Foundational Verification of Running-Time Bounds for Interactive Programs

by

Andrew Tockman

S.B. Computer Science and Engineering, Mathematics, MIT, 2023

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

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2025

Authored by: Andrew Tockman
Department of Electrical Engineering and Computer Science
May 16, 2025

Certified by: Adam Chlipala
Professor of Computer Science, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Foundational Verification of Running-Time Bounds for Interactive Programs

by

Andrew Tockman

Submitted to the Department of Electrical Engineering and Computer Science
on May 16, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

The field of formal methods has a rich history of practical application in verification of the correctness of software. Existing verification tooling operates at a wide range of rigor, from proving relatively weak properties via traditional static analysis to powerful theorem provers that can express very precise specifications. It is sometimes desirable to prove properties about programs that make reference to not just semantic behavior but also to other metaproperties of the program's execution, such as runtime or I/O histories. There is also a wide variety of existing tooling for proving bounds on program runtime. However, there is no prior work on a maximally rigorous verification system that can prove predicates involving all of semantic behavior, runtime, and I/O. Our contribution is exactly that – we extend the existing Bedrock2 framework, which implements a C-like systems language within a powerful proof engine together with a verified compiler capable of expressing arbitrary proof conditions involving behavior and I/O, and augment it to add the capacity to reason about runtime as well. As a capstone proof of concept, we apply the new metrics machinery to an IoT lightbulb controller (already verified with respect to the previous framework) and produce a new specification with time bounds based on arrival of network packets.

Thesis supervisor: Adam Chlipala
Title: Professor of Computer Science

Acknowledgments

There are many people I would like to thank, without whom this thesis would not have been possible: Adam Chlipala, my academic and thesis advisor, for his patience, excellent guidance, and extremely helpful mentorship both with this project and throughout my academic career, for which I am deeply grateful; Samuel Gruetter and Andres Erbsen, for their immediate willingness to contribute their expertise and knowledge with the Bedrock2 framework to help with this project, as well as their time spent directly working on it; and Pratap Singh, also for his collaboration with this project.

I also thank all of the aforementioned for their contributions to the paper submission that this thesis drew substantially from. Much of the text that I wrote for that paper appears in this thesis, with their minor edits, and the structure and organization of this thesis was also partially inspired by their contributions.

Additionally, I would like to thank Thomas Carotti, who laid the original foundation for metrics logging in the Bedrock2 codebase.

Finally, I would like to thank my friends from before my time at MIT, the countless people and communities I've spent time with during my time at MIT, and my family, to all of whom I owe so much of the happiness in my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and motivation

Because this project involved both theoretical and practical considerations, with interesting decisions on both fronts, I will split much of the discussion in this thesis based on that division, for the sake of clarity and intuition.

I will introduce the project by first providing some baseline background information, which a reader familiar with basic programming-language theory and formal methods can skip, and then a brief overview of the project-specific framework. Finally, I will outline the rest of the thesis at the end of this chapter.

1.1 Background

1.1.1 Theory: Big-step operational semantics

The term ***operational semantics*** refers to the act of specifying a programming language's behavior by means of providing ***inference rules*** that specify the manner in which a program is executed. Given a compiler from a high-level language to a low-level language, the definition of the semantics of each language forms the baseline for a correctness proof of the compiler – namely, if the input to the compiler satisfies some postcondition with respect to the high-level semantics, the output of the compiler should satisfy the same postcondition with respect to the low-level semantics. The language semantics also provide the definitions that allow one to formally verify behavioral properties on particular source programs, which is the ultimate practical goal of projects like this one.

Before describing the different styles of defining language semantics in detail, I will first establish a common setting for the purpose of comparing them. The illustrative programming language for the theoretical sections of this thesis will be quite similar to Bedrock2, with a few features excluded for simplicity (I will describe the difference more precisely in Section 1.2.2).

In this setting, programs have several state components:

- A ***memory*** map m from machine words, representing memory locations, to bytes.
- A ***locals*** map ℓ from strings, representing variable names, to words.
- A ***trace*** τ, which is a list of what I/O operations have occurred during the program's execution thus far. The trace is ghost state; in other words, it appears solely in the *formal* semantics, in service of allowing us to refer to I/O events when writing propositions to prove about source programs, but a concrete language compiler would not involve the trace at all.

We will have ***expressions*** that evaluate to words, which can be any of the following:

- A ***literal*** that evaluates to a given word v, denoted `literal v`.
- A ***variable*** x, denoted `var x`, which evaluates to $\ell[x]$ (the value of x in the locals map).
- A ***load*** on some expression a, denoted `load a`, which first evaluates a as an expression and then looks up the corresponding word in the memory map m.
- A binary ***operation*** denoted `op o e1 e2`, which evaluates e_1 and e_2 as expressions, then performs the operation o on them (which can be e.g. addition, comparison, etc.).

The baseline semantics includes a function $\mathsf{evalexpr}(m, \ell, e)$ that computes values of expressions. I will describe it in more detail when necessary in Section 1.2.1, but for now it is mostly self-explanatory.

Finally, we have the following ***commands***:

- The `skip` command, a no-op.
- The `set` command, which takes a string and expression, and sets the value of that local variable to the evaluation of the expression.
- The `unset` command, which takes only a string, and removes that local variable from the locals map ℓ.
- The `store` command, which takes two expressions representing an address and a value, and sets the memory at that address to that value. (For simplicity we will treat the second value as a single byte and assume addresses are always in-bounds.)

- The `seq` command, which takes two commands, and runs them sequentially.
- The `cond` command, which takes an expression and two commands, and runs the first command if the expression is nonzero and the second command otherwise.
- The `while` command, which takes an expression and a command, and runs the command if the expression is nonzero, then repeats.

Note that the trace has thus far been irrelevant. It will come into play in Section 1.2.1.

A ***configuration*** is a 4-tuple consisting of a command, a memory map, a locals map, and a trace. It represents a possibly unevaluated or partially-evaluated command together with its environment, which can be thought of as the "current state" at the time of running the command. We denote a configuration as $c/m/\ell/\tau$, extending the traditional notation of c/s representing a command c with state s.

Two commonly used strategies for defining language semantics are called ***small-step*** semantics and ***big-step*** semantics. Given an unevaluated or partially-evaluated command, a small-step operational semantics specifies all the possible ways to reduce the command by an individual step. Figure 1.1 gives the small-step semantics rules for all commands introduced so far. Conversely, a big-step operational semantics specifies the entire evaluation of an command to its fully reduced form. Figure 1.2 gives the big-step semantics rules for all commands introduced so far.

For the purposes of this project, it is sufficient to know that the specifications of the relevant languages will roughly be built upon the ideas of big-step semantics. However, I will make a few more comments here. First, note that Figure 1.1 and Figure 1.2 display a fair amount of parallelism. Some rules are markedly different, like the sequencing rule(s), but many rules are quite similar. You might hope that the two can be formally related, and in fact the ***transitive closure***[1] of the small-step judgment $\rightarrow$, which we denote as $\rightarrow^*$ and roughly means "zero or more applications of $\rightarrow$," can often be shown by structural induction to be "equivalent" to the big-step judgment $\Downarrow$ in some sense. Still, these two styles of definition are consequentially distinct. In particular, suppose you have a non-terminating command, such as an infinite loop. It is possible to apply small-step semantics rules to this expression, but there is no available big-step semantics rule to apply, since part of a big-step judgment is the fully-evaluated program state. Therefore, we will henceforth only consider terminating programs. Of course, many programs in practice are intended to run forever, like network servers; I will discuss a trick that Bedrock2 uses to sidestep this issue in Section 1.2.2.

[1] Technically, we want the *transitive-reflexive closure*, because there is no rule that says an expression can evaluate to itself.

EVAL-SET

$$\frac{\mathsf{evalexpr}(m, \ell, e) = v}{(\mathrm{set}\, p\ e)/m/\ell/\tau \to (\mathrm{skip})/m/\ell[p := v]/\tau}$$

EVAL-UNSET

$$\frac{}{(\mathrm{unset}\, p)/m/\ell/\tau \to (\mathrm{skip})/m/(\ell \setminus p)/\tau}$$

EVAL-STORE

$$\frac{\mathsf{evalexpr}(m, \ell, x) = a \qquad \mathsf{evalexpr}(m, \ell, y) = v}{(\mathrm{store}\, x\ y)/m/\ell/\tau \to (\mathrm{skip})/m[a := v]/\ell/\tau}$$

EVAL-SEQ

$$\frac{c_1/m/\ell/\tau \to c_1'/m'/\ell'/\tau'}{(c_1; c_2)/m/\ell/\tau \to (c_1'; c_2)/m'/\ell'/\tau'}$$

EVAL-SEQ-SKIP

$$\frac{}{(\mathrm{skip}; c_2)/m/\ell/\tau \to c_2/m'/\ell'/\tau'}$$

EVAL-IF-TRUE

$$\frac{\mathsf{evalexpr}(m, \ell, e) = v \qquad v \neq 0}{(\mathrm{if}\, e\, \mathrm{then}\, c_1\, \mathrm{else}\, c_2)/m/\ell/\tau \to c_1/m/\ell/\tau}$$

EVAL-IF-FALSE

$$\frac{\mathsf{evalexpr}(m, \ell, e) = v \qquad v = 0}{(\mathrm{if}\, e\, \mathrm{then}\, c_1\, \mathrm{else}\, c_2)/m/\ell/\tau \to c_2/m/\ell/\tau}$$

EVAL-WHILE-AGAIN

$$\frac{\mathsf{evalexpr}(m, \ell, e) = v \qquad v \neq 0}{(\mathrm{while}\, e\, \mathrm{do}\, c)/m/\ell/\tau \to (c; \mathrm{while}\, e\, \mathrm{do}\, c)/m/\ell/\tau}$$

EVAL-WHILE-DONE

$$\frac{\mathsf{evalexpr}(m, \ell, e) = v \qquad v = 0}{(\mathrm{while}\, e\, \mathrm{do}\, c)/m/\ell/\tau \to (\mathrm{skip})/m/\ell/\tau}$$

Figure 1.1: Small-step operational-semantics rules for example imperative language

1.1.2 Practice: Coq

In order to write an implementation of a program semantics that can be productively put to use in practice, formal-methods researchers often use some form of automated ***theorem prover*** or ***proof assistant***. Broadly, a theorem prover is a piece of software that can be fed a proof in a machine-checkable format, so that to know that a long and complicated proof is correct, one only has to trust that the statement being proved corresponds to the intended one and that the theorem prover core is correct.

The particular proof assistant we use in this project is ***Coq***. Like many other proof assistants, underlying its construction is the ***Curry-Howard isomorphism***, which is the observation that proofs of mathematical propositions can be thought of as corresponding exactly to well-typed expressions in a programming language. Fundamentally, Coq is nothing more than a programming language with an extremely strong type system and some interactive features including a language of proof tactics that streamline the "programming" process.

This type system in particular is known as the ***calculus of inductive constructions***, and I will not describe it in detail here. The relevance is that it allows us to define new ***inductive types***, a powerful type-generating mechanism. An inductive type is defined by a

EVAL-SKIP

$$\frac{}{(\text{skip})/m/\ell/\tau \Downarrow m/\ell/\tau}$$

EVAL-SET

$$\frac{\mathsf{evalexpr}(m, \ell, e) = v}{(\text{set}\, p\ e)/m/\ell/\tau \Downarrow m/\ell[p := v]/\tau}$$

EVAL-UNSET

$$\frac{}{(\text{unset}\, p)/m/\ell/\tau \Downarrow m/(\ell \setminus p)/\tau}$$

EVAL-STORE

$$\frac{\mathsf{evalexpr}(m, \ell, x) = a \qquad \mathsf{evalexpr}(m, \ell, y) = v}{(\text{store}\, x\ y)/m/\ell/\tau \Downarrow m[a := v]/\ell/\tau}$$

EVAL-SEQ

$$\frac{c_1/m/\ell/\tau \Downarrow m'/\ell'/\tau' \qquad c_2/m'/\ell'/\tau' \Downarrow m''/\ell''/\tau''}{(c_1; c_2)/m/\ell/\tau \Downarrow m''/\ell''/\tau''}$$

EVAL-IF-TRUE

$$\frac{\mathsf{evalexpr}(m, \ell, e) = v \qquad v \neq 0 \qquad c_1/m/\ell/\tau \Downarrow m'/\ell'/\tau'}{(\text{if}\, e\, \text{then}\, c_1\, \text{else}\, c_2)/m/\ell/\tau \Downarrow m'/\ell'/\tau'}$$

EVAL-IF-FALSE

$$\frac{\mathsf{evalexpr}(m, \ell, e) = v \qquad v = 0 \qquad c_2/m/\ell/\tau \Downarrow m'/\ell'/\tau'}{(\text{if}\, e\, \text{then}\, c_1\, \text{else}\, c_2)/m/\ell/\tau \Downarrow m'/\ell'/\tau'}$$

EVAL-WHILE-AGAIN

$$\frac{\mathsf{evalexpr}(m, \ell, e) = v \qquad v \neq 0 \qquad c/m/\ell/\tau \Downarrow m'/\ell'/\tau' \qquad (\text{while}\, e\, \text{do}\, c)/m'/\ell'/\tau' \Downarrow m''/\ell''/\tau''}{(\text{while}\, e\, \text{do}\, c)/m/\ell/\tau \Downarrow m''/\ell''/\tau''}$$

EVAL-WHILE-DONE

$$\frac{\mathsf{evalexpr}(m, \ell, e) = v \qquad v = 0}{(\text{while}\, e\, \text{do}\, c)/m/\ell/\tau \Downarrow m/\ell/\tau}$$

Figure 1.2: Big-step operational-semantics rules for example imperative language

```
Inductive list (A : Type) : Type :=
  | nil : list A
  | cons : A → list A → list A.
```

Figure 1.3: Inductive definition of list type

list of all possible ***constructors***, where the resulting type is the smallest set that satisfies all rules implied by the constructors.

As an example, Figure 1.3 gives an inductive definition of the type `list A`, i.e. the type of lists of entries of type `A`, in Coq syntax. It has two constructors, which you can think of as representing "rules." The first rule states that `nil` must be a `list A`. The second rule states that if `a` has type `A` and `as` has type `list A`, then `cons a as` also has type `list A`. So we can conclude from this definition that e.g. `cons 1 (cons 5 (cons 9 nil))` has type `list nat`.

As a short detour, I will briefly comment on the significance of the term "smallest set" in the definition of inductive types above. Any set satisfying the two rules from Figure 1.3 clearly must be infinite, but there are many different such sets that satisfy those rules. For instance, the set of finite lists of `A` is one, but so is the set that contains all finite and infinite lists of `A`, or even the set that contains all finite lists of `A` plus lists of the form "a finite list of `A` concatenated with an infinite list of some particular `a : A`." To disambiguate, we pick the *smallest* such set – intuitively, start by putting in the elements given by the non-self-referential rules, in this case just `nil`, and then add only the elements that the self-referential rules force you to add. In this case, we get the type of finite lists. If instead we took the *largest* possible set, we would get a ***coinductive type***; in this case, the type of finite or infinite lists.

Given a value of an inductive type, we know it must have been made with one of the given constructors. So if we want to prove a proposition about all values of an inductive type, we can prove it for each constructor individually, assuming that the proposition holds for the parameters of the same type in the case of the self-referential ones. This is the principle of mathematical (structural) induction, hence the name. Via the Curry-Howard isomorphism, induction corresponds to recursion in the programming-language setting, providing a concrete proof strategy. (Dually, coinductive types give rise to proofs by coinduction and corecursive functions, which has very interesting consequences in the context of program verification – for instance, a corecursively-defined big-step semantics enables reasoning about non-terminating programs with infinitely large evaluation trees. However, this is far beyond the scope of this thesis.)

Returning to the project at hand, big-step inference rules like those in Figure 1.2 can

```
Inductive exec : cmd → mem → locals → trace →
                 mem → locals → trace → Prop :=
  | EvalSkip : forall m l t, exec skip m l t m l t
  | EvalSet : forall m l t x e v,
      eval_expr m l e = Some v →
      exec (set x e) m l t m (map.put l x v) t
  | EvalSeq : forall c1 c2 m l t m' l' t' m'' l'' t'',
      exec c1 m l t m' l' t' →
      exec c2 m' l' t' m'' l'' t'' →
      exec (seq c1 c2) m l t m'' l'' t''
  [...]
```

Figure 1.4: Selected rules in inductive definition of big-step semantics judgment

SMALL-INPUT-WRONG

$$\overline{(\text{input}\, x)/m/\ell/\tau \to (\text{skip})/m/\ell[x := n]/\tau +\!\!+ (\mathsf{IN}\ n)}$$

BIG-INPUT-WRONG

$$\overline{(\text{input}\, x)/m/\ell/\tau \Downarrow m/\ell[x := n]/\tau +\!\!+ (\mathsf{IN}\ n)}$$

Figure 1.5: Incorrect attempts at introducing nondeterminism to traditional semantics

be translated directly into inductive constructors. Figure 1.4 shows a code example in Coq syntax, truncated for brevity. The use of inductive types to encode inference rules provides a very streamlined environment for writing formally verified proofs about these semantics, by structural induction on the `exec` judgment.

1.2 Framework

Building on the background from Section 1.1, I will now establish the existing framework that the system for verifying runtimes was constructed on top of.

1.2.1 Theory: Big-step omnisemantics

As motivation for omnisemantics, consider a hypothetical extension to our example language: a command $(\text{input}\, x)$, which reads a value input by the user and stores it in the variable x. The most natural attempt at adding this command to the language semantics is shown in Figure 1.5.

It may seem that the trace ghost state fully accounts for this nondeterminism – since

the value that was read is recorded in the trace, we can't "cheat" and use a different value than the one given. Unfortunately, Figure 1.5 does not capture the desired semantics. We can still prove false facts about a program's execution, to conclude that e.g. the program `input x; while x do skip` always halts by applying either inference rule with $n = 0$.

The solution is ***omnisemantics*** – a surprisingly similar but much less widely studied variant on traditional operational semantics [Charguéraud et al. 2023]. Essentially, the key insight of omnisemantics is to replace the right-hand side of the evaluation judgment with a *set of possible results* instead of a single result. This set is permitted to contain results that can never occur, but it is required to contain all results that might occur. In this sense, it is actually an *overapproximation* of possible result states.

For instance, $(\text{skip})/m/\ell/\tau \Downarrow \{m/\ell/\tau\}$ is a valid omni-big-step inference rule, simply copying the ordinary big-step inference rule and wrapping the right-hand side in a set. However, $(\text{skip})/m/\ell/\tau \Downarrow \{m'/\ell'/\tau' \mid \text{any } m', \ell', \tau'\}$ is also a valid omni-big-step inference rule, where the right-hand side is the set of all configurations – though it is probably less useful, as after applying it all information about the current state is lost. The most general statement of the omni-big-step inference rule for this command is $(\text{skip})/m/\ell/\tau \Downarrow Q$, with the premise $m/\ell/\tau \in Q$.

I have been treating the overapproximating configuration space Q as a mathematical set. However, we can also think of Q as a predicate on states, where $Q(m, \ell, \tau)$ iff $m/\ell/\tau$ is in the set. This formulation is more natural in the formal-methods setting, and it is logically equivalent, so I will switch freely between the two notations. Figure 1.6 shows the omni-big-step rules, in functional notation, corresponding to the ordinary big-step rules from Figure 1.2.

None of the rules in Figure 1.6 actually took advantage of the new machinery, since all of these rules were deterministic anyway. But equipped with this framework, we can now write a correct inference rule for the $(\text{input}\, x)$ command, shown in Figure 1.7. The crucial difference is the for-all quantifier in the premise, which ensures that all possible inputs are accounted for.

1.2.2 Practice: Bedrock2

I have organized this introduction to lead into the programming language we use in this project, Bedrock2, as smoothly as possible. In brief, Bedrock2 is a C-like programming language implemented in Coq with omni-big-step semantics quite similar to those presented in Figure 1.6. The full Coq definition of Bedrock2's semantics is reproduced in Section A.1. I will briefly summarize the parts I have omitted in the discussion thus far:

EVAL-SKIP
$$\frac{Q(m, \ell, \tau)}{(\text{skip})/m/\ell/\tau \Downarrow Q}$$

EVAL-SET
$$\frac{\mathsf{evalexpr}(m, \ell, e) = v \qquad Q(m, \ell[p := e], \tau)}{(\text{set}\, p\ e)/m/\ell/\tau \Downarrow Q}$$

EVAL-UNSET
$$\frac{Q(m, \ell \setminus p, \tau)}{(\text{unset}\, p)/m/\ell/\tau \Downarrow Q}$$

EVAL-STORE
$$\frac{\mathsf{evalexpr}(m, \ell, x) = a \qquad \mathsf{evalexpr}(m, \ell, y) = v \qquad Q(m[a := v], \ell, \tau)}{(\text{store}\, x\ y)/m/\ell/\tau \Downarrow Q}$$

EVAL-SEQ
$$\frac{c_1/m/\ell/\tau \Downarrow Q' \qquad (\forall m'\ \ell'\ \tau'.Q'(m', \ell', \tau') \to c_2/m'/\ell'/\tau' \Downarrow Q)}{(c_1; c_2)/m/\ell/\tau \Downarrow Q}$$

EVAL-IF-TRUE
$$\frac{\mathsf{evalexpr}(m, \ell, e) = v \qquad v \neq 0 \qquad c_1/m/\ell/\tau \Downarrow Q}{(\text{if}\, e\, \text{then}\, c_1\, \text{else}\, c_2)/m/\ell/\tau \Downarrow Q}$$

EVAL-IF-FALSE
$$\frac{\mathsf{evalexpr}(m, \ell, e) = v \qquad v = 0 \qquad c_2/m/\ell/\tau \Downarrow Q}{(\text{if}\, e\, \text{then}\, c_1\, \text{else}\, c_2)/m/\ell/\tau \Downarrow Q}$$

EVAL-WHILE-AGAIN
$$\frac{\begin{array}{c}\mathsf{evalexpr}(m, \ell, e) = v \\ v \neq 0 \qquad c/m/\ell/\tau \Downarrow Q' \qquad (\forall m'\ \ell'\ \tau'.Q'(m', \ell', \tau') \to (\text{while}\, x\, \text{do}\, c)/m'/\ell'/\tau' \Downarrow Q)\end{array}}{(\text{while}\, x\, \text{do}\, c)/m/\ell/\tau \Downarrow Q}$$

EVAL-WHILE-DONE
$$\frac{\mathsf{evalexpr}(m, \ell, e) = v \qquad v = 0 \qquad Q(m, \ell, \tau)}{(\text{while}\, x\, \text{do}\, c)/m/\ell/\tau \Downarrow Q}$$

Figure 1.6: Omni-big-step operational semantics rules for example imperative language

EVAL-INPUT
$$\frac{\forall n.Q(m, \ell[x := n], \tau +\!\!+ (\mathsf{IN}\ n))}{(\text{input}\, x)/m/\ell/\tau \Downarrow Q}$$

Figure 1.7: Correct implementation of nondeterminism using omni-big-step semantics

- There is an `inlinetable` expression, which is effectively a load from a piece of memory included within the code.
- There is an `ite` expression, an expression version of if-then-else.
- There is a `stackalloc` command, a nondeterministic heap allocation operation that returns a separate block of memory (in the separation-logic sense).
- Programmers can define functions in Bedrock2, and there is a `call` command which calls them.
- There is an `interact` command, which to greatly simplify is an "external" function call that records an I/O action in the program trace.
- Outside of the semantics specification, there is also a slightly idiosyncratic event-loop mechanism, which allows writing proofs about invariants preserved by an infinite-looped body. This is how Bedrock2 circumvents the aforementioned issue about the inability of big-step semantics to handle nontermination. We use this mechanism to prove correctness theorems about programs such as network servers that run forever.

The Bedrock2 project involves not just the Bedrock2 language, but also two other languages. A slightly lower-level language called FlatImp (designed specifically for Bedrock2) is involved in intermediate stages of the compiler, which ultimately compiles down to RISC-V machine code. We therefore also have formally specified semantics for FlatImp and RISC-V.

The compiler is written in Coq, and it is *end-to-end verified*. This means that there is a proof that given a postcondition with respect to the Bedrock2 omni-big-step semantics, the compiled code will satisfy the same postcondition (after appropriate abstraction transformations) with respect to the RISC-V omni-big-step semantics. Since a configuration contains a trace term, these postconditions can make arbitrary logical statements about the I/O trace and the relationship to the program's behavior.

1.3 Project overview

Having established the framework we will be working within, I can finally discuss the contributions that this project adds on top.

We augment the verified compiler pipeline to support postconditions that talk about not only program behavior and input/output history, but also *upper bounds on runtime*, all simultaneously and with the maximum possible generality of specification. We achieve this by means of metrics logging, which tracks these runtime bounds by means of a counter

mechanism. The compiler remains end-to-end verified, so that runtime bounds at the source semantics level translate to runtime bounds at the RISC-V level.

We are now fully prepared to dive in to the discussion of metrics logging. In Chapter 2, I first briefly summarize some pieces of related work and clarify the novel contribution of this project. In Chapter 3, I describe how we build a metrics-logging framework and augment the language semantics. In Chapter 4, I explain the threading of these metrics proofs through all the stages of the compiler pipeline. In Chapter 5, I show how the newly developed metrics-aware framework can be put into practice to produce concrete runtime proofs, giving two proof-of-concept case studies that demonstrate the successful application of the framework. Finally, in Chapter 6, I list some potential ways to continue this project and how it would be possible to improve upon it still.

Chapter 2

Related work

At this point, I should provide an explanation regarding what the title of this thesis means. Chapter 1 provides sufficient background to understand the two key words.

By ***foundational***, we mean that our framework is proven from as close to first principles as possible. A formal verification is only as trustworthy as its least trusted component, and traditional runtime analyses might have many: static analysis tools, or unverified compilers, or axioms used in proofs, or proof checkers themselves. All of these might be buggy, and any one faulty component might undermine the soundness of the entire toolchain. Since the Bedrock2 language, compiler, verification tools, and applications are all implemented directly in Coq, our entire trusted base consists of the Coq verifier and the correctness of our definitions within it, the smallest set one could reasonably hope for. The terminology "foundational" comes from [Appel 2001].

By ***interactive***, we mean programs that dynamically engage in input and output while they run. This precludes standard notions of runtime bounds, which typically speak in terms of a single input given at the start of the program. We would like to be able to write proofs about physical systems, such as that in Section 5.4, which depend on external factors such as how long it takes for network packets to arrive. Furthermore, we would like complete freedom to use this information in our runtime specifications.

We believe that this work is the first end-to-end verified software toolchain with these key properties. There have, however, been many related projects similar to ours; here we briefly summarize several of them.

[Necula and Lee 1998; Crary and Weirich 2000] represents some of the earliest related work, the former addressing runtime verification incidentally and the latter building upon the former. More specifically, Crary and Weirich employ Necula and Lee's idea of proof-carrying code (PCC) to build a certifying compiler that takes code annotated with runtime data and output a certified executable, which can later be checked. Once a certified executable

has been produce, the trusted base required to check it is pleasantly small; but since the compiler is not verified, it might produce code that behaves incorrectly or fails the time-bound correctness check, and additionally the time bounds cannot depend on I/O.

[Bonenfant et al. 2007] provides upper bounds on runtime for the functional language Hume. It accomplishes this by compiling to imperative instructions (on the Hume Abstract Machine, or HAM), then to C, then to assembly; and then using the commercial tool aiT to derive upper bounds (via abstract interpretation) which the authors then translate back to the Hume level. This yields guarantees on worst-case runtime, which the paper shows are empirically quite accurate for actual runtimes. This is a significant advantage over our approach, which gives very loose bounds, but it is neither foundational – the trusted base is very large indeed given that none of the compilers or aiT are verified – nor interactive – time bounds are constants and cannot even reference the values of input variables, for instance.

[Ayache, Amadio, and Régis-Gianas 2012; Cuoq et al. 2012; Amadio et al. 2014] is a collection of work building upon each other, which roughly allows automatically annotating C source code with code that increments instruction counters, which are then analyzed at the machine-code level. This work is "more foundational" than those mentioned so far, in that correctness for this entire process is verified by a combination of the proof assistant Matita and automated theorem provers. However, the pipeline is a piecemeal combination of tools requiring the output of one to be fed manually to the next, introducing an extra needed layer of trust, whereas our framework is end-to-end and can generate a single theorem – capable of referring to any aspect of the program's execution, including I/O – all inside Coq.

[Maroneze et al. 2014] is a similar example of foundational work, built on a similar C-based toolchain, which is capable of automatically generating time bounds on entire C programs, formally verified to be correct. The ability to automate this entire process is at the same time both a significant advantage over our approach, as proving a time bound in Bedrock2 requires substantial human proof effort; and a significant disadvantage, as it can only generate constant bounds (not even dependent on the input).

[Haslbeck and Lammich 2022] may be the work most similar to ours, as it presents a foundational verification of time bounds on LLVM programs. In particular, the higher-level source language comes equipped with a nondeterminism monad, which allows the specification of different abstract quantities of time usage depending on the result of a computation. No other examples listed involved reasoning about nondeterminism to any extent. The Bedrock2 omnisemantics style of nondeterminism differs, though, in that it allows much more general reasoning: while Haslbeck and Lammich's framework can only accept a map from computation result to time usage, our framework can deal with arbitrary Coq expressions involving any combination of data values and I/O trace. One other difference is that

our work extends all the way down to machine code, and it therefore captures subtleties such as costs of register spilling not captured at the LLVM level in any way.

Chapter 3

Implementation of metrics logging

Now we are ready to summarize how we extend the full Bedrock2 stack with support for metrics logging. The basic structure used to track metrics throughout the program logic and compiler is the `MetricLog` object, which is a 4-tuple of integers (*instructions*, *stores*, *loads*, *jumps*). These components refer to the number of RISC-V instructions, memory stores, memory loads (including instruction fetches), and jump instructions respectively. Most of our development can be considered parametric in the details of metrics to track, setting the stage for follow-on work that tracks more detailed metrics as needed for more precise modeling of modern processors. For instance, we might want to store some information on memory-access locality, to predict cache behavior.

We track metrics at this level of granularity because it is coarse enough to be reasonable to work with in practice, while simultaneously being precise enough to derive nontrivial wall-clock time upper bounds on realistic hardware. For instance, for load and store instructions, we can account for memory caches by using worst-case latencies for accessing main memory. For microcontrollers with modest use of caches, these bounds may be precise enough to be useful. To model nontrivial processor pipelines where timing depends on which other instructions are running, we can also use worst-case timing. Again, a natural future-work direction is tracking more granular metrics to improve on such modeling.

It is important to note that we only concern ourselves with *upper bounds* on program metrics. This choice greatly simplifies the handling of optimization stages in the compiler, as the high-level operational semantics need not concern themselves with how much optimization might occur. However, we recognize that in some scenarios, such as in the design of cryptographically safe protocols, matching lower bounds might be desirable. This capability is currently out of the scope of this work.

Recall from Chapter 1 that Bedrock's baseline semantics works with configurations of the form $c/m/\ell/\tau$.

- c is a command, i.e. a piece of Bedrock2 source code.
- m is a memory state, i.e. a partial map from memory locations to values.
- ℓ is a set of local variables, i.e. a partial map from identifiers to values.
- τ is an I/O trace, i.e. the list of input and output events so far.

To implement metrics logging, we augment the configuration type to a 5-tuple $c/m/\ell/\tau/\mu$. The new entry μ is a `MetricLog` object, as detailed above. Like the I/O trace, it can be thought of as the metrics "used so far." I will elaborate on this perspective further in Section 3.1 and Section 4.1.

With metric logs in the configuration type, we may now simply use the same proof machinery as correctness proofs, while also allowing postconditions to talk about the difference between starting and ending metrics after an evaluation. Crucially, the postcondition remains an arbitrary Coq expression – which means timing statements have complete access to the memory state, local variables, and I/O trace of the program. Metrics bounds may therefore be as coarse- or fine-grained as desired, from static constant bounds to highly specific predicates that incorporate the entire I/O history of the program execution.

3.1 Theory: Metrics-aware big-step omnisemantics

3.1.1 Expressions

Bedrock2 features a language of pure expressions, which is simple enough that semantics can be deterministic and generally standard. They make a good opportunity to introduce the main ideas of how we model metrics.

Recall again from Chapter 1 that the function $\mathsf{evalexpr}$ computes values of expressions. For instance:

$$\mathsf{evalexpr}(m, \ell, \text{literal } v) = v$$
$$\mathsf{evalexpr}(m, \ell[x \mapsto v], \text{var } x) = v$$

We extend $\mathsf{evalexpr}$ with metrics μ as an extra input and output.

$$\mathsf{evalexpr}(m, \ell, \text{literal } v, \mu) = (v, C_{\text{LIT}}(\mu))$$
$$\mathsf{evalexpr}(m, \ell[x \mapsto v], \text{var } x, \mu) = (v, C_{\text{GET}}(\mu))$$

Functions C_{LIT} and C_{GET} compute worst-case timing effects of all of the ways we anticipate their respective operations might be compiled. At the lower levels of the compiler pipeline, the language semantics can determine the appropriate value of the $C\_$ cost functions directly, by simply writing down the appropriate numbers based on the instructions emitted. In order to enable metrics proofs about Bedrock2 source programs that transfer fully down to the RISC-V level, we then propagate these costs up the compiler pipeline with as tight of bounds as is possible in each compiler stage. This flow leads to various peculiarities in the top-level definitions for the Bedrock2 semantics.

For instance, the precise number of memory accesses for a basic arithmetic operation at the Bedrock2 level depends on whether the operands are already stored in registers. We could simply assume no variables will be stored in registers and give pessimal bounds on each use of a variable, but the resulting bounds would be much looser than desired. Instead, we do slightly better by establishing a naming convention on Bedrock2-level variables. Specifically, any Bedrock2 variable whose name is prefixed with `reg_` is required to be placed in a register by the register allocator, or else the register-allocation compiler phase fails. This way, a Bedrock2-level metrics proof can safely use the tighter bounds for these variables, and the proof will correctly propagate down the pipeline.

In order to implement this strategy in practice, the semantics of all languages in the Bedrock2 stack are parameterized over a value `isReg`, which is a boolean function on variable names that determines whether they should be considered to be "register variables." At the Bedrock2 level, we use `isRegStr`, which simply checks whether the (string) name starts with `reg_`. Recall from Section 1.2.2 that Bedrock2 also includes an intermediate language called FlatImp; at that level, variable names start as strings (for which we still use `isRegStr`) but turn into integers in the register-allocation phase. Here we use `isRegZ`, which checks whether the integer is less than 32 (these are the ones that compile down to registers at the RISC-V level).

Since we prove in each compiler phase that `isReg` will never go from true to false for a given variable (note that the converse is safe and manifests in-practice), the $C\_$ cost functions can safely access this information and use it to compute the best known bound for each variable access. Hence, we amend the cost functions to also take variable names as parameters:

$$\mathsf{evalexpr}(m, \ell[x \mapsto v], \operatorname{var} x, \mu) = (v, C_{\text{SET}}(\texttt{isReg}, x, \mu))$$

The remaining cases of evalexpr are modified similarly. The full definition, both before and after the metrics augmentation, is reproduced in Appendix A.

3.1.2 Commands

The same approach adapts quite directly to the omnisemantics of commands, where in a sense it picks up compatibility with I/O-trace reasoning "for free." The omni-big-step semantics rules from Figure 1.6 can all be modified this way, resulting in the metrics-aware omni-big-step semantics rules in Figure 3.1. Note that the special value ? appears in some cost-function parameters. This value represents a variable name that is not yet known, so must be assumed to be a non-register. For instance, the cost function C_{IF} is parameterized at the FlatImp/RISC-V level by the variables involved in the test comparison, of which there might be one (for a unary condition) or two (for a binary condition). Unfortunately, at the Bedrock2 level, we only know that the condition is an expression, which might or might not be a primitive comparison that compiles directly to a FlatImp if. Therefore, we make the pessimal assumption that nothing involved in the comparison comes from a register. Improving precision in these areas to get tighter Bedrock2-level bounds would be a natural future-work direction.

In addition to the `isReg` parameter, each language semantics is also parameterized over a value `phase` (not shown explicitly in Figure 3.1), representing the compiler phase to compute metrics bounds with respect to. This mechanism is necessary because in some cases, compiler phases can *increase* metrics costs. In particular, the spilling phase adds a preamble and postamble to every function body and function call, which represents an unavoidable increase in bounds we must account for somewhere.

Therefore, we currently pass a `phase` that is simply either `PreSpill` or `PostSpill`. If the phase is `PreSpill`, then the relevant cost functions in the semantics are artificially more expensive, and the specification of a function call also has added padding. In other words, to prove a metrics bound on a high-level source program, the user must write a proof with respect to a semantics where these operations take longer than they "should." Then, we arrange the compiler pipeline so that `phase` changes exactly once, from `PreSpill` to `PostSpill` exactly at the spilling phase.

At every compiler phase where this parameter does not change, it is effectively invisible, so the same proof goes through. However, at the spilling step, we get a proof obligation that is weaker than the hypothesis; i.e. we only need to prove the metrics bound with respect to a semantics where function calls take the correct amount of time. So we offset the fixed preamble and postamble cost this way, without ever producing a visible effect in the postcondition itself.

EVAL-SKIP

$$\frac{Q(m, \ell, \tau, \mu)}{(\text{skip})/m/\ell/\tau/\mu \Downarrow Q}$$

EVAL-SET

$$\frac{\mathsf{evalexpr}(m, \ell, e, \mu) = (v, \mu') \qquad Q(m, \ell[p := v], \tau, C_{\text{SET}}(\texttt{isReg}, p, ?, \mu'))}{(\text{set}\, p\ e)/m/\ell/\tau/\mu \Downarrow Q}$$

EVAL-UNSET

$$\frac{Q(m, \ell \setminus p, \tau, \mu)}{(\text{unset}\, p)/m/\ell/\tau/\mu \Downarrow Q}$$

EVAL-STORE

$$\frac{\begin{array}{c}\mathsf{evalexpr}(m, \ell, x, \mu) = (a, \mu') \\ \mathsf{evalexpr}(m, \ell, y, \mu') = (v, \mu'') \qquad Q(m[a := v], \ell, \tau, C_{\text{STORE}}(\texttt{isReg}, ?, ?, \mu''))\end{array}}{(\text{store}\, x\ y)/m/\ell/\tau/\mu \Downarrow Q}$$

EVAL-SEQ

$$\frac{c_1/m/\ell/\tau/\mu \Downarrow Q' \qquad (\forall m'\ \ell'\ \tau'\ \mu'.Q'(m', \ell', \tau', \mu') \to c_2/m'/\ell'/\tau'/\mu' \Downarrow Q)}{(c_1; c_2)/m/\ell/\tau/\mu \Downarrow Q}$$

EVAL-IF-TRUE

$$\frac{\mathsf{evalexpr}(m, \ell, e, \mu) = (v, \mu') \qquad v \neq 0 \qquad c_1/m/\ell/\tau/C_{\text{IF}}(\texttt{isReg}, ?, ?, \mu') \Downarrow Q}{(\text{if}\, e\, \text{then}\, c_1\, \text{else}\, c_2)/m/\ell/\tau/\mu \Downarrow Q}$$

EVAL-IF-FALSE

$$\frac{\mathsf{evalexpr}(m, \ell, e, \mu) = (v, \mu') \qquad v = 0 \qquad c_2/m/\ell/\tau/C_{\text{IF}}(\texttt{isReg}, ?, ?, \mu') \Downarrow Q}{(\text{if}\, e\, \text{then}\, c_1\, \text{else}\, c_2)/m/\ell/\tau/\mu \Downarrow Q}$$

EVAL-WHILE-AGAIN

$$\frac{\begin{array}{c}\mathsf{evalexpr}(m, \ell, e, \mu) = (v, \mu') \qquad v \neq 0 \qquad c/m/\ell/\tau/\mu' \Downarrow Q' \\ (\forall m'\ \ell'\ \tau'\ \mu''.Q'(m', \ell', \tau', \mu'') \to (\text{while}\, e\, \text{do}\, c)/m'/\ell'/\tau'/C_{\text{WHILE-AGAIN}}(\texttt{isReg}, ?, ?, \mu'') \Downarrow Q)\end{array}}{(\text{while}\, e\, \text{do}\, c)/m/\ell/\tau/\mu \Downarrow Q}$$

EVAL-WHILE-DONE

$$\frac{\mathsf{evalexpr}(m, \ell, e, \mu) = (v, \mu') \qquad v = 0 \qquad Q(m, \ell, \tau, C_{\text{WHILE-DONE}}(\texttt{isReg}, ?, ?, \mu'))}{(\text{while}\, e\, \text{do}\, c)/m/\ell/\tau/\mu \Downarrow Q}$$

Figure 3.1: Metrics-aware omni-big-step operational semantics rules for example imperative language

$$\textsc{alternative-if-true}\quad \frac{\mathsf{evalexpr}(m,\ell,e) = (v,\mu_1) \qquad v \neq 0 \qquad c_1/m/\ell/\tau \Downarrow_{\mu_2} Q}{(\mathrm{if}\, e\, \mathrm{then}\, c_1\, \mathrm{else}\, c_2)/m/\ell/\tau \Downarrow_{C_{\text{IF}}(\texttt{isReg},?,?,\mu_1+\mu_2)} Q}$$

Figure 3.2: Alternative formulation of metrics-logging within omni-big-step semantics

3.1.3 Design alternatives

One possibly more natural style one might imagine is to attach metrics to the inference rules of the semantics itself, since the semantics corresponds so closely to the execution of a program. As an arbitrary example, it may be more intuitive to write the branch rule EVAL-IF-TRUE from Figure 3.1 as instead an ordinary omni-big-step rule with a single metrics object threaded through, where for each operation and inductive constructor, instead of taking a starting metrics and giving an ending metrics, we simply return an upper bound on how long that step took, as in Figure 3.2

There are two reasons this approach is markedly less convenient. First, we can directly observe that in the EVAL-IF-TRUE rule, the metrics modifications are made in series: first μ to μ', then μ' to $C_{\text{IF}}(\mu')$, then $C_{\text{IF}}(\mu')$ to the argument of the postcondition. Conversely, in the ALTERNATIVE-IF-TRUE rule, the metrics modifications are made in parallel: μ_1 from the expression evaluation and μ_2 from the recursive semantic judgment are added together. In practice, when proving concrete programs, this style would result in highly branching expressions that do not lend themselves to proof automation nearly as well as the highly linear expressions resulting from our actual formulation.

The second advantage to our style is that it mirrors the structure of the inductive proofs used internally in e.g. the compiler-correctness proof. When proving a fact by structural induction on the evaluation judgment, our style automatically relates the metrics of the goal judgment with precisely the desired starting point in the inductive hypotheses. The precise meaning of this difference is more evident in Section 4.1.1.

3.2 Practice: Adaptation of Bedrock2 semantics

I will now briefly review the details of the concrete implementation of this theory within the Bedrock2 codebase. I will split the discussion by source file, as a natural conceptual division of the affected components.

First, I will note that the Bedrock2 codebase is split cleanly into a "bedrock2" half, which deals exclusively with the Bedrock2 language itself; and a "compiler" half, which

deals with the compiler and all related paraphernalia (such as the intermediate FlatImp language). Within the "bedrock2" half, it is sometimes desirable to retain access to the old metrics-unaware semantics – for instance, perhaps a separate project that doesn't care about runtime uses Bedrock2 as a library. We therefore separate the metrics-aware versions of the program logic, semantics, weakest-precondition predicate, and loop lemmas into separate modules, compatible with the rest of the codebase. While this structure results in some code duplication between the metrics and non-metrics versions of these modules, it allows timing proofs to be merged into the main branch without affecting existing proofs.

3.2.1 Breakdown of changes

3.2.1.1 MetricLogging component. The first two components will be unique to the metrics version of the language, so they have no corresponding metrics-unaware version. The file `MetricLogging.v` provides the basic definitions of metric logs that the rest of the framework is built upon. It also supplies a range of helper functions, notations, lemmas, and proof tactics.

3.2.1.2 MetricCosts component. Again, this component is unique to the metrics version of Bedrock2. The file `MetricCosts.v` deals with the cost functions denoted $C\_$ in Section 3.1. Here, we define the cost functions, define the `isReg` predicates, and provide a few helper tactics.

3.2.1.3 MetricSemantics component. The remaining components are metric-adapted variants of already-existing Bedrock2 modules. (The naming scheme is consistent: the metrics version of `XYZ.v` is `MetricXYZ.v`.) The file `MetricSemantics.v` defines the metrics-aware omni-big-step semantics for Bedrock2 described in Section 1.2. The relevant portions of this file are reproduced in Appendix A. It also provides a substantial number of lemmas related to the language semantics.

This file imports the baseline `Semantics.v` file for some auxiliary definitions shared between the metrics-aware and metrics-unaware semantics, accounting for the 90 removed lines in Table 3.1.

3.2.1.4 MetricProgramLogic component. The file `MetricProgramLogic.v` primarily provides a library of tactics for proving correctness of Bedrock2 source programs, culminating in the `straightline` tactic, which automates away the verification of most straightline code. The changes here are almost exclusively patches to tactic automation scripts to account for the fact that e.g. several functions now take an additional parameter (the metrics log). These

changes, despite often being so simple (an extra underscore to represent an extra parameter), also ended up being the most devious to debug, in that a single missed underscore could result in spurious failures at seemingly random points within tactic scripts caused by missed branches in proof automation.

The 14 deletions in Table 3.1 actually come from an unrelated backend change that happens to have not yet propagated to the metrics-unaware version.

3.2.1.5 MetricWeakestPrecondition and MetricWeakestPreconditionProperties components. These files relate to the weakest-precondition semantics for the Bedrock2 language. The primary definition in `MetricWeakestPrecondition.v` is `cmd`, which can roughly be thought of as an explicit recursive version of the $\Downarrow$ big-step omnisemantics evaluation judgment. It takes the same parameters, but returns a concrete predicate saying whether the postcondition is satisfied. In some cases (function calls and loops), it simply calls the operational semantics directly, but it generally tries to unroll the generated condition as much as is practical. Several useful lemmas related to this weakest-precondition semantics are proved in `MetricWeakestPreconditionProperties.v`, such as the fact that it agrees with the operational semantics (`sound_cmd`). The program logic uses `cmd` directly to generate the verification condition for a program.

Like in the program-logic component, most of the changes in these two files are mechanical adjustments to existing code to account for extra metrics parameters. However, a few proofs also involved small but nontrivial logical modifications. The 70 deletions in `MetricWeakestPreconditionProperties.v` are two lemmas that were used in proofs of specific programs and happened to be in that file for incidental reasons, but were not needed for the metrics extension.

3.2.1.6 MetricLoops component. As the name suggests, `MetricLoops.v` contains a number of lemmas for dealing with loops when verifying programs. Once again, some of these changes were rote threading of metrics arguments, though made slightly more complex due to the loop machinery. However, since reasoning about loops is where the proof logic gets more complex, several statements needed meaningfully new proof code. Some lemmas, such as `atleastonce_localsmap`, had a bit of an unfortunate "backwards-compatibility" issue – whereas most lemma statements can be kept intact other than the addition of a metrics argument, some require restructuring some of the non-metrics arguments. Fortunately, this is always confined to reordering or rearranging the arguments somehow, so while adapting existing proof scripts is not quite as trivial as it could be, it is still very close.

I will also make the passing remark that the addition of metrics logic made some of

file	changes	adds	dels
MetricLogging		178	
MetricCosts		153	
MetricSemantics	118	149	90
MetricProgramLogic	33	10	14
MetricWeakestPrecondition	108	8	0
MetricWeakestPreconditionProperties	127	23	70
MetricLoops	244	26	0

Table 3.1: Overview of changes to Bedrock2 language components to add metrics-logging support

the proofs "more understandable," in a sense. Specifically, since metric costs are named for their functions (`cost_loop_true`, `cost_loop_false`, etc.), the addition of metrics in the appropriate places almost serves to annotate what is going on there.

3.2.2 Summary of effort

Table 3.1 summarizes the number of lines of code changed to implement metrics logging. The number of lines changed, added, and deleted sums to 1,351, of which about 25% were new definitions and 75% were integrations with the existing codebase. Given the level of complexity of the Bedrock2 compiler, and the mechanical nature of most of the integration work, I would subjectively judge this as a point in favor of the feasibility and practicality of an end-to-end foundational approach to verifying upper bounds on runtime, showing that our techniques are not just theoretically possible but also realistic.

Chapter 4

Verification of the compiler

After augmenting all the relevant Bedrock2 definitions with the formally specified metrics-logging machinery described in Chapter 3, the remaining task is to patch the correctness proof of the compiler, which must prove that postconditions – now containing metrics-log parameters – are still preserved.

4.1 Theory: Compiler pipeline correctness proof

The Bedrock2 compiler consists of several *compiler phases*, in each of which a "higher-level" input program p_H is compiled to a "lower-level" output program p_L. Without consideration of metrics, the statement of the correctness of a compiler stage is relatively simple:

$$(p_H/m/\ell/\tau \Downarrow Q) \to p_L/m/\ell/\tau \Downarrow Q$$

We can write the version extended to consider metrics in either the set-theoretic style:

$$\begin{gathered}(p_H/m/\ell/\tau/\mu_H \Downarrow Q) \to \\ p_L/m/\ell/\tau/\mu_L \Downarrow \{(m', \ell', \tau', \mu'_L) \mid (m', \ell', \tau', \mu'_H) \in Q,\ \mu'_L - \mu_L \leq \mu'_H - \mu_H\}\end{gathered}$$

Or the functional style, which as described in Section 1.2.1 is equivalent:

$$\begin{gathered}(p_H/m/\ell/\tau/\mu_H \Downarrow Q) \to \\ p_L/m/\ell/\tau/\mu_L \Downarrow \lambda m'.\lambda \ell'.\lambda \tau'.\lambda \mu'_L.\exists \mu'_H\, Q(m', \ell', \tau', \mu'_H) \wedge \mu'_L - \mu_L \leq \mu'_H - \mu_H\end{gathered} \tag{4.1}$$

This correctness statement looks much more complicated, but it is fairly intuitive. As a hypothesis, we have that the high-level program p_H with starting metrics μ_H lands within Q. Now we want to provide a set that we can prove the low-level program p_L with starting

metrics μ_L lands in. This is essentially the same set Q, except that if an element of Q has ending metrics μ'_H, we must include at least one ending configuration with the same ending state, and ending metrics log μ'_L such that the difference $\mu'_L - \mu_L$, i.e. the upper bound on how long the low-level program took, is at most the difference $\mu'_H - \mu_H$, i.e. the upper bound on how long the high-level program took.

Why is it important that we include *all* such ending configurations? In other words, why does Equation 4.1 have a $\leq$ symbol instead of a $=$ symbol? The answer is that some compiler stages will improve the metrics bounds according to the (possibly new) language semantics – for example, we might learn that a variable is stored in a register, or we might perform a compiler optimization. Instead of adding some kind of inelegant machinery to allow for manually weakening metrics bounds, we take the much cleaner approach of taking advantage of the "overapproximating" nature of omnisemantics wherever relevant. Indeed, note that the postcondition on the right-hand side contains clearly impossible ending configurations such as ones where the low-level program takes a negative amount of time. Since the postcondition is an overapproximation of possible configurations, the best bound we can get out of it is the worst one, which in fact is the one where equality holds in Equation 4.1.

We can prove that correctness of compiler phases is still preserved under composition: if we perform a compilation phase from p_H to p_M followed by p_M to p_L, the predicate Q becomes (under the functional viewpoint)

$$\begin{aligned}
&\exists \mu'_M. \, (\exists \mu'_H. \, Q(m', \ell', \tau', \mu'_H) \wedge \mu'_M - \mu_M \leq \mu'_H - \mu_H) \wedge \mu'_L - \mu_L \leq \mu'_M - \mu_M \\
\Leftrightarrow \; &\exists \mu'_H . \exists \mu'_M . \, Q(m', \ell', \tau', \mu'_H) \wedge \mu'_L - \mu_L \leq \mu'_M - \mu_M \leq \mu'_H - \mu_H \\
\Leftrightarrow \; &\exists \mu'_H . \, Q(m', \ell', \tau', \mu'_H) \wedge \mu'_L - \mu_L \leq \mu'_H - \mu_H,
\end{aligned}$$

where the equivalence holds in the forward direction because we can arbitrarily choose e.g. $\mu'_M = \mu'_L - \mu_L + \mu_M$ (and the reverse direction is clear).

To greatly simplify, the correctness theorem for each compiler phase is effectively an instance of Equation 4.1. The full compiler-correctness theorem is then effectively the composition of each of these theorems for compiler phases, as above.

Importantly, the evaluation judgment $\Downarrow$ is a parameter of the compiler stage and may have a different meaning on either side of the implication. In particular, some evaluation judgments for the same language (Bedrock2, FlatImp, or RISC-V) are defined with differing cost semantics. As described in Section 4.2.1.4. this variation allows us to account for compiler phases that increase time bounds, such as register spilling, without making it necessary to modify the postcondition. Accounting for compiler phases that decrease time bounds, like inlining and dead-code elimination, is automatic, since we only deal with upper bounds.

4.1.1 Design alternatives

Notably, the given version of the metrics statement is more general and nicer to work with than this possibly more intuitive and ostensibly simpler formulation.

$$(p_H/m/\ell/\tau/\mu \Downarrow Q) \to p_L/m/\ell/\tau/\mu \Downarrow \{(m', \ell', \tau', \mu'_L) \mid (m', \ell', \tau', \mu'_H) \in Q \wedge \mu'_L \leq \mu'_H\}$$

The critical difference is that our formulation allows the "starting metrics" of the high-level program and low-level program to differ. In particular, this choice has the technical advantage that we can prove such a statement directly by structural induction on the omni-big-step judgment in the hypothesis. Note that the shape of an induction proof on the "simpler" form gives inductive hypotheses with the same *starting* metrics as the original program, whereas to prove the compiler phase, we want the *ending* metrics in the inductive hypotheses to match the starting metrics of the original program.

4.2 Practice: Adaptation of Bedrock2 compiler phases

We again return to the Bedrock2 codebase to see how these theoretical ideas translate to concrete code. Much like in Section 3.2, I will review the details of the implementation in each compiler phase, roughly corresponding to source files.

4.2.1 Breakdown of changes

4.2.1.1 FlattenExpr phase. This phase performs the initial transformation of Bedrock2 code to FlatImp code, where the primary distinction is that nested expressions in Bedrock2 are flattened to separate imperative statements in FlatImp (hence the name). Since the transformation is quite simple, the adaptation of the proof was relatively straightforward and unnotable.

4.2.1.2 UseImmediate and DeadCodeElim phases. These phases are exclusively compiler optimizations, so no interesting innovations were needed. The only necessary changes were to thread metrics through the existing correctness proofs, verifying along the way that the metrics cannot inadvertently increase.

4.2.1.3 RegAlloc phase. The register-allocation phase is responsible for choosing which FlatImp variables are assigned to the 32 available RISC-V registers, and which ones will be stored on the stack instead. Specifically, the register allocator transforms FlatImp code with

string variables to FlatImp code with integer variables, where integers less than 32 correspond to registers, and integers greater than 32 correspond to virtual stack addresses. This phase does not affect the behavior of the code itself; the next phase, spilling (Section 4.2.1.4), will generate the code that handles stack variables.

This phase underwent the most significant changes during the metrics-logging augmentation. Recall from Section 3.1 that the best time bounds on many operations depend on whether their arguments are assigned to registers, which is not yet known at the Bedrock2 level. We therefore parameterize all language semantics over a boolean function `isReg`, used to indicate which variables should be considered "register variables." In the input to this phase, we still use `isRegStr`, which checks whether the variable name starts with `reg_`. Therefore, we had to modify the register allocator to prioritize assigning such variables to register locations. Of course, this is not always possible (since there are only 32 registers but arbitrarily many live string variables), so the register allocator fails if an input register variable ends up mapping to an output stack variable.

4.2.1.4 Spilling phase. The spilling phase is responsible for inserting the boilerplate code to perform loads and stores on stack variables, as well as on caller-saved registers around function calls. The input and output language semantics for this phase are the same, but we verify that after the spilling phase, *all* variables are registers (i.e. integers less than 32).

This phase also required some modifications to support metrics logging. For the boilerplate generated around stack variables, the cost incurred by added loads and stores is exactly accounted for by the difference between the costs of register and non-register variables in the definitions of the language semantics. (In fact, this is how those values were chosen.) For the boilerplate generated around function calls, we cannot employ a similar strategy, since the function call itself still looks the same. But recall again from Section 3.1 that we parameterize the cost semantics by another value `phase`, which goes from `PreSpill` in the input to this phase to `PostSpill` in the output. The `PreSpill` costs add a conservative upper bound to all function calls on the number of loads and stores required to save and restore all caller-saved registers.

Such augmentation is both necessary and convenient for our framework, in that it allows metrics bounds at the highest level of abstraction to be propagated unchanged down to the RISC-V level. Since the spilling phase adds boilerplate code around every function, the only way to account for the increase in upper bounds is to hardcode this difference into the cost semantics – but this behavior is actually an advantage, as it means the programmer can think entirely about high-level semantics and not what happens in the compiler internals; the framework handles these minutiae automatically.

compiler phase	code?	semantics?	source diff
FlattenExpr	no	yes	27
UseImmediate	no	no	125
DeadCodeElim	no	no	71
RegAlloc	yes	yes	90+115
Spilling	no	yes	403
FlatToRiscv	no	yes	133

Table 4.1: Overview of changes to compiler phases to add metrics-logging support

4.2.1.5 FlatToRiscv phase. Finally, this phase transforms FlatImp code with all variables in registers to RISC-V instructions. Much like the first phase, the modifications required in this phase were mundane and mostly amounted to verifying that the metrics semantics for FlatImp indeed match the metrics semantics for RISC-V.

4.2.2 Summary of effort

Table 4.1 summarizes the number of lines of code changed to integrate metrics logging with the compiler, as well as which compiler phases emit different code and which ones now have different semantics. As noted in Section 4.2.1.3, the only phase that needed implementation changes was RegAlloc, whose diff is split between changes to the unverified register allocator and the verified translation validator. The number of lines changed, added, and deleted sums to 964, which, as in Section 3.2.2, strikes me as an indication of the feasibility of our approach.

Chapter 5

Verification of source-level time bounds

Now equipped with a well-defined operational semantics incorporating metrics-logging together with a compiler verified with respect to those semantics, it is finally possible to write end-to-end timing proofs that, given a Bedrock2 source program, upper-bound the runtime metrics of the RISC-V output of the compiler on that program. In this chapter, I present a few miscellaneous pieces of commentary on the ergonomics and practical process of proving these bounds, as well as some case-study examples.

5.1 Compatibility with non-metrics proofs

One advantage of the way our metrics framework is structured is that source-program proofs have complete freedom as to the extent to employ it. Indeed, the metrics-instrumented proof framework can be used for correctness proofs that do not say anything about metrics at all. If the program specification does not mention anything about metrics, then no metrics proof obligations will be generated, and the proof will look identical to a proof written without the framework.

As a result, the process of turning an existing correctness proof into a proof-with-metrics is very streamlined. If the specification is left untouched, the conversion process is typically limited entirely to renaming autogenerated variable names (or perhaps not, if the original proof was robust enough) and occasionally moving some subgoal proofs around due to differing lemma structures.

Another side benefit of structuring the framework this way is that correctness and timing proofs can often work off of each other in tandem. As a simple example, proving that a loop terminates (which is already required for any correctness proof, since Bedrock2's Hoare logic

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

Figure 5.1: Exponentiation by squaring

enforces total correctness) and proving a time bound on a loop can reuse the same machinery: the same decreasing measure used for the termination proof can appear in the timing portion of the loop invariant, since they are highly conceptually similar (in both cases, we want an upper bound on how much more looping can occur).

Nevertheless, it is sometimes desirable for non-metrics proofs to coexist in the same codebase, so that metrics developments can live side-by-side with unrelated endeavors that do not need them. Fortunately, thanks to the careful separation of metrics in the Bedrock2 codebase described in Section 3.2, we retain access to the metrics-unaware version of Bedrock2 if desired.

5.2 Proving metric bounds

Adding metrics to a fully straightline program proof is nearly fully automated. The straightline tactic used for correctness proofs will automatically carry metrics through purely straightline code, thanks to the structure of the definition of the semantics as described in Section 3.1.3. The only additional task for the programmer is to discharge the final metrics goal, which is trivial via the automation described in Section 5.3.

Adding metrics to slightly more complicated programs is quite easy. For example, the sample program in Figure 5.1, which raises an integer `x` to the `e`-th power by repeated squaring, can be shown to have a time bound of some constant plus another constant times the most significant bit of the exponent. The difference between the correctness proof and the correctness-plus-metrics proof is only switching the framework from the non-metrics version to the metrics version, adding the metrics bound in the specification, and 13 real lines of proof code (plus one arithmetic lemma about most significant bit).

By way of example, we show two modifications of specifications to existing programs that add time bounds. First we again look at integer power, whose specifications without and with time bounds are given in Figure 5.2. This program is simple enough that we prove the

```
(* spec without metrics *)

#[export] Instance spec_of_ipow : spec_of "ipow" :=
  fnspec! "ipow" x e ~> v,
  { requires t m mc := True;
    ensures t' m' mc' := unsigned v = unsigned x ^ unsigned e mod 2^64
  }.


(* definitions for metrics *)

Definition initCost := {| instructions := 12; stores := 2; loads := 13; jumps := 0 |}.
Definition iterCost := {| instructions := 76; stores := 16; loads := 98; jumps := 2 |}.
Definition endCost :=  {| instructions := 6; stores := 1; loads := 9; jumps := 1 |}.

Definition msb z := match z with
                    | Zpos _ ⇒ Z.log2 z + 1
                    | _ ⇒ 0
                    end.

(* spec with metrics *)

#[export] Instance spec_of_ipow : spec_of "ipow" :=
  fnspec! "ipow" x e ~> v,
  { requires t m mc := True;
    ensures t' m' mc' := unsigned v = unsigned x ^ unsigned e mod 2^64 ∧
      (mc' - mc <= initCost + (msb (word.unsigned e)) * iterCost + endCost)%metricsH
  }.
```

Figure 5.2: Comparison of the specification of exponentiation by squaring with and without metrics

tightest possible bounds that can be proved in the framework. The time bound is given in terms of the most significant bit of the input, which is one off from its base-2 logarithm. The specification itself simply adds a clause to the postcondition, describing an upper bound on how much the metrics log could have changed.

The second example comes from the much larger-scale proof of concept described in slightly greater detail in Section 5.4. The specific example in Figure 5.3 comes from a function in the lowest-level driver. It performs a basic write operation by first busylooping until an I/O device is ready. Each query to the devices leaves an entry in the I/O trace; therefore, the specification can be written in terms of the length of the trace to account for the amount of time spent looping. In particular, our metrics specification here is a constant bound for the non-loop portion of the function plus a multiplicative bound with respect to the length of the (high-level abstracted) trace.

```
(* function definition *)

Definition spi_write := func! (b) ~> busy {
    busy = $-1;
    i = $patience; while i { i = i - $1;
      io! busy = MMIOREAD($0x10024048);
      if !(busy >> $31) { i = i^i }
    };
    if !(busy >> $31) {
      output! MMIOWRITE($0x10024048, b);
      busy = (busy ^ busy)
    }
  }.


(* spec without metrics *)

Global Instance spec_of_spi_write : spec_of "spi_write" := fun functions ⇒ forall t m b mc,
  word.unsigned b < 2 ^ 8 →
  MetricWeakestPrecondition.call functions "spi_write" t m [b] mc (fun T M RETS MC ⇒
    M = m ∧ exists iol, T = t ;++ iol
    ∧ exists ioh, mmio_trace_abstraction_relation ioh iol
    ∧ exists err, RETS = [err] ∧ Logic.or
      (((word.unsigned err <> 0) ∧ lightbulb_spec.spi_write_full _ ^* ioh
        ∧ Z.of_nat (length ioh) = patience))
      (word.unsigned err = 0
        ∧ lightbulb_spec.spi_write word (byte.of_Z (word.unsigned b)) ioh)).


(* definitions for metrics *)

Definition mc_spi_write_const := mkMetricLog 348 227 381 204.
Definition mc_spi_mul := mkMetricLog 157 109 169 102.

(* spec with metrics *)

Global Instance spec_of_spi_write : spec_of "spi_write" := fun functions ⇒ forall t m b mc,
  word.unsigned b < 2 ^ 8 →
  MetricWeakestPrecondition.call functions "spi_write" t m [b] mc (fun T M RETS MC ⇒
    M = m ∧ exists iol, T = t ;++ iol
    ∧ exists ioh, mmio_trace_abstraction_relation ioh iol
    ∧ exists err, RETS = [err] ∧ Logic.or
      (((word.unsigned err <> 0) ∧ lightbulb_spec.spi_write_full _ ^* ioh
        ∧ Z.of_nat (length ioh) = patience))
      (word.unsigned err = 0
        ∧ lightbulb_spec.spi_write word (byte.of_Z (word.unsigned b)) ioh)
        ∧ (MC - mc <= mc_spi_write_const + Z.of_nat (length ioh) * mc_spi_mul)%metricsH).
```

Figure 5.3: SPI write example

5.3 Proof automation

I have developed a moderately large array of tactics to streamline the process of discharging metrics goals as much as possible. I briefly summarize them here.

At the lowest level, the custom tactics that operate directly on metric logs – namely, `unfold_MetricLog` and `simpl_MetricLog` – culminate in `solve_MetricLog`, where after the metric-log objects are sufficiently unpacked, we simply apply the linear-arithmetic solver `blia` to solve the resulting systems of linear inequalities automatically. Not only is the programmer freed from spelling out arguments that are not illuminating, but an amusing method of confirming tightness of bounds is enabled: keep tweaking the specification with tighter and tighter bounds until automated proof no longer succeeds. (The failure does manifest in a way that is helpful for understanding *why* it occurs, focusing in on one formula of linear arithmetic that could not be proved.)

Over the course of a program proof, a metrics object will often accumulate a long chain of addition operations, since each straightline command adds on to the previous metrics. Crucially, these are highly linear and nonbranching chains, as explained in Section 3.1.3. We can therefore employ a tactic `flatten_MetricLog`, which turns these long chains of additions into a single one, combining all constants. This measure is one of several to boost practical performance, making the linear-arithmetic-solver strategy viable for real and complicated programs like the example in Section 5.4.

At a higher level, we also have several tactics that deal with the cost functions denoted $C\_$ in this paper. Notably, these functions are often deeply nested as a result of the aforementioned long chains of additions, and internal details related to the Coq kernel cause severe performance issues at `Qed`-time when they are unfolded naively. We therefore provide a carefully crafted `cost_unfold` tactic that performs these unfolds in a manner that placates the kernel, which I will shortly describe in more detail in Section 5.3.1.

Finally, we provide some convenience tactics `cost_solve` and `cost_hammer`, which perform necessary unfolding and syntactic manipulation of these cost functions and then pass the goals to several of the lower-level metric-log-solving tactics. Since we provide a wide array of tactics of varying strength, we are able to choose the appropriate performance/convenience tradeoff for any given metrics subgoal.

5.3.1 Aside: Coq internals

In the initial stages of work on verifying concrete programs, I implemented the `cost_unfold` tactic in Figure 5.4. On the surface, this tactic appears to serve mostly for user convenience

```
Ltac cost_unfold :=
  unfold cost_interact, cost_call, cost_load, cost_store,
  cost_inlinetable, cost_stackalloc, cost_lit, cost_op, cost_set,
  cost_if, cost_loop_true, cost_loop_false, EmptyMetricLog in *.
```

Figure 5.4: Naive unfolding leading to `Qed` runtime blowup

and should have little influence on the internal workings of the proof engine: it simply replaces the `cost_*` functions with their expanded definitions, so that the user can manipulate them further.

However, as I began to verify more complicated programs, I noticed that while interactively executing a proof would proceed smoothly, the time it took for Coq to process the final `Qed` command to complete the proof grew increasingly long, eventually to the point where it was not practical to wait for it to finish. My initial suspicion was that the automated linear-arithmetic tactics were the culprit, but by binary-searching on the proof script (by picking points to stop, forcibly resolving all remaining subgoals by an added axiom with type `forall {T}, T`, and trying `Qed` then), and then manually unrolling layers of nested tactics, I eventually discovered that `cost_unfold` was to blame.

This in itself is not too surprising – it is known that simplification-style tactics such as `simpl`, `cbv`, and `unfold` can have this behavior. But, at least to me, it is very difficult to predict when this will happen, and debugging such issues can feel like blindly negotiating with a fickle spirit until you happen upon the particular magical incantation it wants.

As a case in point, replacing `unfold ... in H` with `cbv [...] in H` still exhibits the same behavior. Definitionally, the following code is equivalent, but instead of hanging on `Qed` it hangs upon applying the tactic instead.

```
let t := type of H in
let t' := eval cbv [...] in t in
replace t with t' in H by reflexivity.
```

And even more definitionally, the following code is equivalent to the above but causes everything to go through quickly.

```
let t := type of H in
let t' := eval cbv [...] in t in
replace t with t' in H by (symmetry; reflexivity)
```

The only difference is `by (symmetry; reflexivity)` instead of `by reflexivity`. Proving that the expression is equal to the simplified version hangs, but proving that the simplified version is equal to the expression is fine.

By some way of explanation, simplification-style tactics are dangerous in terms of `Qed`-

```
(* awkward tactic use to avoid Qed slowness *)
(* this is slow with (eq_refl t) and fast with (eq_refl t') due to black box heuristics *)
Ltac cost_unfold :=
  repeat (
    let H := match goal with
             | H : context[cost_interact] ⊢ _ ⇒ H
             | H : context[cost_call] ⊢ _ ⇒ H
             | H : context[cost_load] ⊢ _ ⇒ H
             | H : context[cost_store] ⊢ _ ⇒ H
             | H : context[cost_inlinetable] ⊢ _ ⇒ H
             | H : context[cost_stackalloc] ⊢ _ ⇒ H
             | H : context[cost_lit] ⊢ _ ⇒ H
             | H : context[cost_op] ⊢ _ ⇒ H
             | H : context[cost_set] ⊢ _ ⇒ H
             | H : context[cost_if] ⊢ _ ⇒ H
             | H : context[cost_loop_true] ⊢ _ ⇒ H
             | H : context[cost_loop_false] ⊢ _ ⇒ H
             end in
    let t := type of H in
    let t' := eval cbv [cost_interact cost_call cost_load cost_store
      cost_inlinetable cost_stackalloc cost_lit cost_op cost_set
      cost_if cost_loop_true cost_loop_false] in t in
    replace t with t' in H by (exact (eq_refl t'))
  );
  cbv [cost_interact cost_call cost_load cost_store cost_inlinetable
  cost_stackalloc cost_lit cost_op cost_set cost_if cost_loop_true
  cost_loop_false];
  unfold EmptyMetricLog in *.
```

Figure 5.5: Workaround for unfolding leading to fast `Qed` runtime

slowness because they require the kernel to check that the expression before simplification is equal to the expression after simplification at the end of the proof. The conversion algorithms for doing so can take arbitrarily long (as this problem is undecidable in general), and it has heuristics that are sensitive to properties such as the order of terms (`A = B` vs. `B = A`). In this case, it seems that the way to get the kernel to figure out how to perform this conversion quickly is to give it explicitly, and also flipped.

Figure 5.5 displays the full tactic definition of the performant `cost_unfold`. Note that `by (exact (eq_refl t'))` is another way to write `by (symmetry; reflexivity)`, where `by (exact (eq_refl t))` would analogously correspond to `by reflexivity` (and exhibits the same slowness).

While this saga is wholly uninteresting to anyone who just wants to prove programs correct, I think it is a fairly instructive example for how one might go about investigating similar problems; it is also a possible area of improvement for Coq usability (although, again, deciding equality of arbitrary terms is undecidable in general, so the "problem" can never

fully be “solved”).

5.4 Lightbulb IoT microcontroller

One of the capstone proof-of-concept examples for the original Bedrock2 verified toolchain was a network-connected microcontroller for controlling a lightbulb, intended as a simple example of a generic physical control system [Erbsen et al. 2021]. Appropriately, we use this same example to demonstrate the viability of our augmented toolchain.

I will not go into too much detail here, as I did only the low-level implementation work for the network subcomponents before handing the task of completing the full lightbulb proof off to Erbsen, but the derived time bounds place an explicit upper limit of about 10 million instructions required to handle each command. Even though our current framework takes no advantage of compiler optimizations and largely adds up worst-case costs without regard for context, this bound is still tight enough to imply a latency of about 40ms (based on reasonable assumptions about the hardware), faster than the mechanical response times of the physical system itself.

Figure 5.6 shows the top-level correctness statement for a single iteration of the event-handler loop of the lightbulb program. Crucially, the expression for the metrics bound refers to both the length of the input packet as well as the length of the I/O trace involved in reading and responding to the packet. This demonstrates the capability of our framework to express time bounds in maximum generality in terms of both input and I/O interactions, as claimed.

The `metric_lightbulb_correct` statement references the `handle_request_spec` definition, which in turn refers to `loop_cost`. The `loop_cost` function expresses the upper bound on time taken by one loop iteration in terms of a number of constants, such as `mc_spi_xchg_const`, which is multiplied by the packet length to account for the low-level SPI reads and writes involved in receiving it; plus `mc_spi_mul`, which is multiplied by the length of the I/O trace to account for the time spent polling for the next network byte; plus some additive constants, for the logic of the processing itself. In `handle_request_spec` (the metrics-aware version of a similar definition under the name `loop_progress` in the original lightbulb proof), we look at the I/O trace, which governs what happened, and give appropriate bounds using `loop_cost` in each case.

```
Definition loop_cost(packet_length trace_length: Z): RiscvMetrics :=
  (60 + 7 * packet_length) * mc_spi_xchg_const +
  lightbulb_handle_cost + trace_length * mc_spi_mul +
  loop_compilation_overhead.

Definition handle_request_spec(t t': trace)(mc mc': RiscvMetrics) :=
  exists dt, t' = dt ++ t ∧
  exists ioh, metric_SPI.mmio_trace_abstraction_relation ioh dt ∧ (
    (* Case 1: Received packet with valid command: *)
    (exists packet cmd,
        (lan9250_recv packet +++ gpio_set 23 cmd) ioh ∧
        lightbulb_packet_rep cmd packet ∧
        (mc' - mc <= loop_cost (length packet) (length ioh))) ∨
    (* Case 2: Received invalid packet: *)
    (exists packet,
        (lan9250_recv packet) ioh ∧
        not (exists cmd, lightbulb_packet_rep cmd packet) ∧
        (mc' - mc <= loop_cost (length packet) (length ioh))) ∨
    (* Case 3: Polled, but no new packet was available: *)
    (lan9250_recv_no_packet ioh ∧
        (mc' - mc <= loop_cost 0 (length ioh))) ∨
    (* Case 4: Received too long packet *)
    (lan9250_recv_packet_too_long ioh) ∨
    (* Case 5: SPI protocol timeout *)
    ((TracePredicate.any +++ spi_timeout) ioh)).

Theorem metric_lightbulb_correct: forall (initial : MetricRiscvMachine) R,
    valid_machine initial →
    getLog initial = [] →
    regs_initialized.regs_initialized (getRegs initial) →
    getNextPc initial = word.add (getPc initial) (word.of_Z 4) →
    getPc initial = code_start ml →
    (program RV32IM (code_start ml) (fst (fst out)) * R *
       LowerPipeline.mem_available (heap_start ml) (heap_pastend ml) *
       LowerPipeline.mem_available (stack_start ml) (stack_pastend ml))%sep
      (getMem initial) →
    subset (footpr (program RV32IM (code_start ml) (fst (fst out))))
      (of_list (getXAddrs initial)) →
    eventually riscv.run1
      (successively riscv.run1
         (fun s s' : MetricRiscvMachine ⇒
            handle_request_spec (getLog s) (getLog s')
              (getMetrics s) (getMetrics s'))) initial.
```

Figure 5.6: End-to-end theorem and its two most important supporting definitions

Chapter 6

Future work

Although this framework for time bounds on interactive programs is already usable in practice, it still has many exciting directions for further research.

The most immediate direction to take is to improve upon the bounds given by the framework itself. In particular, there are at present several places where our framework makes more conservative estimates than it could. For instance, the buffer added to function calls when performing spilling is currently a static constant amount, whereas the framework has access to several pieces of information (function arity, variables in registers) that could give tighter bounds. Furthermore, the bounds given when evaluating complex expressions assume that all intermediate results are stored in non-register variables, which is in fact unlikely in practice. Both of these cases are examples of places where the framework, although theoretically sound, could see practical improvement.

An orthogonal direction that could also be interesting is to allow the framework also to prove lower bounds on programs, as alluded to in Chapter 3. While the framework is already fully general in the domain of program and compiler-stage specifications, the restriction to upper bounds is baked into the evaluation and program semantics. In principle, it would be possible to provide both upper and lower bounds in these places. However, more work would be necessary to make actual use of the resulting lower bounds, since the compiler has several optimization stages, and exactly what effects they have on lower bounds would be challenging to determine and prove rigorously. Nevertheless, a verified optimizing compiler with bounds on both sides would also be a novel development in this area for the future.

Appendix A

Bedrock2 semantics definitions

What follows is the Coq definition of the Bedrock2 semantics, both without and with metrics. I have omitted some irrelevant snippets of code, marked with the comment `(* [...] *)`, and made a few other minor edits for legibility.

A.1 Baseline semantics

```
(* BW is not needed on the rhs, but helps infer width *)
Definition LogItem{width: Z}{BW: Bitwidth width}
                  {word: word.word width}{mem: map.map word byte} :=
  ((mem * String.string * list word) * (mem * list word))%type.

Definition trace{width: Z}{BW: Bitwidth width}
                {word: word.word width}{mem: map.map word byte} :=
  list LogItem.

Definition ExtSpec{width: Z}{BW: Bitwidth width}
                  {word: word.word width}{mem: map.map word byte} :=
  (* Given a trace of what happened so far,
     the given-away memory, an action label and a list of function call arguments, *)
  trace → mem → String.string → list word →
  (* and a postcondition on the received memory and function call results, *)
  (mem → list word → Prop) →
  (* tells if this postcondition will hold *)
  Prop.

(* [...] *)

Section binops.
```

```
  Context {width : Z} {word : Word.Interface.word width}.
  Definition interp_binop (bop : bopname) : word → word → word :=
    match bop with
    | bopname.add ⇒ word.add
    | bopname.sub ⇒ word.sub
    | bopname.mul ⇒ word.mul
    | bopname.mulhuu ⇒ word.mulhuu
    | bopname.divu ⇒ word.divu
    | bopname.remu ⇒ word.modu
    | bopname.and ⇒ word.and
    | bopname.or ⇒ word.or
    | bopname.xor ⇒ word.xor
    | bopname.sru ⇒ word.sru
    | bopname.slu ⇒ word.slu
    | bopname.srs ⇒ word.srs
    | bopname.lts ⇒ fun a b ⇒
      if word.lts a b then word.of_Z 1 else word.of_Z 0
    | bopname.ltu ⇒ fun a b ⇒
      if word.ltu a b then word.of_Z 1 else word.of_Z 0
    | bopname.eq ⇒ fun a b ⇒
      if word.eqb a b then word.of_Z 1 else word.of_Z 0
    end.
End binops.

Definition env: map.map String.string Syntax.func := SortedListString.map _.
#[export] Instance env_ok: map.ok env := SortedListString.ok _.

Section semantics.
  Context {width: Z} {BW: Bitwidth width} {word: word.word width} {mem: map.map word byte}.
  Context {locals: map.map String.string word}.
  Context {ext_spec: ExtSpec}.

  Section WithMemAndLocals.
    Context (m : mem) (l : locals).

    Local Notation "x ← a ; f" := (match a with Some x ⇒ f | None ⇒ None end)
      (right associativity, at level 70).

    Fixpoint eval_expr (e : expr) : option word :=
      match e with
      | expr.literal v ⇒ Some (word.of_Z v)
      | expr.var x ⇒ map.get l x
      | expr.inlinetable aSize t index ⇒
```

```
          index' ← eval_expr index;
          load aSize (map.of_list_word t) index'
      | expr.load aSize a ⇒
          a' ← eval_expr a;
          load aSize m a'
      | expr.op op e1 e2 ⇒
          v1 ← eval_expr e1;
          v2 ← eval_expr e2;
          Some (interp_binop op v1 v2)
      | expr.ite c e1 e2 ⇒
          vc ← eval_expr c;
          eval_expr (if word.eqb vc (word.of_Z 0) then e2 else e1)
      end.

    Fixpoint eval_call_args (arges : list expr) :=
      match arges with
      | e :: tl ⇒
        v ← eval_expr e;
        args ← eval_call_args tl;
        Some (v :: args)
      | _ ⇒ Some nil
      end.

  End WithMemAndLocals.
End semantics.

Module exec. Section WithParams.
  Context {width: Z} {BW: Bitwidth width} {word: word.word width} {mem: map.map word byte}.
  Context {locals: map.map String.string word}.
  Context {ext_spec: ExtSpec}.
  Section WithEnv.
  Context (e: env).

  Implicit Types post : trace → mem → locals → Prop.
  Inductive exec: cmd → trace → mem → locals →
                  (trace → mem → locals → Prop) → Prop :=
  | skip: forall t m l post,
      post t m l →
      exec cmd.skip t m l post
  | set: forall x e t m l post v,
      eval_expr m l e = Some v →
      post t m (map.put l x v) →
      exec (cmd.set x e) t m l post
```

```
| unset: forall x t m l post,
    post t m (map.remove l x) →
    exec (cmd.unset x) t m l post
| store: forall sz ea ev t m l post a v m',
    eval_expr m l ea = Some a →
    eval_expr m l ev = Some v →
    store sz m a v = Some m' →
    post t m' l →
    exec (cmd.store sz ea ev) t m l post
| stackalloc: forall x n body t mSmall l post,
    Z.modulo n (bytes_per_word width) = 0 →
    (forall a mStack mCombined,
      anybytes a n mStack →
      map.split mCombined mSmall mStack →
      exec body t mCombined (map.put l x a)
        (fun t' mCombined' l' ⇒
          exists mSmall' mStack',
            anybytes a n mStack' ∧
            map.split mCombined' mSmall' mStack' ∧
            post t' mSmall' l')) →
    exec (cmd.stackalloc x n body) t mSmall l post
| if_true: forall t m l e c1 c2 post v,
    eval_expr m l e = Some v →
    word.unsigned v <> 0 →
    exec c1 t m l post →
    exec (cmd.cond e c1 c2) t m l post
| if_false: forall e c1 c2 t m l post v,
    eval_expr m l e = Some v →
    word.unsigned v = 0 →
    exec c2 t m l post →
    exec (cmd.cond e c1 c2) t m l post
| seq: forall c1 c2 t m l post mid,
    exec c1 t m l mid →
    (forall t' m' l', mid t' m' l' → exec c2 t' m' l' post) →
    exec (cmd.seq c1 c2) t m l post
| while_false: forall e c t m l post v,
    eval_expr m l e = Some v →
    word.unsigned v = 0 →
    post t m l →
    exec (cmd.while e c) t m l post
| while_true: forall e c t m l post v mid,
    eval_expr m l e = Some v →
    word.unsigned v <> 0 →
```

```
      exec c t m l mid →
      (forall t' m' l', mid t' m' l' → exec (cmd.while e c) t' m' l' post) →
      exec (cmd.while e c) t m l post
  | call: forall binds fname arges t m l post params rets fbody args lf mid,
      map.get e fname = Some (params, rets, fbody) →
      eval_call_args m l arges = Some args →
      map.of_list_zip params args = Some lf →
      exec fbody t m lf mid →
      (forall t' m' st1, mid t' m' st1 →
          exists retvs, map.getmany_of_list st1 rets = Some retvs ∧
          exists l', map.putmany_of_list_zip binds retvs l = Some l' ∧
          post t' m' l') →
      exec (cmd.call binds fname arges) t m l post
  | interact: forall binds action arges args t m l post mKeep mGive mid,
      map.split m mKeep mGive →
      eval_call_args m l arges = Some args →
      ext_spec t mGive action args mid →
      (forall mReceive resvals, mid mReceive resvals →
          exists l', map.putmany_of_list_zip binds resvals l = Some l' ∧
          forall m', map.split m' mKeep mReceive →
          post (cons ((mGive, action, args), (mReceive, resvals)) t) m' l') →
      exec (cmd.interact binds action arges) t m l post.

  (* [...] *)

  End WithEnv.

  (* [...] *)

  End WithParams.
End exec. Notation exec := exec.exec.
```

A.2 Metrics-aware semantics

```
Local Notation UNK := String.EmptyString.

Section semantics.
  Context {width: Z} {BW: Bitwidth width} {word: word.word width} {mem: map.map word byte}.
  Context {locals: map.map String.string word}.
  Context {ext_spec: ExtSpec}.

  Local Notation metrics := MetricLog.
```

```
Section WithMemAndLocals.
  Context (m : mem) (l : locals).

  Local Notation "' x ← a | y ; f" := (match a with x ⇒ f | _ ⇒ y end)
    (right associativity, at level 70, x pattern).

  (* TODO possibly be a bit smarter about whether things are registers,
     for tighter metrics bounds at bedrock2 level *)
  Fixpoint eval_expr (e : expr) (mc : metrics) : option (word * metrics) :=
    match e with
    | expr.literal v ⇒ Some (word.of_Z v, cost_lit isRegStr UNK mc)
    | expr.var x ⇒ match map.get l x with
                   | Some v ⇒ Some (v, cost_set isRegStr UNK x mc)
                   | None ⇒ None
                   end
    | expr.inlinetable aSize t index ⇒
        'Some (index', mc') ← eval_expr index mc | None;
        'Some v ← load aSize (map.of_list_word t) index' | None;
        Some (v, cost_inlinetable isRegStr UNK UNK mc')
    | expr.load aSize a ⇒
        'Some (a', mc') ← eval_expr a mc | None;
        'Some v ← load aSize m a' | None;
        Some (v, cost_load isRegStr UNK UNK mc')
    | expr.op op e1 e2 ⇒
        'Some (v1, mc') ← eval_expr e1 mc | None;
        'Some (v2, mc'') ← eval_expr e2 mc' | None;
        Some (interp_binop op v1 v2, cost_op isRegStr UNK UNK UNK mc'')
    | expr.ite c e1 e2 ⇒
        'Some (vc, mc') ← eval_expr c mc | None;
        eval_expr (if word.eqb vc (word.of_Z 0) then e2 else e1)
                  (cost_if isRegStr UNK (Some UNK) mc')
    end.

  Fixpoint eval_call_args (arges : list expr) (mc : metrics) :=
    match arges with
    | e :: tl ⇒
      'Some (v, mc') ← eval_expr e mc | None;
      'Some (args, mc'') ← eval_call_args tl mc' | None;
      Some (v :: args, mc'')
    | _ ⇒ Some (nil, mc)
    end.
```

```
  End WithMemAndLocals.
End semantics.

Module exec. Section WithParams.
  Context {width: Z} {BW: Bitwidth width} {word: word.word width} {mem: map.map word byte}.
  Context {locals: map.map String.string word}.
  Context {ext_spec: ExtSpec}.
  Section WithEnv.
  Context (e: env).

  Local Notation metrics := MetricLog.

  Implicit Types post : trace → mem → locals → metrics → Prop.
  Inductive exec :
    cmd → trace → mem → locals → metrics →
    (trace → mem → locals → metrics → Prop) → Prop :=
  | skip
    t m l mc post
    (_ : post t m l mc)
    : exec cmd.skip t m l mc post
  | set x e
    t m l mc post
    v mc' (_ : eval_expr m l e mc = Some (v, mc'))
    (_ : post t m (map.put l x v) (cost_set isRegStr x UNK mc'))
    : exec (cmd.set x e) t m l mc post
  | unset x
    t m l mc post
    (_ : post t m (map.remove l x) mc)
    : exec (cmd.unset x) t m l mc post
  | store sz ea ev
    t m l mc post
    a mc' (_ : eval_expr m l ea mc = Some (a, mc'))
    v mc'' (_ : eval_expr m l ev mc' = Some (v, mc''))
    m' (_ : store sz m a v = Some m')
    (_ : post t m' l (cost_store isRegStr UNK UNK mc''))
    : exec (cmd.store sz ea ev) t m l mc post
  | stackalloc x n body
    t mSmall l mc post
    (_ : Z.modulo n (bytes_per_word width) = 0)
    (_ : forall a mStack mCombined,
        anybytes a n mStack →
        map.split mCombined mSmall mStack →
        exec body t mCombined (map.put l x a) (cost_stackalloc isRegStr x mc)
```

```
        (fun t' mCombined' l' mc' ⇒
          exists mSmall' mStack',
            anybytes a n mStack' ∧
            map.split mCombined' mSmall' mStack' ∧
            post t' mSmall' l' mc'))
   : exec (cmd.stackalloc x n body) t mSmall l mc post
| if_true t m l mc e c1 c2 post
  v mc' (_ : eval_expr m l e mc = Some (v, mc'))
  (_ : word.unsigned v <> 0)
  (_ : exec c1 t m l (cost_if isRegStr UNK (Some UNK) mc') post)
  : exec (cmd.cond e c1 c2) t m l mc post
| if_false e c1 c2
  t m l mc post
  v mc' (_ : eval_expr m l e mc = Some (v, mc'))
  (_ : word.unsigned v = 0)
  (_ : exec c2 t m l (cost_if isRegStr UNK (Some UNK) mc') post)
  : exec (cmd.cond e c1 c2) t m l mc post
| seq c1 c2
  t m l mc post
  mid (_ : exec c1 t m l mc mid)
  (_ : forall t' m' l' mc', mid t' m' l' mc' → exec c2 t' m' l' mc' post)
  : exec (cmd.seq c1 c2) t m l mc post
| while_false e c
  t m l mc post
  v mc' (_ : eval_expr m l e mc = Some (v, mc'))
  (_ : word.unsigned v = 0)
  (_ : post t m l (cost_loop_false isRegStr UNK (Some UNK) mc'))
  : exec (cmd.while e c) t m l mc post
| while_true e c
    t m l mc post
    v mc' (_ : eval_expr m l e mc = Some (v, mc'))
    (_ : word.unsigned v <> 0)
    mid (_ : exec c t m l mc' mid)
    (_ : forall t' m' l' mc'', mid t' m' l' mc'' →
         exec (cmd.while e c) t' m' l' (cost_loop_true isRegStr UNK (Some UNK) mc'') post)
  : exec (cmd.while e c) t m l mc post
| call binds fname arges
    t m l mc post
    params rets fbody (_ : map.get e fname = Some (params, rets, fbody))
    args mc' (_ : eval_call_args m l arges mc = Some (args, mc'))
    lf (_ : map.of_list_zip params args = Some lf)
    mid (_ : exec fbody t m lf mc' mid)
    (_ : forall t' m' st1 mc'', mid t' m' st1 mc'' →
```

```
          exists retvs, map.getmany_of_list st1 rets = Some retvs ∧
          exists l', map.putmany_of_list_zip binds retvs l = Some l' ∧
          post t' m' l'  (cost_call PreSpill mc''))
    : exec (cmd.call binds fname arges) t m l mc post
  | interact binds action arges
      t m l mc post
      mKeep mGive (_: map.split m mKeep mGive)
      args mc' (_ :  eval_call_args m l arges mc = Some (args, mc'))
      mid (_ : ext_spec t mGive action args mid)
      (_ : forall mReceive resvals, mid mReceive resvals →
          exists l', map.putmany_of_list_zip binds resvals l = Some l' ∧
          forall m', map.split m' mKeep mReceive →
          post (cons ((mGive, action, args), (mReceive, resvals)) t) m' l'
            (cost_interact PreSpill mc'))
    : exec (cmd.interact binds action arges) t m l mc post.

  (* [...] *)

  End WithEnv.

  (* [...] *)

  End WithParams.
End exec. Notation exec := exec.exec.
```

References

Amadio, Roberto M. et al. (2014). "Certified Complexity (CerCo)". In: *Foundational and Practical Aspects of Resource Analysis*. Ed. by Ugo Dal Lago and Ricardo Peña. Vol. 8552. Cham: Springer International Publishing, pp. 1–18. ISBN: 978-3-319-12465-0 978-3-319-12466-7. DOI: 10.1007/978-3-319-12466-7_1. URL: http://link.springer.com/10.1007/978-3-319-12466-7_1 (visited on 05/12/2022).

Appel, Andrew W. (2001). "Foundational Proof-Carrying Code". In: *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*. LICS '01. USA: IEEE Computer Society, p. 247.

Ayache, Nicolas, Roberto M. Amadio, and Yann Régis-Gianas (2012). "Certifying and Reasoning on Cost Annotations in C Programs". In: *Formal Methods for Industrial Critical Systems*. Ed. by David Hutchison et al. Vol. 7437. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 32–46. ISBN: 978-3-642-32468-0 978-3-642-32469-7. DOI: 10.1007/978-3-642-32469-7_3. URL: http://link.springer.com/10.1007/978-3-642-32469-7_3 (visited on 05/12/2022).

Bonenfant, Armelle, Christian Ferdinand, Kevin Hammond, and Reinhold Heckmann (2007). "Worst-Case Execution Times for a Purely Functional Language". In: *Implementation and Application of Functional Languages*. Ed. by Zoltán Horváth, Viktória Zsók, and Andrew Butterfield. Berlin, Heidelberg: Springer, pp. 235–252. ISBN: 978-3-540-74130-5. DOI: 10.1007/978-3-540-74130-5_14.

Charguéraud, Arthur, Adam Chlipala, Andres Erbsen, and Samuel Gruetter (Mar. 2023). "Omnisemantics: Smooth Handling of Nondeterminism". In: *ACM Transactions on Programming Languages and Systems* 45.1, 5:1–5:43. ISSN: 0164-0925. DOI: 10.1145/3579834. URL: https://dl.acm.org/doi/10.1145/3579834 (visited on 08/15/2023).

Crary, Karl and Stephnie Weirich (2000). "Resource Bound Certification". In: *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '00*. Boston, MA, USA: ACM Press, pp. 184–198. ISBN: 978-1-58113-125-3. DOI: 10.1145/325694.325716. URL: http://portal.acm.org/citation.cfm?doid=325694.325716 (visited on 05/13/2022).

Cuoq, Pascal, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski (2012). "Frama-C". In: *Software Engineering and Formal Methods*. Ed. by George Eleftherakis, Mike Hinchey, and Mike Holcombe. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 233–247. ISBN: 978-3-642-33826-7.

Erbsen, Andres, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala (2021). "Integration Verification across Software and Hardware for a Simple Embedded System". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Association for Computing Machinery, pp. 604–619. ISBN: 9781450383912. DOI: 10.1145/3453483.3454065. URL: https://doi.org/10.1145/3453483.3454065.

Haslbeck, Maximilian P. L. and Peter Lammich (July 2022). "For a Few Dollars More: Verified Fine-Grained Algorithm Analysis Down to LLVM". In: *ACM Transactions on Programming Languages and Systems* 44.3, 14:1–14:36. ISSN: 0164-0925. DOI: 10.1145/3486169. URL: https://doi.org/10.1145/3486169 (visited on 12/04/2023).

Maroneze, André, Sandrine Blazy, David Pichardie, and Isabelle Puaut (2014). "A Formally Verified WCET Estimation Tool". In: *14th International Workshop on Worst-Case Execution Time Analysis (2014)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 11–20. DOI: 10.4230/OASIcs.WCET.2014.11. URL: https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.WCET.2014.11 (visited on 11/05/2024).

Necula, George C. and Peter Lee (1998). "Safe, Untrusted Agents Using Proof-Carrying Code". In: *Mobile Agents and Security*. Ed. by Giovanni Vigna. Berlin, Heidelberg: Springer, pp. 61–91. ISBN: 978-3-540-68671-2. DOI: 10.1007/3-540-68671-1_5. URL: https://doi.org/10.1007/3-540-68671-1_5 (visited on 06/10/2024).