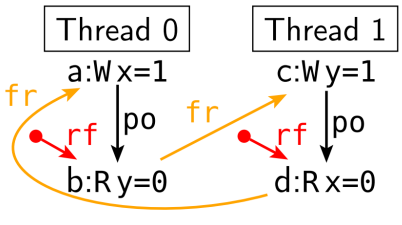


Figure 8: The *SB* litmus test (left) and its candidate execution graph (right)[11].

But clearly there must be some constraint on what timestamp a read can access. For example, if there were full memory barriers in-between the stores and reads on both threads then the reads should not both be allowed to read from initial memory. The promising model constrains reads by maintaining a number of *views* on each thread which restrict what timestamps can be read from. The example of the interleaved memory barriers is properly rejected because when the promising model evaluates a

[11]Candidate execution graph (right) from Sewell et al. [15].

DMB.SY memory barrier, it updates its vdmb view to that of the most recent memory access from the same thread. And reads are constrained to timestamps after or equal to the timestamp of the thread's vdmb view. The model as I implemented it maintains 8 different views per-thread plus coherence views at each memory location to handle similar ordering requirements.

Out-of-order writes. Allowing reads to come from old memory values is not enough to properly model the litmus test execution of Figure 9 which is allowed in Arm because the stores can speculatively execute before the reads.

LB Arm

Initially all memory and registers 0	
Thread 0	Thread 1
LDR R0, [x]; (a)	LDR R0, [y]; (c)
MOV [y], #1; (b)	MOV [x], #1; (d)
Final: 0:R0=1; 1:R0=1;	

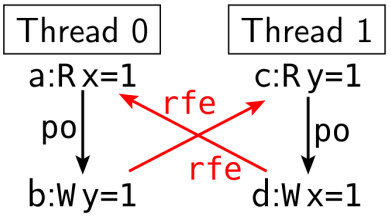


Figure 9: The *LB* litmus test (left) and its candidate execution graph (right)[12].

The promising model handles cases like this by allowing certain writes to be *promised* (added to the memory history before they are executed). If a thread can take multiple sequential steps on its own to produce a write, then that write is allowed to be promised in the current thread state. When a thread makes a promise, it must later be fulfilled; non-deterministic execution paths in which promises are not fulfilled are discarded. The Figure 9 candidate execution is allowed by the following reasoning:

1. Thread 0 promises y=1, which is allowed since it *could* execute *(a)* and *(b)* right now to produce that write
2. Thread 1 executes *(c)* and *(d)*, reading from the promised y=1
3. Thread 0 executes *(a)* which reads from *(d)*'s x=1
4. Thread 0 executes *(b)*, fulfilling the promise to write y=1

Promising first. Given a litmus test, the naive way to evaluate all allowed executions according to the promising model is as follows: at each step of execution, non-deterministically choose a promise to make or an instruction to run. So each non-deterministic trace takes a sequence of steps that are each either making a promise or running an instruction. It has been proven by Pulte et al. [19] that any promising trace of this form has the same final state as some trace that first does all promising steps and then does all instruction steps. Hence we can reduce non-determinism by forcing all promises to be made up-front. This equivalent but significantly more efficient execution scheme is called the *promising-first model* and I have implemented it in addition to the *naive model*.

Fixing correctness bugs in ArchSem-Rocq. While writing the Lean promising model I made frequent reference to both the promising paper [19] and the ArchSem-Rocq promising implementation. In the process of reading and understanding these, I discovered two separate bugs in the existing Rocq implementation, each of which broke model correctness. My PRs to fix these bugs have been upstreamed into ArchSem-Rocq.

De-duplication of model states. While non-deterministically executing the promising model, we often find that multiple execution paths arrive at the same architecture state and so we need to

[12]Candidate execution graph (right) from Sewell et al. [15].

de-duplicate these repeated states. To avoid premature optimisation, I initially used a naive $O(n^2)$ list de-duplication working on pairwise equality checks. However, performance profiling revealed this significantly impacted runtime. Therefore I switched to an $O(n)$ hash-based de-duplication algorithm that works by inserting all architecture states into a hash set and then reading out the distinct elements. An interesting part of my architecture state hashing algorithm is the hashing of memory maps which works by XORing the result of individually hashing all address-value pairs. In Appendix D I have written a paper-maths proof showing that the probability of collision is negligible.

3.5. Parsing Litmus Tests

After implementing the promising memory model, I manually wrote Lean definitions for the initial memory and register states of three litmus tests[13]. I then ran the promising model with Tiny-Arm ISA semantics against these tests and verified the outcome was as expected in a regression test, fulfilling the design requirement:

> 4. **Correctness**. The executable Tiny-Arm architecture specification must show equivalent behaviour to the Rocq implementation for at least three small litmus tests.

This section discusses implementing an extension of this requirement: automatically parsing litmus tests. Implementing such a feature will aid my evaluation chapter, where I will test the correctness of my promising model against a large corpus of litmus tests and validate the results are equivalent to an existing test oracle.

I implemented a parser for the same `.archsem.toml` litmus test format used by ArchSem-Rocq. An example of this format is in Appendix E. This litmus test format specifies:
- Initial register state for each thread
- Initial memory state
- Termination condition
- Expected final state(s)

The test format must support multiple processor architectures, and we should be able to convert to the particular formats required by each memory model. Hence, there are 4 passes involved:
1. The file is parsed into a TOML datastructure
2. This is parsed into an architecture-generic litmus test
3. This is specialized into an architecture-specific litmus test
4. This is specialized into a memory-model-specific litmus test

The parser is a good demonstration of the benefits of using Lean over Rocq. I implemented it using Lean's built-in IO library with code looking similar to how it would in any other functional language despite the fact that Lean is also a theorem proving language. Rocq is not a very capable general-purpose functional language; as a result, ArchSem-Rocq chose to implement its parser in OCaml, adding complexity to the project by having to extract Rocq code to OCaml.

After parsing was implemented, I added a regression test which parsed and ran 21 well-known litmus tests against the Tiny-Arm promising model, validating their output is as expected.

[13] These were the *MP* test, the *MP* test with full memory barriers, and a single-threaded *EOR* test.

3.6. Proof of Fuel Monotonicity

To help evaluate the suitability of Lean's interactive proof system for ArchSem, I have written an original machine-checked proof in Lean for the *fuel monotonicity* of the promising model in both its naive and promise-first forms. This correctness property is explained below.

To make our operational models useful for proof, we must ensure they always terminate, even when asked to execute machine code containing an infinite loop. We guarantee this by starting the model with an integer *fuel* argument which counts down on every operational step the model takes, throwing an "out of fuel" error if it reaches zero. The *fuel monotonicity* property is as follows: The set of allowed final states reported by the model when run with fuel n is a subset of final states reported when run with some fuel greater than n. In other words, reducing the fuel cannot introduce more allowed behaviour. The theorem statement in Lean is as follows:

```
theorem naive_promising_monotonic_fuel_full {isem : SailM Unit} {fuel : Nat}
    : ∀ fuel' ≥ fuel,
    (createNaiveModel isem fuel).weaker (createNaiveModel isem fuel')
theorem promise_first_monotonic_fuel_full {isem : SailM Unit} {fuel : Nat}
    : ∀ fuel' ≥ fuel,
    (createPromiseFirstModel isem fuel).weaker (createPromiseFirstModel isem fuel')
```

Where `.weaker` is defined between `ComputationalTerminatingModels`. Model m_1 is weaker than m_2 iff m_1 always produces a subset of m_2's final states when running on any litmus test.

```
def ComputationalTerminatingModel.weaker
    (m₁ m₂ : ComputationalTerminatingModel) : Prop :=
  ∀ (nThreads : Nat) (termCond : TerminationCondition nThreads)
    (init final : ArchState nThreads) (t : final.has_terminated termCond),
  ∀ r ∈ (m₁ nThreads termCond init),
    r = ModelResult.finalState final t →
    r ∈ (m₂ nThreads termCond init)
```

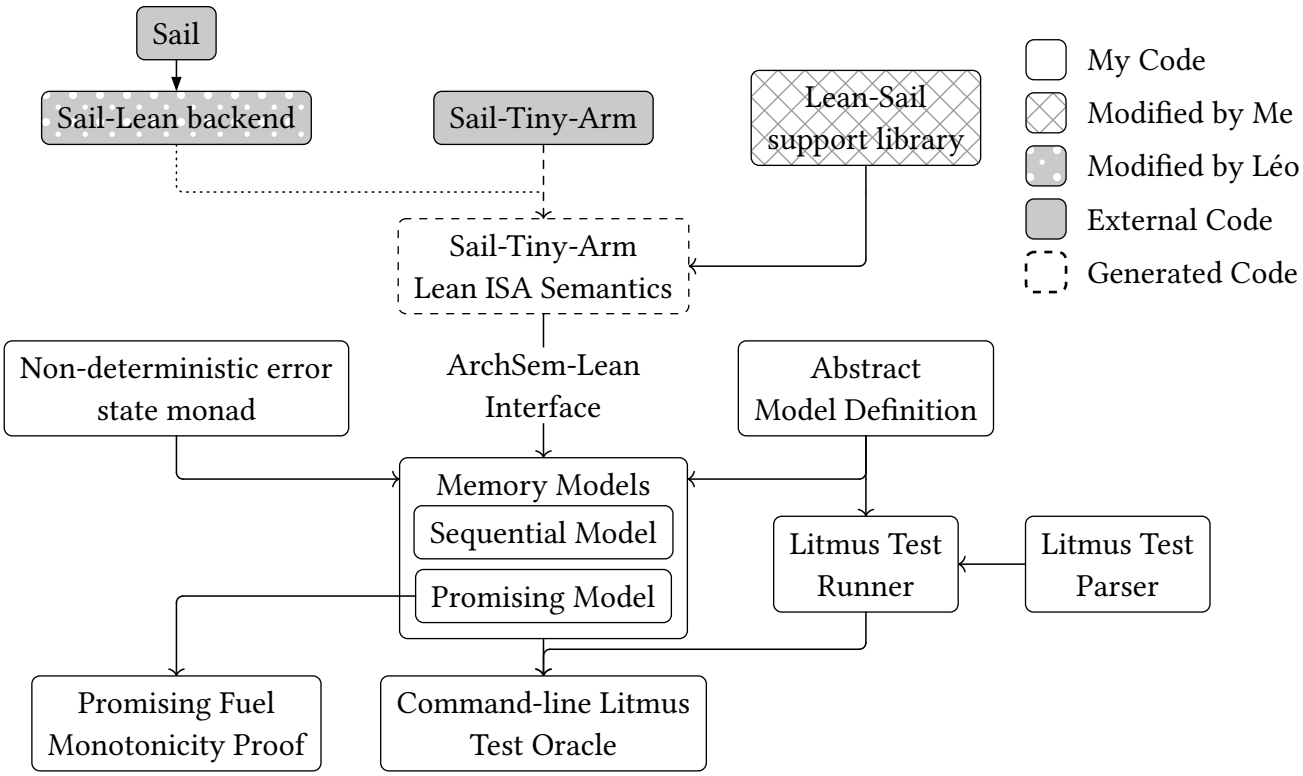
This was a substantial proof that sufficiently demonstrated Lean's interactive theorem proving capability for realistic proofs. In 407 non-whitespace lines of proof, I wrote twenty-four lemmas which build up to the fuel monotonicity proofs, summarised below:

- Reflexivity and transitivity of `ComputationalTerminatingModel.weaker`
- 9 lemmas about my implementation of the non-deterministic error state monad including:
  - The composition of monotonic monads is monotonic
  - Congruence rules for the monadic bind
  - Unwrapping composition of monadic binds. I.e., for non-deterministic state transitions f and g, $s \in g(f(\{s_0\})) \Leftrightarrow \exists s'.s \in g(\{s'\}) \wedge s' \in f(\{s_0\})$
  - Simplification of binds with common trivial state transitions
- 13 lemmas specific to the promising model including:
  - Running a thread in isolation until its termination condition is monotonic in fuel
  - If running a thread in isolation until its termination does not run out of fuel then it will also not run out of fuel if given more fuel
  - As you increase the fuel you give to running a thread in isolation until its termination, the results will eventually stop changing
  - Finding all promises that can be made from some thread state is monotonic in fuel

3.7. Repository Overview

The code relating to ArchSem-Lean is primarily split across four different repositories. The Sail-to-Lean backend lives in **`Sail`**'s code repository. I use this backend against the existing **`Sail-Tiny-Arm`** specification to generate a Lean ISA specification which depends upon the **`Lean-Sail`** support library. Finally, **`ArchSem-Lean`** implements memory models to build full architecture models from the generated ISA specification.

For my project, I wrote `ArchSem-Lean` by using the theory developed in `ArchSem-Rocq` and I modified the `Lean-Sail` support library to implement the ArchSem interface. After some back-and-forth discussion with Léo Stefanesco, he modified the Sail-to-Lean backend to make use of the changes I made to the support library. The following dependency graph and listing of important code files illustrates the components of the project.



```
├─ ./ArchSem-Lean/
│  ├─ ./Main.lean ..................  CLI litmus test runner (Code:  67, Comments:   3)
│  ├─ ./ArchSem/
│  │  ├─ ./Defs.lean .............  Common type definitions (Code:  59, Comments:  35)
│  │  ├─ ./NondeterministicMonad.lean . .  Monad definition (Code: 173, Comments:  46)
│  │  └─ ./TerminatingModel.lean .  Abstract model definition (Code:  75, Comments:  29)
│  ├─ ./LitmusTest/
│  │  ├─ ./Defs.lean .............  Common type definitions (Code:  61, Comments:  32)
│  │  ├─ ./Parse.lean .................  Litmus test parsing (Code: 187, Comments:  28)
│  │  └─ ./Run.lean .................  Running litmus tests (Code: 121, Comments:  23)
│  └─ ./ArchSemTinyArm/
│     ├─ ./Defs.lean ....  ArchExtra and printing Arch types (Code: 114, Comments:  12)
│     ├─ ./Promising.lean .......  Promising memory model (Code: 507, Comments: 243)
│     ├─ ./PromisingFirstProof.lean .....  Section 3.6 proof (Code: 321, Comments:   6)
│     ├─ ./PromisingTest.lean ....  3 hard-coded litmus tests (Code: 177, Comments:  39)
│     ├─ ./Sequential.lean ......  Sequential memory model (Code:  94, Comments:  17)
│     ├─ ./SequentialTest.lean ...  2 hard-coded litmus tests (Code:  50, Comments:  13)
│     └─ ./Test.lean .............  21 regression litmus tests (Code:  37, Comments:   6)
└─ ./lean-sail/ ..............................  Lean-Sail support library that I modified
   ├─ ./Sail/
   │  ├─ ./ArchSem.lean ............  The ArchSem interface (Code: 274, Comments:  50)
   │  ├─ ./Common.lean ..........  Common type definitions (Code: 246, Comments:   9)
   │  └─ ./ConcurrencyInterfaceV1.lean . .  The old interface (Code: 272, Comments:   0)
   └─ ./SailTest/
      └─ ./VectorConversion.lean .........  Regression tests (Code:  24, Comments:   0)
```

4. Evaluation

This section critically evaluates the success of my project. I will show that ArchSem-Lean is significantly more performant and maintainable than its Rocq counterpart and showcase strong evidence for its correctness of implementation. Section 4.5 demonstrates that all success criteria and design requirements were met or exceeded. In the subsequent conclusion chapter, we shall answer our original question: to what extent is Lean a suitable language for formalising architecture semantics?

4.1. Generating Litmus Tests

This section explains how I automatically generate the litmus tests used in the following evaluation. Recall that litmus test candidate executions can be viewed as a directed graph between memory events with labelled edges. The `diy7` tool uses this perspective to automatically generate many litmus tests. It generates all graphs containing exactly one cycle, bounded by user-supplied size, number of threads and edge types. Then for each graph, it generates a litmus test with a final condition which can only be met by a candidate execution containing that cycle of memory events. The result is many litmus tests which each demonstrate a non-sequential memory behaviour which may or may not be allowed by the chosen architecture. The exact options passed to diy for each test suite can be found in Appendix F.

Large test corpus. I generated this corpus of 15,793 litmus tests to make use of all memory concurrency primitives supported by Tiny-Arm so that we get good coverage of the promising model when testing for correctness.

Small test corpus. I found that ArchSem-Rocq, Isla-Axiomatic and Rmem Flat were too slow to run the large test corpus on my hardware within 12 hours so I generated this smaller test corpus of only 108 litmus tests for the purpose of comparing performance.

Wide test corpus. This test corpus contains 184 tests varying from very small tests taking milliseconds to evaluate, up to very large tests that take minutes. It is used to investigate how the performance compares between ArchSem-Lean and Rocq for varying test sizes.

4.2. Correctness of the Relaxed Memory Model

Herd [1] provides an axiomatic specification of Armv8′s memory model in the form of a Cat file[14]. This axiomatic model is implemented in a fundamentally different way to our operational promising model and the ISA semantics are implemented independently. This makes it a good target to test against, because it is unlikely that we make the same kind of implementation errors. I use both ArchSem-Lean and Herd7 to evaluate which of the **large test corpus** tests are architecturally allowed and I discover that they agree exactly.

4.3. Comparing Performance

For each tool discussed in Section 2.4, I measure the time taken to evaluate the architecturally allowed behaviour of all tests in the **small test corpus**. The reported durations, shown in Figure 10, are the minimum of three runs to reduce the effect of background system activity on results.

[14]https://github.com/herd/herdtools7/blob/1ca343e16a2038e406d1ac674e7e3a1b722b36c7/herd/libdir/aarch64.cat

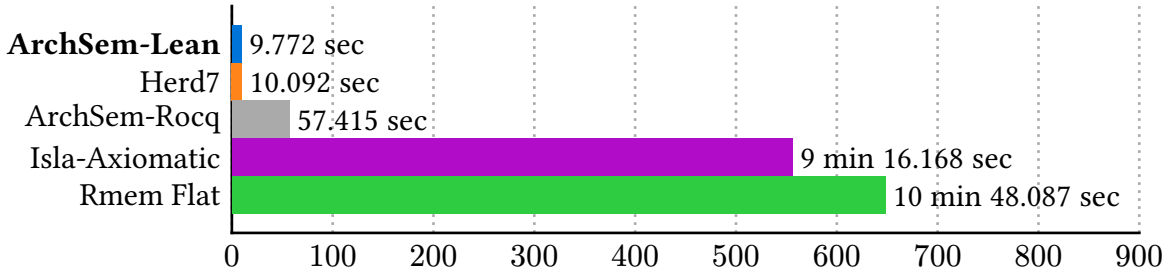


Figure 10: **Small test corpus** runtime. See Appendix G for version details.

The Isla-Axiomatic and Rmem Flat models were using the full Armv8 ISA semantics as they do not have corresponding Tiny-Arm implementations. I suspect this is the reason they run so much slower. On this particular test suite, ArchSem-Lean's performance was comparable with Herd7. However, the axiomatic models tend to perform better for larger litmus tests. The exciting result is the over 5x performance gain of ArchSem-Lean over ArchSem-Rocq, which exceeded my expectations. The initial motivations for porting ArchSem to Lean were primarily for its language features and ecosystem but it is encouraging to see that performance is yet another advantage of ArchSem-Lean.

To extrapolate our performance gains to other test suites, I investigate how performance compares on individual litmus tests of differing sizes. I measure the *size* of a litmus test as the number of distinct executions according to our promising model. The **wide test corpus** contains tests of a range of sizes from 8 to 161,280. For each test, I measure the time taken to evaluate allowed executions for both Lean and Rocq ArchSem implementations and plot against test size in Figure 11 (top) using log axis. By inspection, the relationship between test size and runtime follows a power law with y-intercept at the time taken to parse the test. So I fit a model of the form $y = w_0 x^{w_1} + c$ to both ArchSem implementations using non-linear least squares, where c is hardcoded to the measured time taken to parse a litmus test. Non-parametric resampling is used to plot a 90% confidence interval around the curves of best fit shown in Figure 11.

Interestingly, it appears that Lean's performance gain is super-linear. This is best demonstrated by Figure 11 (bottom) which plots the relative performance of Lean and Rocq implementations by test size. As you can see, for smaller tests around size 10^3, Lean's performance gain is only around 4x, but for the larger test we have observed almost 14x performance gain. I attribute the improvement to Lean's *functional but in place* memory management paradigm [23, 36] which avoids allocation when objects are functionally updated. If the Lean compiler were simply reducing the number of instructions in each function, we would only expect to see a linear performance gain. But a reduction in memory allocation could explain the super-linear improvement due to non-uniform memory access times in the cache hierarchy.

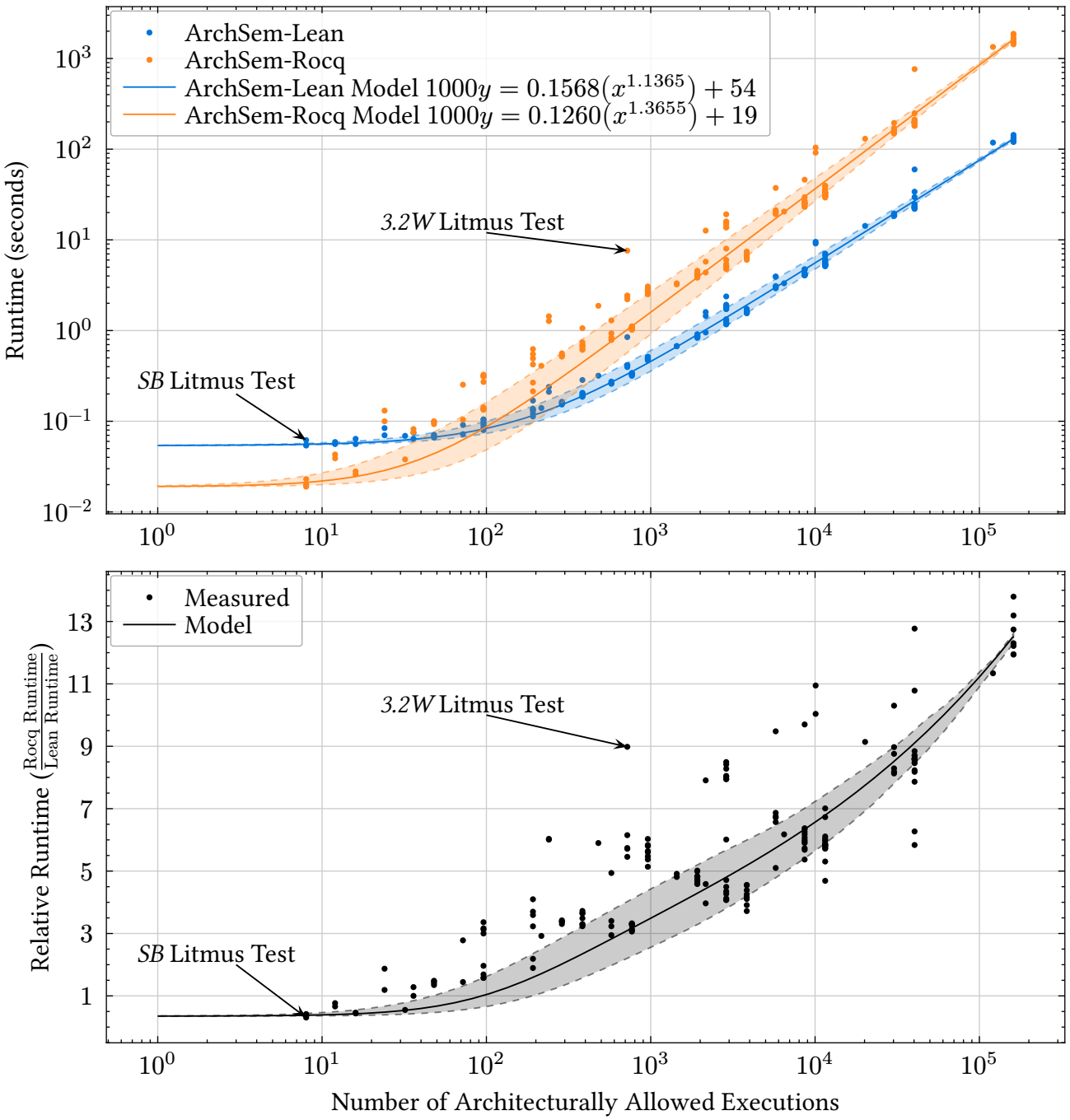


Figure 11: (top) The measured runtime of all litmus tests in the **wide test corpus** and the best-fit power curves with equal-tailed 90% confidence intervals. (bottom) The same data and model as above except the y-axis is the relative performance difference between ArchSem-Lean and Rocq.

4.4. Comparing Maintainability

Boilerplate. In the introduction chapter, I noted that of the 22,054 lines of Rocq code in the existing ArchSem, 7,933 are implementing functionality that exists in Lean's standard library. ArchSem-Rocq is also balancing three programming languages: `Rocq` for theorem proving, `Ltac2` for metaprogramming, and `OCaml` for general-purpose IO programming. My ArchSem-Lean implementation successfully leverages the standard library, and is able to use Lean as the single language for theorem proving, metaprogramming, and IO.

Error messages. One significant advantage of ArchSem-Lean over the Rocq implementation is the improved error messages from the Lean compiler. To demonstrate this, I created the same intentional type error in Rocq and Lean and compared the error messages thrown by Lean and Rocq respectively. The full Lean error message and the first 6 of the 28 lines of the Rocq message are shown below. The full error messages are included in Appendix H. Lean throws a sensible "Type mismatch" error while Rocq's error message is largely incomprehensible to a developer unfamiliar with its internals.

Lean's Error Message

```
error: ArchSemTinyArm/Promising.lean:538:4: Type mismatch
  ((), none)
has type
  Unit × Option ?m.899
but is expected to have type
  NEStateM String ProjectedModelState ((InstructionEffect.barrier
(Barrier.Barrier_ISB ())).ret × Option View)
```

Rocq's Error Message (truncated)

```
File "./ArchSemArm/UMPromising.v", line 520, characters 2-3462:
Error: UNDEFINED EVARS:
 ?X1523==[tid initmem |- MBind ?M] (parameter MBind of mbind) {?MBind}
 ?X1528==[tid initmem ts |- MBind ?M0] (parameter MBind of mbind) {?MBind0}
 ?X1537==[tid initmem ts |-
           MCall (MState (PPState.t TState.t ?mEvent ?iis_t)) ?M1]
           (parameter MCall0 of mset) {?MCall0}
```

4.5. Success Criteria and Design Requirements Met

This section reflects upon the original success criteria as defined in my project proposal and the more concrete design requirements written in Section 2.8. All criteria and requirements were met and some were exceeded.

Success Criteria:

1. Model CPU architecture semantics in Lean by combining one ISA semantics with at least one relaxed concurrency model.

I have written the promising relaxed concurrency model [19] for Sail's Tiny-Arm ISA semantics as discussed in Section 3.4. From the resulting architecture model I build the executable test oracle used in the above evaluation.

2. Demonstrate that at least three small litmus tests have equivalent behaviour in both the existing Rocq ArchSem implementation and the new Lean implementation.

The ArchSem-Lean GitHub repository's continuous integration validates that the *MP* test, the *MP* test with full memory barriers and a single-threaded *EOR* test each have equivalent allowed final states compared to the Rocq implementation[15]. Thanks to my implementation of automatic litmus test parsing, I also validated that all 15,793 litmus tests in the **large test corpus** agree with Herd (Section 4.2).

Functional Requirements:

1. **Sail ISA semantics**. ArchSem-Lean should use ISA semantics generated from Sail's *Tiny-Arm* specification.

ArchSem-Lean indeed does use Tiny-Arm ISA semantics as discussed in Section 3.2. My correctness testing is evidence that this ISA semantics is correct with respect to the Herd implementation.

2. **Promising memory model**. The promising memory-model [19] should be implemented to complete the *Tiny-Arm* architecture specification.

I implemented the promising model in Section 3.4 and discovered that my implementation is more performant than ArchSem-Rocq's in Section 4.3.

3. **Executable**. The ArchSem-Lean Tiny-Arm architecture specification should be executable on small litmus tests. Given an initial multiprocessor state and termination condition, it should be able to evaluate all architecturally allowed final states.

My command line executable test oracle was successfully used for all correctness and performance tests discussed earlier in this chapter.

4. **Correctness**. The executable Tiny-Arm architecture specification must show equivalent behaviour to the Rocq implementation for at least three small litmus tests.

Three tests were shown equivalent to ArchSem-Rocq and the **large test corpus** was tested against Herd in Section 4.2.

[15] See `ArchSemTinyArm/PromisingTest.lean` from the ArchSem-Lean repository and `ArchSemArm/tests/UMPromisingTest.v` from ArchSem-Rocq.

Non-Functional Requirements:

5. **Proof-ready**. ArchSem-Lean should facilitate writing machine-checked proofs about the Tiny-Arm architecture specification.

Thanks to proof-friendly design decisions such as using the extensional tree map to represent memory (see Section 3.3), our model is indeed suitable for formal verification. I trialled this with great success in Section 3.6 where I write a substantial proof for a correctness property of my memory model.

6. **Portability**. ArchSem-Lean should not require large dependencies such as Mathlib so that we can use the latest Lean-nightly build without waiting for dependencies to update.

The only dependencies of ArchSem-Lean are a small command line parsing library, `lean4-cli`, and the generated ISA semantics. ArchSem-Lean uses a nightly build of Lean4 since I make use of a list membership lemma that was added to the standard library in February 2026[16].

7. **Memory model generic**. The Sail-to-Lean backend should generate the Tiny-Arm ISA semantics for Lean without enforcing a specific memory model.

In Section 3.1.1 I implement the free monad and in Section 3.2 I discuss how the Sail-to-Lean backend was modified to use it. In Appendix A I explain how the free monad can be later specialised to any memory-model-specific monad that can implement our instruction effects.

[16] https://github.com/leanprover/lean4/pull/11811

5. Conclusions

My project has demonstrated that Lean is a highly suitable language for formalising the semantics of computer architectures. In the evaluation chapter, I quantified performance advantages and reduced boilerplate compared to ArchSem-Rocq and in my subjective experience I have found Lean to be a nicer language to work in as its modern language design feels polished and coherent. Now, for the first time, the Lean community has access to a framework for formalising multithreaded architecture semantics, instantiated with a simplified Arm architecture.

This section summarises the work completed, reflects on the lessons I learnt and explains the future work that I intend to complete.

5.1. Summary of Work Completed

ArchSem interface. ISA semantics communicate with memory concurrency semantics by issuing instruction effects through the ArchSem interface. Porting this to Lean required implementing a suitable free monad, a definition of instruction effects and an architecture typeclass containing all architecture-specific types observable from the memory model.

Updating the Lean-Sail backend. The Lean Sail backend previously generated ISA semantics in a hard-coded sequential memory model. In collaboration with Léo Stefanesco, the Lean-Sail backend was updated to use the new ArchSem-Lean interface.

Sequential memory model. The free monad encoding the Tiny-Arm ISA semantics was interpreted into a non-deterministic state monad over sequential architecture state for an executable model of the single-threaded architecture.

Promising model. I implemented the promising model of Pulte et al. [19] for Tiny-Arm using the ArchSem-Rocq implementation as a reference. I fixed two correctness bugs in ArchSem-Rocq's implementation.

Proof of promising fuel monotonicity. A meta-property of the promising model required for its correctness was proven in Lean.

Testing correctness. I implemented parsing of litmus tests and validated that a suite of over 15,000 litmus tests had the same behaviour in ArchSem-Lean and Herd. I observed a significant performance advantage over ArchSem-Rocq.

5.2. Reflection on the Lessons Learnt

I have come to appreciate the importance of an unambiguous contract between layers of abstraction in complex software systems. Although this project focused on formalising the architecture specification, I feel the same skills translate well to formalising other software interfaces such as network protocols or file formats. For example, I can imagine using the free monad to model the expected behaviour of a compliant SMTP server.

I also felt that learning a theorem proving language for the first time allowed me to transfer intuitions I have for writing code into a mathematical intuition. It was particularly interesting to observe how my existing intuition for how to write maintainable code was being transferred to writing maintainable proofs. After learning Lean, I noticed a significant improvement in my comprehension

during lectures from the more mathematical courses taught in Part II, which I attribute to this newfound mathematical intuition.

5.3. Future Work

I will continue development on ArchSem-Lean during a paid 2 week work placement in the CL labs after my exams. During this placement I plan to work on the following:

An improved non-deterministic state monad. In both Lean and Rocq versions of ArchSem, there are two immediate problems with the implementation of the non-deterministic state monad, stemming from the use of a list to express a set of states. Duplicates cannot be efficiently removed from the list and the order of states in the list is unnecessarily exposed to the user: this breaks associativity of the monadic bind because for non-deterministic transitions to be proven equal, they must map to the same order of states in addition to the same set of states. I plan to write an improved non-deterministic state monad that de-duplicates states after each transition and for which equality is defined extensionally.

Axiomatic models. An important feature of ArchSem is its ability to support multiple concurrency models in the same framework. But so far I have only implemented the operational promising model for Tiny-Arm. I plan to support parsing and execution of Cat axiomatic models so that one could, for example, use ArchSem-Lean to prove the equivalence of particular axiomatic and operational models.

Bibliography

[1] Jade Alglave, Luc Maranget, and Michael Tautschnig, "Herding cats: Modelling, simulation, testing, and data mining for weak memory," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 36, no. 2, pp. 1–74, 2014.

[2] Thibaut Pérami, Thomas Bauereiss, Brian Campbell, Zongyuan Liu, Nils Lauermann, Alasdair Armstrong, and Peter Sewell, "ArchSem: Reusable Rigorous Semantics of Relaxed Architectures," *Proceedings of the ACM on Programming Languages*, vol. 10, no. POPL, pp. 204–234, 2026.

[3] Arm Limited, "Arm Architecture Reference Manual for A-profile architecture version M.a.a." Cambridge, UK, Dec. 2025.

[4] Alastair Reid, "Trustworthy specifications of ARM® v8-A and v8-M system level architecture," in *2016 Formal Methods in Computer-Aided Design (FMCAD)*, 2016, pp. 161–168.

[5] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E Gray, Robert M Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, and others, "ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–31, 2019.

[6] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell, "Isla: Integrating full-scale ISA semantics and axiomatic concurrency models," in *International Conference on Computer Aided Verification*, 2021, pp. 303–316.

[7] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams, "Understanding POWER multiprocessors," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 175–186.

[8] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell, "Fences in weak memory models," in *International Conference on Computer Aided Verification*, 2010, pp. 258–272.

[9] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, and others, "The Coq proof assistant reference manual," *INRIA, version*, vol. 6, no. 11, pp. 17–21, 1999.

[10] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O Myreen, and Jade Alglave, "The semantics of x86-CC multiprocessor machine code," *ACM SIGPLAN Notices*, vol. 44, no. 1, pp. 379–391, 2009.

[11] Mike Gordon, "A Formalization of Simplified Subset of Alpha Shared Memory Model." Accessed: May 06, 2026. [Online]. Available: http://www.cl.cam.ac.uk/ftp/hvg/papers/AARM.ps.gz

[12] The mathlib Community, "The lean mathematical library," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, in CPP 2020. New Orleans, LA, USA: Association for Computing Machinery, 2020, pp. 367–381. doi: 10.1145/3372885.3373824.

[13] Freek Wiedijk, [Online]. Available: https://www.cs.ru.nl/~freek/100/

[14] Freek Wiedijk, "The QED manifesto revisited," 2007.

[15] Peter Sewell, Christopher Pulte, and Shaked Flur, “Multicore Semantics and Programming Lecture Slides.” University of Cambridge, 2026.

[16] Scott Owens, Susmit Sarkar, and Peter Sewell, “A better x86 memory model: x86-TSO,” in *International Conference on Theorem Proving in Higher Order Logics*, 2009, pp. 391–407.

[17] SPARC International Inc and David L Weaver, *The SPARC architecture manual*. Prentice-Hall Englewood Cliffs, NJ, USA, 1994.

[18] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell, “Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–29, 2017.

[19] Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur, “Promising-ARM/RISC-V: a simpler and faster operational concurrency model,” in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 1–15.

[20] Hadrien Renaud, “Executable semantics of Arm's Architecture Specification Language,” in *35es Journées Francophones des Langages Applicatifs (JFLA 2024)*, 2024.

[21] Leonardo De Moura and Nikolaj Bjørner, “Z3: An efficient SMT solver,” in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.

[22] Dominic P Mulligan, Scott Owens, Kathryn E Gray, Tom Ridge, and Peter Sewell, “Lem: reusable engineering of real-world semantics,” *ACM SIGPLAN Notices*, vol. 49, no. 9, pp. 175–188, 2014.

[23] Leonardo de Moura and Sebastian Ullrich, “The lean 4 theorem prover and programming language,” in *International Conference on Automated Deduction*, 2021, pp. 625–635.

[24] Thierry Coquand and Gérard Huet, “The calculus of constructions,” *Information and Computation*, vol. 76, no. 2, pp. 95–120, 1988, doi: https://doi.org/10.1016/0890-5401(88)90005-3.

[25] Per Martin-Löf, “A Theory of Types,” 1971, Unpublished manuscript, University of Stockholm.

[26] Jean-Yves Girard, “Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur,” Doctoral dissertation, Université Paris VII, 1972.

[27] Antonius JC Hurkens, “A simplification of Girard's paradox,” in *International Conference on Typed Lambda Calculi and Applications*, 1995, pp. 266–278.

[28] Thierry Coquand, “An Analysis of Girard's Paradox,” in *Logic in Computer Science*, 1986. [Online]. Available: https://api.semanticscholar.org/CorpusID:5533529

[29] Accessed: Apr. 18, 2026. [Online]. Available: https://leanprover-community.github.io/contribute/naming.html

[30] Accessed: May 12, 2026. [Online]. Available: https://www.gnu.org/licenses/license-list.html#FreeBSD

[31] Accessed: Apr. 18, 2026. [Online]. Available: https://opensource.org/license/BSD-2-Clause

[32] Accessed: Apr. 12, 2026. [Online]. Available: https://opensource.org/license/apache-2.0

[33] Free Software Foundation Europe, "REUSE Software." [Online]. Available: https://reuse.software/

[34] Oleg Kiselyov and Hiromi Ishii, "Freer monads, more extensible effects," *ACM SIGPLAN Notices*, vol. 50, no. 12, pp. 94–105, 2015.

[35] Patrick M Rondon, Ming Kawaguchi, and Ranjit Jhala, "Liquid types," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 159–169.

[36] Alex Reinking, Ningning Xie, Leonardo De Moura, and Daan Leijen, "Perceus: Garbage free reference counting with reuse," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 96–111.

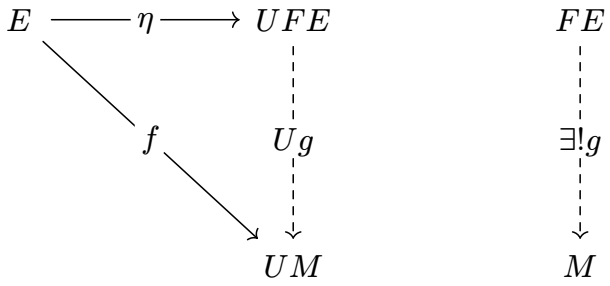
[37] Steve Awodey, *Category theory*, vol. 52. OUP Oxford, 2010.

A. Categorical Justification for ArchSem's use of the Free Monad

Let U be the forgetful functor from the category of monads to the category of endofunctors (forgetting the `pure` and `bind` operations). We will consider instruction effects E to be an endofunctor on `Type` similar to the freer monad [34] as described in Section 3.1.1. A monad M can "implement" the instruction effects if there exists an endofunctor homomorphism from E to UM.

We want to express the ISA semantics as a "generic" monad that we can later specialise into any memory-model-specific monad of our choice. In other words, we want a monad homomorphism from the generic monad to any other monad that can implement all the instruction effects. So for any instruction effect endofunctor E, we require a "generic" monad FE such that:

- There exists some endofunctor homomorphism $\eta : E \to UFE$ that embeds effects from E into the generic monad's underlying endofunctor.
- For all memory-model-specific monads M with some endofunctor homomorphism $f : E \to UM$ that can accept effects from E, there should exist a unique monad homomorphism $g : FE \to M$ that specializes the generic monad such that $f = Ug \circ \eta$, the monad homomorphism is consistent with the effect embeddings. i.e. the following diagram commutes.



By requiring that the monad homomorphism be unique, we ensure that the generic monad contains no redundancy. This property is equivalent to the canonical definition of a left adjunction $F \dashv U$, if we consider F to be a functor from endofunctors to monads [37]. A left adjunction to a forgetful functor is known as a *free-forgetful* adjunction in which we call F a free functor. Hence we shall call our F the *free monad* and FE the monad freely generated from effects E.

To appreciate the significance of the free-forgetful adjunction, consider the following analogy to linear algebra. Given an arbitrary set, say $\{a, b, c\}$, we can *freely generate* a vector space using this set as the basis vectors. The vector space would contain vectors such as $2a + 4b - c$. It just so happens that this *freely generating* transformation can be expressed as a functor left adjoint to the forgetful functor that sends a vector space to the set of its points. Just as we can freely generate a vector space from a set of basis vectors, we can freely generate a monad from a set of effects and the free-forgetful adjunction generalises this concept.

B. Proof that ArchSem Free Monad Effects are less Expressive than Freer Monad Effects

This is my original proof that defining effects like ArchSem does with a type `Eff : Type` and a function `f : Eff → Type` is less expressive than the freer monad `Eff : Type → Type` specification.

Let U be the set of values so that the set of `Type`'s is $P(U)$. I say that a freer monad effect definition $F : P(U) \to P(U)$ defines the same effect as an ArchSem style definition $\langle Y \in P(U), f : Y \to P(U) \rangle$ iff $\forall \iota \in P(U).F\iota \simeq \{y \in Y \mid fy = \iota\}$.

I will show that there exists an $F : P(U) \to P(U)$ that has no ArchSem style definition $\langle Y \in P(U), f : Y \to P(U) \rangle$ for the same effect. Namely $F(x) \triangleq x$.

I proceed by contradiction. Suppose that there exists a $\langle Y \in P(U), f : Y \to P(U) \rangle$ such that $\forall \iota \in P(U).F\iota \simeq \{y \in Y \mid fy = \iota\}$.

Expanding the definition of F gives $\forall \iota \in P(U).\iota \simeq \{y \in Y \mid fy = \iota\}$ (1).

The remainder of the proof will proceed as a generalised version of Cantor's theorem.

Let $D = \{y \in Y \mid y \notin fy\}$.

- If $D = \{\}$ then $\forall y \in Y.y \in fy$ (2).
  Let $a, b \in U$.
  By instantiating (1) with $\iota = \{a\}, \iota = \{b\}$ and $\iota = \{a, b\}$, there must exist some $s_a, s_b, s \in Y$ such that $fs_a = \{a\} \wedge fs_b = \{b\} \wedge fs = \{a, b\}$.
  By (2), it must be that $s_a = a \wedge s_b = b \wedge s \in \{a, b\}$ (3).
  So $fa = \{a\} \wedge fb = \{b\}$. So $fs = \{a\} \vee fs = \{b\}$, by (3). But this contradicts $fs = \{a, b\}$
- If $D \neq \{\}$ (4).
  By instantiating (1) with $\iota = D$ and by (4), there must be some $s \in Y$ such that $fs = D$, (5).
  - If $s \in D$ then $s \in fs$ by (5) and $s \notin fs$ by definition of D.
    Contradiction.
  - If $s \notin D$ then $s \notin fs$, by (5) and $s \in fs$ by definition of D.
    Contradiction.

■

C. ArchSem-Lean Instruction Effects

This table shows all the instruction effects I have implemented for ArchSem-Lean.

Effect Name	Arguments	Return Type
`regRead`	`(reg : Arch.register)` `(accessType : Option Arch.sys_reg_id)`	`Arch.register_type reg`

`regWrite`	`(reg : Arch.register)`
	`(accessType : Option Arch.sys_reg_id)`
	`(value: Arch.register_type reg)`
		`Unit`

`memRead`	`(memReq : MemRequest)`	`Except Arch.abort (BitVec (8 * memReq.size) × BitVec (memReq.numTag))`

`memWrite`	`(memReq : MemRequest)`
	`(value : BitVec (8 * memReq.size))`
	`(tags : BitVec (memReq.numTag))`
		`Except Arch.abort Unit`

`memWriteAnnounce`	`(memReq : MemRequest)`	`Unit`
`barrier`	`(barrier : Arch.barrier)`	`Unit`
`cacheOp`	`(op : Arch.cache_op)`	`Unit`
`tlbOp`	`(op : Arch.tlbi)`	`Unit`
`choice`	`(n : Nat)`	`Fin n`
`clockCycle`		`Unit`
`getCycleCount`		`Nat`
`translationStart`	`(translationStart : Arch.trans_start)`	`Unit`
`translationEnd`	`(translationEnd : Arch.trans_end)`	`Unit`
`archException`	`(exception : Arch.exn)`	`Unit`
`returnException`		`Unit`
`printMessage`	`(msg : String)`	`Unit`

D. Proof for the Negligible Collision Probability of my Memory Map Hashing Algorithm

My algorithm for hashing a `MemoryMap` (below) is constructed using an existing 64-bit hash function for address-value pairs. In this appendix, I will show that for two unequal memory maps, the probability they hash to the same 64-bit value using my algorithm is no more than chance ($\frac{1}{2^{64}}$) provided we assume the address-value pair hash is a random oracle. That is, for each unique input, the address-value pair hash will select a random output hash uniformly.

```
instance : Hashable MemoryMap where
  hash mem := mem.foldl (fun acc addr value => acc.xor (hash (addr, value))) 0
```

Let m_a and m_b be memory maps that are not equal (1).

We are required to prove that $P(H(m_a) = H(m_b)) = \frac{1}{2^{64}}$.

Let M_a and M_b be the set of key-value pairs in m_a and m_b. Since a memory map is uniquely defined by its set of address-value pairs, we have that $M_a \neq M_b$ (2).

Let $D := (M_a \cup M_b) \setminus (M_a \cap M_b)$. By (2), we have that $|D| > 0$ (3).

$$P(H(m_a) = H(m_b)) = P\left(\bigoplus_{(a,v)\in M_a} H((a,v)) = \bigoplus_{(a,v)\in M_b} H((a,v))\right), \text{ the hashing algorithm}$$

$$= P\left(\bigoplus_{(a,v)\in D} H((a,v)) = 0\right), \text{ hashes cancel and rearrange}$$

So it suffices to show that $P\left(\bigoplus_{(a,v)\in D} H((a,v)) = 0\right) = \frac{1}{2^{64}}$.

We conclude by showing the more general statement $\forall k. P\left(\bigoplus_{(a,v)\in D} H((a,v)) = k\right) = \frac{1}{2^{64}}$ by induction on $|D|$. By (3) we have $|D| > 0$ so we start with base case $|D| = 1$. Note that since we are only hashing elements of this set D, we do not need to worry about repeat hashes and may assume that each address-value hash is randomly assigned an output.

- Base case: $|D| = 1$.
  We have $D = \{(a', v')\}$ for some (a', v').
  $\forall k. P\left(\bigoplus_{(a,v)\in D} H((a,v)) = k\right) = P(H((a', v')) = k)$
  $= \frac{1}{2^{64}}$, by the random oracle assumption.
- Inductive case: $|D| = n + 1$.
  We have $D = \{(a', v')\} \cup D'$ for some (a', v') and D' where $|D'| = n$.
  $\forall k'. P\left(\bigoplus_{(a,v)\in D} H((a,v)) = k'\right) = P\left(\bigoplus_{(a,v)\in D'} H((a,v)) = k' \oplus H((a', v'))\right)$
  $= \frac{1}{2^{64}}$, by the inductive hypothesis with $k = (k' \oplus H((a', v')))$

■

E. Example Litmus Test in the ArchSem Format

This appendix contains an example of a litmus test in the `.archsem.toml` format which I wrote a parser for. It is a modified example test from the ArchSem-Rocq repository at: https://github.com/rems-project/archsem/blob/3eca0e8bd6587a5db11bf087c4ddf8b778a498d6/cli/tests/arm/um/MP.archsem.toml

```
arch="Arm"
name="MP"

# Thread 1
[[registers]]
_PC = 0x500
R0 = 0x1000
R1 = 0x100
R2 = 0x2a
R3 = 0x1000
R4 = 0x200
R5 = 0x1

# Thread 2
[[registers]]
_PC = 0x600
R0 = 0x1000
R1 = 0x100
R2 = 0x0
R3 = 0x1000
R4 = 0x200
R5 = 0x0

# Thread 1 Instructions
[[memory]]
addr = 0x500
kind = "code"
step = 4
data = [0xf8206822, 0xf8236885]

# Thread 2 Instructions
[[memory]]
addr = 0x600
kind = "code"
step = 4
data = [0xf8636885, 0xf8606822]

# Shared Data
[[memory]]
addr = 0x1100
step = 8
data = 0
```

```
[[memory]]
addr = 0x1200
step = 8
data = 0

[[termCond]]
_PC = 0x508

[[termCond]]
_PC = 0x608

[[outcome]]
observable.1.R5 = 0x0
observable.1.R2 = 0x2a

[[outcome]]
[outcome.observable.1]
R5 = 0x0
R2 = 0x0

[[outcome]]
observable.1.R5 = { op = "eq", val = 0x1 }
observable.1.R2 = { op = "eq", val = 0x0 }

[[outcome]]
observable.1 = { R5 = { op = "eq", val = 0x1 }, R2 = { op = "eq", val = 0x2a } }
```

F. Commands used to Generate the Litmus Tests for Evaluation

This appendix gives the `diy7` command invocations used to generate the tests suites used in the evaluation chapter. For all invocations, I'm using diy version `7.58`.

Large test corpus (15,793 tests):
This invocation of the `diy7` command generates 64-bit Arm-A litmus tests for up to three threads and a cycle length of 8. The `-save` option gives the list of edge types to include which in this case is all the edge types supported by Tiny-Arm.

```
diy7 \
  -arch AArch64 -num false -type uint64_t \
  -nprocs 3 -size 8 \
  -safe Pos**,Pod**,Rfi,Rfe,Coi,Coe,Fri,Fre,DMB.SYs**,DMB.SYd**,DMB.STs**,\
DMB.STd**,DMB.LDs**,DMB.LDd**,ISBs**,ISBd**
```

Small test corpus (108 tests):
Here I include only a subset of edge types to reduce the test size.

```
diy7 \
  -arch AArch64 -num false -type uint64_t \
  -nprocs 4 -size 7 \
  -safe Pos**,Pod**,Rfe,Coe,Fre,DMB.SYs**
```

Wide test corpus (184 tests):
The `WW+WW+WW+WW+WW` test is excluded because it is significantly larger than the others and took too long to run.

```
diy7 \
  -arch AArch64 -num false -type uint64_t \
  -nprocs 5 -size 10 \
  -safe Pod**,Rfe,Coe,Fre \
  -o herdtests
rm herdtests/WW+WW+WW+WW+WW.litmus
```

G. Profiling Details

This appendix details the program versions used for the performance comparison in the evaluation chapter.

- **ArchSem-Lean**
  - Commit hash `cf69a0d3164c8065057f538d47e6b974c719efe7` (April 2026)
- **ArchSem-Rocq**
  - Commit hash `cb186e471017cce0f1845aba13baf4486ccba147` (April 2026)
  - Built in release mode with `dune build --release`
- **Herd7**
  - Herdtools version `7.58`
  - Ocaml version `4.14.2`
  - Cat axiomatic model used: https://github.com/herd/herdtools7/blob/1ca343e16a2038e406d1ac674e7e3a1b722b36c7/herd/libdir/aarch64.cat
- **Isla-axiomatic**
  - Installed with cargo `isla v0.2.0`
  - Rust version `1.90.0`
  - Z3 version `4.8.12`
  - Sail ISA snapshot used: https://github.com/rems-project/isla-snapshots/blob/7e4bfd5ea249f8199e1602179abb89d2b728e631/aarch64.ir
  - Sail ISA config used: https://github.com/rems-project/isla/blob/75ca1ae51c7d55056118370c746c2c5cfb8a1a01/configs/aarch64.toml
  - Cat axiomatic model used: https://github.com/rems-project/isla/blob/75ca1ae51c7d55056118370c746c2c5cfb8a1a01/web/client/dist/aarch64.cat
- **Rmem**
  - Rmem version `0.1`
  - Ocaml version `4.11.2`

H. Comparing Lean and Rocq Error Messages

This appendix compares the error messages generated from Lean and Rocq by making the same type error in the promising models of ArchSem-Lean and ArchSem-Rocq.

- ArchSem-Lean commit: `cf69a0d3164c8065057f538d47e6b974c719efe7`
- ArchSem-Rocq commit: `cb186e471017cce0f1845aba13baf4486ccba147`

ArchSem-Lean change made:

```
def runEffect (tid : Nat) (initmem : InitialMem) (eff : InstructionEffect)
    : NEStateM String ProjectedModelState ((InstructionEffect.ret eff) × Option
View) :=
  match eff with
  | ...
  | .barrier (Barrier.Barrier_ISB ()) => do
    let ts := (← get).threadState
    modify ({ · with threadState := {ts with visb := max ts.visb ts.vcap} })
-   return ((), none)
+   ((), none)
  | ...
```

ArchSem-Lean resulting error message:

```
error: ArchSemTinyArm/Promising.lean:538:4: Type mismatch
  ((), none)
has type
  Unit × Option ?m.899
but is expected to have type
  NEStateM String ProjectedModelState ((InstructionEffect.barrier
(Barrier.Barrier_ISB ())).ret × Option View)
```

ArchSem-Rocq change made:

```
 Equations run_outcome (out : outcome) :
     Exec.t (PPState.t TState.t Msg.t IIS.t) string (eff_ret out * option view) :=
   | ...
   | Barrier (Barrier_ISB ()) =>
     ts ← mget PPState.state;
     mset PPState.state $ TState.update TState.visb (TState.vcap ts);;
-    mret ((), None)
+    ((), None)
   | ...
```

ArchSem-Rocq resulting error message:

```
File "./ArchSemArm/UMPromising.v", line 520, characters 2-3462:
Error: UNDEFINED EVARS:
 ?X1523==[tid initmem |- MBind ?M] (parameter MBind of mbind) {?MBind}
 ?X1528==[tid initmem ts |- MBind ?M0] (parameter MBind of mbind) {?MBind0}
 ?X1537==[tid initmem ts |-
           MCall (MState (PPState.t TState.t ?mEvent ?iis_t)) ?M1]
           (parameter MCall0 of mset) {?MCall0}
 ?X1538==[tid initmem ts |- MBind ?M1] (parameter H of mset) {?H}
```

```
 ?X1543==[tid initmem ts |- Setter PPState.state] (parameter H0 of mset) {?H0}
 ?X1549==[tid initmem |-
           MCall (MState (PPState.t TState.t ?mEvent0 ?iis_t0)) ?M2]
           (parameter MCall0 of mget) {?MCall00}
 ?X1550==[tid initmem |- FMap ?M2] (parameter H of mget) {?H0}
CONSTRAINTS:
 [tid initmem] [ts u] |- (() * option ?A@{y:=u})%type <= ?M0 ?B
[tid initmem] [ts] |- ?M1 <= ?M0
[tid initmem] [ts] |- ?M0 <= ?M
[tid initmem] [] |- ?M2 <= ?M
[tid initmem] [] |- ?M ?B <=
  PPState.t TState.t Msg.t IIS.t
  → Exec.res (PPState.t TState.t Msg.t IIS.t * string)
      (PPState.t TState.t Msg.t IIS.t *
       (eff_ret (Barrier (Barrier_ISB ())) * option view))
TYPECLASSES:?X1523 ?X1528 ?X1537 ?X1538 ?X1543 ?X1549 ?X1550
SHELF:?X1550 ?X1549 ?X1543 ?X1538 ?X1537 ?X1528 ?X1523 ?X1550 ?X1549 ?X1543
?X1538 ?X1537 ?X1528 ?X1523
FUTURE GOALS STACK:?X1554 ?X1553 ?X1550 ?X1549 ?X1548 ?X1546 ?X1545 ?X1543
?X1542 ?X1541 ?X1538 ?X1537 ?X1534 ?X1528 ?X1525 ?X1523 ?X1522
```

I. Project Proposal

ArchSem in Lean: integrating ISA and concurrency semantics

Candidate `4485B`
Project Proposal
October 2025

Blind Grading Number: `4485B`
Day-to-Day Supervisor: Thibaut Pérami <thibaut.perami@cl.cam.ac.uk>
Marking Supervisor: Peter Sewell <peter.sewell@cl.cam.ac.uk>
Director of Studies: Ramsey Faragher <rmf25@cam.ac.uk>

1. Introduction and background

"And the rain fell, and the floods came, and the winds blew and beat on that house, but it did not fall, because it had been founded on the rock." (Matthew 7:25 ESV), building on solid foundations is common wisdom reiterated in texts and proverbs throughout history. Yet, it seems that the fast moving and rapidly growing world of computing has forgotten this advice - the XZ utils backdoor, Spectre, and countless other security vulnerabilities and system failures can attest. The focus of my project is on perhaps one of the most foundational component of today's software ecosystem - the relaxed CPU architecture - and in formalising its precise semantics.

ArchSem is a framework written for the Rocq theorem proving language which integrates *Instruction Set Architecture* (ISA) semantics with concurrency models to realise a definition of multi-core CPU architecture that "suffices for foundational formal proofs, both about the architecture as a whole and about specific humanwritten or compiler-generated code" [ArchSem paper draft introduction]. It has been used to successfully prove that a user-level model is a sound abstraction of an underlying virtual-memory system model. However, it is still in early in its development, and much remains to be explored in the design space.

Lean is a general purpose functional programming language which can prove theorems using the Calculus of Constructions. In many senses, it is a modern alternative to Rocq. It has widespread adoption, comprehensive documentation, a healthy ecosystem and as of recently is able to express theorems about ISA semantics thanks to a new Sail-to-Lean backend which converts ISA semantics written in the Sail DSL to Lean. A natural next-step is to begin reasoning about multi-core CPU architectures in Lean as well.

Therefore, **my project seeks to re-implement ArchSem for Lean**, making changes where necessary to fit idiomatically in Lean. By doing this we lay the foundations for full-ISA proofs and small program proofs to be written in Lean. We will also discover if idiomatic differences of Lean compared to Rocq cause it to be better suited for the needs of ArchSem.

2. Description of work

The behaviour of multi-core architectures can largely be factored into two components. The *Instruction Set Architecture* (ISA) semantics which describes the behaviour of instructions as if they were executing on a single thread and the *concurrency model* which determines how threads view the memory operations of each other. I plan to use the Sail-to-Lean backend to generate ISA semantics in Lean, and to implement a relaxed concurrency model by hand.

2.1. Implementing the ArchSem interface in Lean

The existing Rocq interface provided by ArchSem between architecture and concurrency semantics is essentially a free monad over a given set of architectural effects such as reading or writing registers and memory.

2.2. Implementing concurrency semantics in Lean

Some concurrency semantics will need to be written in Lean. I plan to implement a sequential model, which can only run a single CPU thread. And at least one relaxed concurrency model chosen from those already implemented in Rocq. It is important that both Lean and Rocq implementations express equivalent concurrency semantics as they will later be tested against each other.

2.3. Integrating ArchSem Lean with a Sail-generated ISA semantics

The semantics for a number of instruction-set architectures are defined in Sail which has recently received support for generation of Lean code. However, it is likely some changes need to be made to this backend before it works directly with a Lean ArchSem interface. After making these changes and generating the Lean ISA model, we will have a CPU architecture semantic that encapsulates both the ISA and the concurrency behaviours.

2.4. Evaluating correctness against ArchSem

At the conclusion of this project, I expect to have integrated an ISA and a concurrency semantics to produce in Lean a model of the CPU that is equivalent to that of the analogous Rocq implementation. To increase confidence in this equivalence, I shall write some litmus tests in both Rocq and Lean that are of the form "if execution begins in this memory and register state, what is the set of possible termination states under some termination condition". We would expect the set of termination states for each implementation to be equal.

3. Success criteria

1. Model CPU architecture semantics in Lean by combining one ISA semantics with at least one relaxed concurrency model.
2. Demonstrate that at least three small litmus tests have equivalent behaviour in both the existing Rocq ArchSem implementation and the new Lean implementation.

4. Possible extensions

We have discussed a wide range of possible extension, three of which I briefly list below.

- Using the Lean framework to prove some simple properties of ISA subsets.
  - Proving that a particular subset of an ISA is deterministic.
  - Proving some of the properties required to show that Sail's 'tiny ARM' ISA provides a sound virtual-memory abstraction, similar to the proof done in the ArchSem paper.
- Adding support for other instruction sets, or extensions to the initial ISA. For example, if the RISC-V base ISA is implemented we may add RISC-V extensions.
- Support both operational and axiomatic models. These are each different approaches to implementing concurrency models.
  - Automatically generating Lean code for an axiomatic model specified in the cat DSP.

5. Work Plan

5.1. Michaelmas Term

- Weeks 1-2 (Thu 09 Oct - Wed 22 Oct)
  - **Deadline** Fri 10 Oct: project proposal deadline.
  - Learn about the Calculus of Constructions upon which both Lean and Rocq is based.
  - Learn how to write functional code in Lean.
  - Understand the fundamental and syntactic differences between Lean and Rocq.
  - **Milestone**: Complete the 'Functional Programming in Lean' and 'Theorem Proving in Lean' tutorials on https://lean-lang.org. Write some basic theorems in Rocq.
- Weeks 2-4 (Thu 23 Oct - Wed 05 Nov)
  - Familiarise myself with the existing ArchSem Rocq source code.
  - Understand how ISA and memory state is expressed in the existing ArchSem.
  - **Milestone**: Write at least one litmus test in the existing ArchSem.
- Weeks 5-6 (Thu 06 Nov - Wed 19 Nov)
  - Determine if the language differences between Rocq and Lean present any challenges in porting the ArchSem interface type.
  - **Milestone**: Write preliminary ArchSem architecture/concurrency interface type in Lean.
- Weeks 7-8 (Thu 20 Nov - Wed 03 Dec)
  - Learn about the Sail ISA specification language with a focus on the generation of theorem prover definitions.
  - **Milestone**: Generate Lean code for the semantics of one ISA from the Sail implementation and adapt to typecheck against the Lean interface.

5.2. Michaelmas Break

- Weeks 1-2 (Thu 04 Dec - Wed 17 Dec)

    - **Milestone**: Implement a relaxed concurrency model in Lean that typechecks against the Sail interface.
- Weeks 3-4 (Thu 18 Dec - Wed 31 Dec)
    - Christmas holiday
- Weeks 5-6 (Thu 01 Jan - Wed 14 Jan)
    - **Milestone**: Run litmus test in Lean and verify the outcome is the same as in Rocq.
- Weeks 7 (Thu 15 Jan - Wed 21 Jan)
    - Holiday break

5.3. Lent Term

- Weeks 1-2 (Thu 22 Jan - Wed 04 Feb)
    - Buffer time to finalise implementation of the ArchSem Lean core.
    - **Milestone**: Submit progress report.
- Weeks 3-4 (Thu 05 Feb - Wed 18 Feb)
    - **Deadline** Fri 06 Feb: progress report deadline.
    - Begin writing the Implementation chapter of the dissertation draft.
    - **Milestone**: First draft for a part of the implementation chapter covering the core of the project.
- Weeks 5-6 (Thu 19 Feb - Wed 04 Mar)
    - Decide what project extensions are worth pursuing.
    - Produce an updated plan for the remainder of the timeline.
    - **Milestone**: Agree with supervisor on the revised timeline for the remainder of the project including the extensions I plan to explore.
- Weeks 7-8 (Thu 05 Mar - Wed 18 Mar)
    - Thu 12 Mar: How to write dissertation lecture.
    - The remainder of time will be spent on extensions and on writing the dissertation.

5.4. Lent Break

- Weeks 1-2 (Thu 19 Mar - Wed 01 Apr)
- Weeks 3-4 (Thu 02 Apr - Wed 15 Apr)
- Weeks 5-6 (Thu 16 Apr - Wed 29 Apr)

5.5. Easter Term

- Weeks 1-2 (Thu 30 Apr - Wed 13 May)
- Weeks 2-4 (Thu 14 May - Wed 27 May)
    - **Deadline** Fri 15 May: source code deadline.

6. Starting Point

My experience in semantics and theorem provers is limited to the Part IB courses and reading the first few sections of a HOL-lite tutorial and a Lean tutorial respectively. I have had a brief read of the ArchSem draft paper and source code. I have some experience in writing assembly and worked on a cache-coherent interconnect software model in a first year ARM summer internship.

7. Resource Declaration

I will be writing code on a Lenovo L440 laptop equipped with an i5-520M processor and 16GiB DDR3 RAM. I have replacements for every part in this laptop and I have a spare laptop. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

All important files are replicated by two separate cloud storage providers in two separate regions. In addition to these backups, the source code and dissertation itself will be version controlled by git and backed by GitHub remotes in the rems-project organisation.

The Sail to Lean backend is a critical resource required for the completion of this project. Léo Stefanesco `<leo.stefanesco@cl.cam.ac.uk>` has read a draft of this project proposal and assures me that adapting the backend to generate Lean code for an ArchSem interface should be relatively straightforward.