A hybrid formal verification method for novel architectures

Trinity Term 2023

Hou Chua

Honour School of Computer Science (Part C)

Word count: 8,075 words (computed with TeXcount)

Abstract

There are a myriad of fascinating and extremely valuable ongoing developments in formally verified systems, such as the CHERI project, and proof automation tools like Islaris. However, new and exciting platforms and tools also means that tooling is not yet mature and cannot be used for many important proof goals.

This project presents a flexible, hybrid approach for performing formal verification on such novel architectures, in a way that makes use of the benefits tooling gives us even as they are not yet ready, while having the potential for compositional and incremental replacement with automated tools as they continue to be developed.

We do so by marrying the ease of proof automation and the flexibility of theorem provers, providing an alternative semantics for Isla traces, and implementing them in Isabelle together with proof methods and full proof automation for broad classes of proofs.

With this method, proof goals for novel systems on new platforms can be achieved *now*, allowing much more rapid development of formally verified systems alongside, rather than after, the relevant proof automation tooling.

Contents

1 Introduction **3**

2 Background **5**

2.1 Context and aims . . . 5

2.1.1 CHERI and CheriOS . . . 5

2.1.2 seL4 . . . 6

2.1.3 Our aims . . . 6

2.2 Proof tooling . . . 7

3 Our approach **9**

4 Determinizing Isla semantics **11**

4.1 Isla syntax and semantics . . . 11

4.2 Deterministic Isla semantics . . . 14

4.2.1 Correctness . . . 17

4.3 Inter-trace semantics . . . 18

4.4 A note on special behaviours . . . 19

5 Constructing proofs **20**

5.1 Isla traces in Isabelle . . . 20

5.2 Properties of Isla traces . . . 24

5.3 Constructing proofs about code . . . 26

6 Analysis **31**

7 Conclusion and prospects **33**

7.1 Future work . . . 33

Chapter 1. Introduction

In the modern world, it is difficult to overstate the importance of the correctness and security of computers and the code that runs on them. With a vast array of systems critical to society, daily life, physical safety, and more, simple mistakes can lead to disastrous and far-reaching consequences [1]–[4].

Despite this, it is notoriously difficult to get both the design and implementation of systems to be correct, especially with ever-increasing complexity of both hardware and software [5], [6]. Two main methods are used to help create correct systems—testing and verification. Testing is the easier of the two, but can only show the presence of bugs. On the other hand, with formal verification, mathematical proofs can be written to show that systems satisfy certain desired properties [4], [5].

However, formal verification is a costly and difficult process, requiring a large amount of specialized expert work [7]. This has led to interest in systems comprising a small and formally verified *trusted computing base*, which maintains security guarantees even with the bulk of the system being comprised of untrusted components. A famous example is the seL4 microkernel [8], [9], which has formally verified correctness guarantees that allow users to piece together untrusted and unverified components without compromising basic guarantees [10], [11].

Also notable is the Digital Security by Design (DSbD) initiative and the Capability Hardware Enhanced RISC Instructions (CHERI) project [12], [13], which provides architecture extensions giving hardware-level memory protection and compartmentalization, eliminating whole classes of memory safety bugs. This also depends on formal verification to prove the correctness of these guarantees [14].

There are two primary approaches to performing such formal verification—one method is theorem proving using proof assistants. This is the approach taken for seL4 using the Isabelle proof assistant. This method requires huge amounts of work and care from specialized experts—the seL4 microkernel has code on the order of 10,000 SLOC (source lines of code), with 480,000 SLOC (!) of Isabelle proofs and specifications, with an estimated effort for the correctness proof of about 20 person-years [9].

The other approach is with full proof automation—an example is Isla and Islaris [15], which uses hardware models [16] and symbolic execution to automate proofs on machine code [17]. This method requires less work to prove properties of specific programs, but requires development of these complex tools in the first place. Applying them to a new platform, such as the myriad arising as a result of the CHERI project requires a significant research effort.

Our approach is the "middle ground" between the two. To do so, we take the simplified traces produced by Isla to its best effort, and fill in the gaps by allowing users to hand-write semantics/proofs for what is not yet supported by Isla/Islaris. We can then take advantage of existing automation while still being afforded the flexibility of hand-writing proofs.

Not only does this approach provide additional flexibility on top of convenient automation, it also allows compositional replacement of components as the automated tools are developed further and less hand-written stopgaps are required. In addition, it allows chaining with hardware models that is much more difficult to accomplish with only hand-written proofs.

Chapter 2. Background

2.1 Context and aims

We begin by examining the context within which this project was carried out—namely, we look at CHERI, CheriOS and seL4.

2.1.1 CHERI and CheriOS

The CHERI project extends existing conventional architectures such as MIPS, RISC-V and Armv8-A by means of instruction set extensions, introducing architectural capabilities implemented at a hardware level [12]. This helps eliminate important classes of memory bugs, while being a hybrid architecture, hence supporting incremental migration of existing systems that would not be possible with only a pure capability architecture [13]. This is supported with extensive ongoing work in formal verification, ranging from correctness of hardware/instruction-set architecture (ISA) specifications, to correctness of implementations and more [14], [18].

However, even with such capabilities, more work is still required to implement secure functionality such as isolation and least privilege, due to architectural capabilities being too coarse-grained. A notable solution presented to this comes in the form of the CheriOS microkernel and nanokernel [19].

In the CheriOS approach [19], the microkernel is not considered part of the trusted computing base, instead being treated as untrusted alongside services and userspace programs. The trusted computing base comprises only the underlying CHERI hardware and a "security hypervisor" referred to as the nanokernel. The nanokernel is written as a set of small assembly routines for CHERI MIPS and CHERI RISC-V (totalling approximately

3,300 SLOC) that implement higher level primitives for use with memory management, capability signing and isolation. In doing so, a small amount of code can give isolation/-correctness guarantees even when a (larger) operating system is unverified—for example, the Reservation primitive for memory management guarantees privacy of allocated memory even with an untrusted allocator. However, correctness of the nanokernel is yet to be formally verified.

This approach shows the value in incrementally providing higher-level guarantees. Rather than proving correctness of an entire operating system at once, we can begin with formally verified hardware and its guarantees, and use them as building blocks for a formally verified nanokernel. The nanokernel's guarantees can then be used to build higher levels of abstraction with increasing levels of guarantees, and proofs can be chained all the way down to the hardware level.

2.1.2 seL4

A different approach is taken by the seL4 microkernel. The entirety of the seL4 microkernel is formally verified, with capabilities (distinct from CHERI capabilities) implemented as a kernel level object in software, and manipulated with system calls [9]. The correctness of the microkernel is proven by formal proofs first on the C implementation [8], and then also at the binary level [20].

The isolation guarantees provided by seL4 capabilities and virtualization allow systems to be built incrementally, including by gradual retrofitting of existing systems by isolating components over time. A famous example is the Boeing ULB autonomous helicopter, where components were pulled out of the trusted computing base in multiple steps [10].

2.1.3 Our aims

With these two examples, it is clear that there is great benefit in proving the correctness of small, trusted layers of abstraction. By doing so, even large systems can benefit from

verification without having to directly perform verification at an impractical scale.

In this context, it is desirable to be able to verify the correctness of the CheriOS nanokernel, or similar implementations that provide higher-level primitives that need to be correct. Hence, this project aims to establish a proof method that can be applied to such proofs, filling in the gap between existing tooling. With that in mind, we next look at existing proof tooling that would be suitable for such verification tasks.

2.2 Proof tooling

Of great interest are the symbolic execution engine Isla [15], verification system Islaris [17] and the Sail ISA definition language [16], [21]. The combination of these tools allows automated verification of programs in the form of machine code, which is done against formal hardware models in Sail.

Firstly, the Sail language is used to specify the semantics of ISAs, which has tooling to produce emulators, run tests, perform symbolic evaluation, produce definitions for provers and so on. Notably, it is used by Isla to perform symbolic execution by producing *traces* from machine code, which are sequences of events like memory and register accesses. These traces and their semantics are then used by Islaris to perform reasoning and verification against a Coq spec [17].

This is much preferable compared to hand-writing ISA semantics, as typically such hand-written semantics cover a small subset of the ISA with simplifications that are based on human ideas of what instructions should do [17], [22], [23]. On the other hand, this approach chains directly to the ISA specification, so there is no gap between the semantics and hardware implementations (provided the implementation fulfills the specification, of course).

Sail ISA specifications are, however, huge and unwieldy for direct use in formal verification, which is where Isla comes in. By performing symbolic execution using the Sail model

to produce traces, which are then simplified automatically and used in Islaris, performing verification using the ISA specification becomes practical.

At a higher level than writing proofs about instructions, there are tools such as AutoCorres that allow verification of C code [24]. However, as we wish to also be able to verify programs like the CheriOS nanokernel written entirely in assembly, we cannot use such tools oriented for higher-level languages.

Unfortunately, we also cannot use Isla and Islaris directly at present. This is because Isla and Islaris currently support AArch64 and RISC-V, with ongoing work on CHERI Morello, meaning it cannot yet be used for verifying CHERI MIPS or CHERI RISC-V assembly.

Chapter 3. Our approach

In such a context, our new approach provides a hybrid method for constructing proofs to formally verify programs that are written for platforms that are currently not yet (or not yet fully) supported by Isla. This is done by combining Isla traces with the more flexible proving approach of hand-written proofs using the Isabelle proof assistant.

An overview of the approach is given in figure 3.1. Beginning with the verification target in the form of assembly instructions, these instructions are put through Isla to produce Isla traces. However, since we are working on platforms not (fully) supported by Isla, we write the traces for instructions that Isla cannot produce traces for by hand. Although Isla traces generated by Isla are large and contain many assumptions/assertions based on the hardware specification, we can include only as much detail as we need for the proofs we are writing.

With the program now represented in Isla traces, these can then be represented natively within the Isabelle proof assistant, which can then be combined with semantics defined in Isabelle to write proofs about the program.

Of course, writing such traces by hand loses us the advantages we get from using Isla

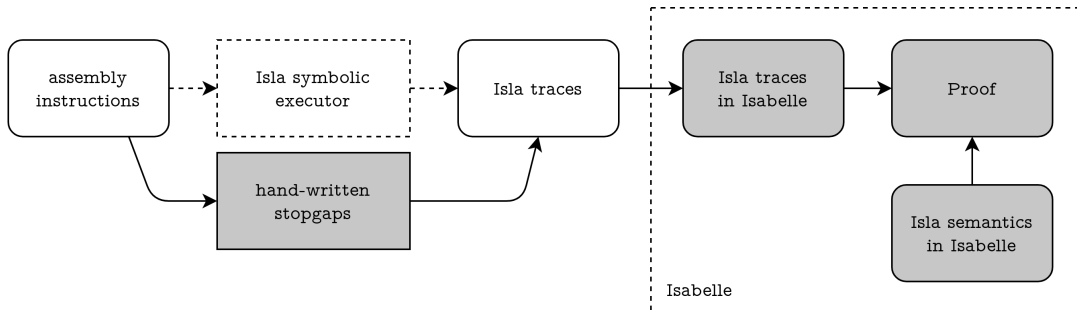

Figure 3.1: Overview of our approach. Greyed boxes are the contributions of this project.

and Sail specifications compared to hand-writing ISA semantics, but the key point is that these are merely temporary stopgaps. By writing these semantics as Isla traces, when it becomes that Isla now supports the platform we are writing proofs on, we can then replace these hand-written traces with Isla-generated ones based on the Sail specification, with little to no change required in the Isabelle proofs. This allows us to write proofs *now* and incrementally replace the stopgaps as and when possible thanks to further Isla development.

In order to achieve this, we first give an alternative deterministic semantics derived from the hugely non-deterministic Isla semantics with justifications for equivalence in chapter 4.

These semantics are then implemented in Isabelle, and proof methods and automation were developed for broad, common classes of proofs in chapter 5.

The full Isabelle theories for the implementation are provided in the appendix.

Chapter 4. Determinizing Isla semantics

4.1 Isla syntax and semantics

The Isla trace syntax used is given in figure 4.1 as presented in [17]. Traces correspond to individual instructions—sequences between instructions are not represented by sequential concatenation of traces, but rather the updating of the program counter (PC) register.

The simplest form of a trace is simply a sequence of events, concatenated by the :: operator. These events effectively happen in sequence, and correspond to register/memory read/writes, declaration and definition of constants, and assertions/assumptions. The events $\mathsf{ReadReg}(r, v)$ and $\mathsf{WriteReg}(r, v)$ correspond to reading/writing the value v from/to the register r respectively. $\mathsf{ReadMem}(b, a, n)$ and $\mathsf{WriteMem}(a, b, n)$ correspond to reading/writing the memory range from a to $a + n$ to/from the value b. $\mathsf{DeclareConst}(x, \tau)$ simply declares the constant x to be of type τ, while $\mathsf{DefineConst}(x, e)$ gives the constant x the value of the SMT expression e. Lastly, $\mathsf{Assert}(e)$ and $\mathsf{Assume}(e)$ simply assert that the SMT expression e evaluates to the Boolean value true, and $\mathsf{AssumeReg}(r, v)$ asserts that register r has value v.

$$
\begin{array}{rcll}
e & ::= & v \mid \mathsf{not}(e) \mid \mathsf{bvadd}(e_1, e_2) \mid \ldots & \text{(SMT-Expr)} \\
v & ::= & b \mid \mathsf{true} \mid \mathsf{false} \mid x \mid \ldots & \text{(Val)} \\
r & ::= & \rho \mid \rho.f & \text{(Reg)} \\
\tau & ::= & \mathsf{BitVec}(n) \mid \mathsf{Boolean} \mid \ldots & \text{(Type)} \\
j & ::= & \mathsf{ReadReg}(r, v) \mid \mathsf{WriteReg}(r, v) & \text{(Event)} \\
 & \mid & \mathsf{ReadMem}(v_d, v_a, n) \mid \mathsf{WriteMem}(v_a, v_d, n) & \\
 & \mid & \mathsf{AssumeReg}(r, v) \mid \mathsf{DeclareConst}(x, \tau) & \\
 & \mid & \mathsf{DefineConst}(x, e) \mid \mathsf{Assert}(e) \mid \mathsf{Assume}(e) & \\
t & ::= & [] \mid j :: t \mid \mathsf{Cases}(t_1, \ldots, t_n) & \text{(Trace)}
\end{array}
$$

Figure 4.1: Isla trace language syntax, directly reproduced from figure 4 of [17].

Branching within a trace happens using the $\mathsf{Cases}(t_1, \ldots, t_n)$ construct. Execution *non-deterministically* splits between each of the traces $t_1, \ldots, t_n$, but the events' assertions narrow down which branch is taken to only one. For example, a trace that would do t_1 if e was true and t_2 otherwise would be represented as $\mathsf{Cases}(\mathsf{Assert}(e) :: t_1, \mathsf{Assert}(\neg e) :: t_2)$.

As an example, a simplified trace of instruction `add x1, x2, x3` would look like

$$\begin{aligned}
&\mathsf{DeclareConst}(v_1, \mathsf{BitVec64}) :: \mathsf{DeclareConst}(v_2, \mathsf{BitVec64}) :: \mathsf{DeclareConst}(v_3, \mathsf{BitVec64}) \\
&\quad :: \mathsf{ReadReg}(\mathtt{x2}, v_1) :: \mathsf{ReadReg}(\mathtt{x3}, v_2) :: \mathsf{DefineConst}(v_4, v_1 + v_2) \\
&\quad :: \mathsf{WriteReg}(\mathtt{x1}, v_4) :: \mathsf{ReadReg}(\mathtt{PC}, v_3) :: \mathsf{DefineConst}(v_5, v_3 + 64) \\
&\quad :: \mathsf{WriteReg}(\mathtt{PC}, v_5)
\end{aligned}$$

which does the following in sequence: declares constants with names v_1, v_2, v_3 of the type of 64-bit bit vectors; reads the values of registers `x2` and `x3` into v_1 and v_2 respectively; computes the (64-bit bit vector) sum into v_4 and writes it to `x1`; reads the value of PC into v_3, increments it by 64 into v_5, and writes it back to PC.

By reducing every instruction to this small set of events in the Isla trace syntax, it then becomes far more practical to write semantics and hence proofs about these constructs rather than directly for each and every instruction in huge ISAs.

The operational semantics as given in [17] are reproduced here in figure 4.2. The state is represented by triple $\Sigma = (R, I, M)$, where R maps registers to values, I maps addresses (64-bit bit vectors) to the traces of the corresponding instructions, and M maps addresses to bytes of memory. Each map is finite and partial.

Note the explosively non-deterministic approach taken by these semantics—as an example, the semantics for DeclareConst allows the constant to take every possible value of the type, which is restricted by other events. In the example of `add x1, x2, x3` given above, the event $\mathsf{DeclareConst}(v_1, \mathsf{BitVec64})$ would non-deterministically allow v_1 to take every possible 64-bit value, but $\mathsf{ReadReg}(\mathtt{x2}, v_1)$ makes it such that every value that does

$$\textsc{step-read-reg-eq}\quad \frac{\Sigma[r] = v}{\langle \mathsf{ReadReg}(r, v) :: t, \Sigma\rangle \rightarrow \langle t, \Sigma\rangle}$$

$$\textsc{step-read-reg-neq}\quad \frac{\Sigma[r] \neq v}{\langle \mathsf{ReadReg}(r, v) :: t, \Sigma\rangle \rightarrow \top}$$

$$\textsc{step-write-reg}\quad \langle \mathsf{WriteReg}(r, v) :: t, \Sigma\rangle \rightarrow \langle t, \Sigma[r \mapsto v]\rangle$$

$$\textsc{step-read-mem-eq}\quad \frac{|b| = n \qquad \Sigma[a..a{+}n] = \mathsf{enc}(b)}{\langle \mathsf{ReadMem}(b, a, n) :: t, \Sigma\rangle \rightarrow \langle t, \Sigma\rangle}$$

$$\textsc{step-read-mem-event}\quad \frac{|b| = n \qquad \Sigma[a..a{+}n] = \bot \qquad \kappa = \mathsf{R}(a, b)}{\langle \mathsf{ReadMem}(b, a, n) :: t, \Sigma\rangle \xrightarrow{\kappa} \langle t, \Sigma\rangle}$$

$$\textsc{step-read-mem-neq}\quad \frac{|b| = n \qquad \Sigma[a..a{+}n] \neq \bot \qquad \Sigma[a..a{+}n] \neq \mathsf{enc}(b)}{\langle \mathsf{ReadMem}(b, a, n) :: t, \Sigma\rangle \rightarrow \top}$$

$$\textsc{step-write-mem}\quad \frac{|b| = n \qquad \Sigma[a..a{+}n] \neq \bot}{\langle \mathsf{WriteMem}(a, b, n) :: t, \Sigma\rangle \rightarrow \langle t, \Sigma[a..a{+}n \mapsto \mathsf{enc}(b)]\rangle}$$

$$\textsc{step-write-mem-event}\quad \frac{|b| = n \qquad \Sigma[a..a{+}n] = \bot \qquad \kappa = \mathsf{W}(a, b)}{\langle \mathsf{WriteMem}(a, b, n) :: t, \Sigma\rangle \xrightarrow{\kappa} \langle t, \Sigma\rangle}$$

$$\textsc{step-declare-const}\quad \frac{v \in \tau}{\langle \mathsf{DeclareConst}(x, \tau) :: t, \Sigma\rangle \rightarrow \langle t[v/x], \Sigma\rangle}$$

$$\textsc{step-define-const}\quad \frac{e \downarrow v}{\langle \mathsf{DefineConst}(x, e) :: t, \Sigma\rangle \rightarrow \langle t[v/x], \Sigma\rangle}$$

$$\textsc{step-assert-true}\quad \frac{e \downarrow \mathsf{true}}{\langle \mathsf{Assert}(e) :: t, \Sigma\rangle \rightarrow \langle t, \Sigma\rangle}$$

$$\textsc{step-assert-false}\quad \frac{e \downarrow \mathsf{false}}{\langle \mathsf{Assert}(e) :: t, \Sigma\rangle \rightarrow \top}$$

$$\textsc{step-assume-true}\quad \frac{e \downarrow \mathsf{true}}{\langle \mathsf{Assume}(e) :: t, \Sigma\rangle \rightarrow \langle t, \Sigma\rangle}$$

$$\textsc{step-assume-reg-true}\quad \frac{R[r] = v}{\langle \mathsf{AssumeReg}(r, v) :: t, \Sigma\rangle \rightarrow \langle t, \Sigma\rangle}$$

$$\textsc{step-cases}\quad \frac{1 \leq i \leq n}{\langle \mathsf{Cases}(t_1, \ldots, t_n), \Sigma\rangle \rightarrow \langle t_i, \Sigma\rangle}$$

$$\textsc{step-nil}\quad \frac{\Sigma[\mathsf{PC}] = a \qquad \Sigma[a] = t}{\langle [], \Sigma\rangle \rightarrow \langle t, \Sigma\rangle}$$

$$\textsc{step-nil-end}\quad \frac{\Sigma[\mathsf{PC}] = a \qquad \Sigma[a] = \bot \qquad \kappa = \mathsf{E}(a)}{\langle [], \Sigma\rangle \xrightarrow{\kappa} \top}$$

$$\textsc{step-fail}\quad \frac{\text{No other rule reduces } \langle t, \Sigma\rangle}{\langle t, \Sigma\rangle \rightarrow \bot}$$

Figure 4.2: Original operational semantics of Isla trace language, directly reproduced from figure 10 of [17].

not correspond to the value of register x2 leads to the failure state $\top$. This form of non-deterministic branching and restriction is also what allows the Cases construct to work.

The reason the semantics are given as such is because the Isla trace language is effectively a set of SMT constraints, which works well with the proof methodology used by Isla and Islaris. However, this presents unnecessary challenges in writing human-written proofs in Isabelle, as sets of possible executions have to be generated and discarded as they reach failure, as opposed to being able to reason about a deterministic execution in a step-wise manner.

4.2 Deterministic Isla semantics

This is why we use an alternate deterministic semantics for the Isla trace language, presented in this section. These deterministic semantics are much easier to reason about as we only need to worry about the state evolving in a single way at each step.

The reason we can do so is because generated Isla traces have some additional properties that restrict the kinds of traces that we see, meaning that not all syntactically possible traces need to be considered.

Firstly, type correctness is always preserved. For example, a constant will never be declared as a type in $\mathsf{DeclareConst}(x, \tau_1)$ before being used as a different type in $\mathsf{ReadReg}(x, v)$ where $v \in \tau_2 \neq \tau_1$. This means that we can completely ignore DeclareConst events in traces.

In addition, expressions are always "fully evaluated" before usage. What this means is that an expression that has yet to be calculated, with uninstantiated constants in it, will have all such constants be substituted by events preceding it. For example, consider the trace

$$\mathsf{ReadReg}(\mathtt{PC}, v_1) :: \mathsf{DefineConst}(v_2, v_1 + 64)$$

In this trace, the expression $v_1 + 64$ has the uninstantiated constant v_1, but when the prior event is evaluated, the value of the PC will be substituted into v_1 and the expression can be fully evaluated before we evaluate the DefineConst. What this means is that we can evaluate events from beginning to end, substituting constants as they are instantiated, without having to worry about later events affecting the values of constants in earlier events—traces such as the following never occur:

$$\mathsf{DefineConst}(v_2, v_1 + 64) :: \mathsf{ReadReg}(\texttt{PC}, v_1)$$

Generated Isla traces also have that Cases constructs will never have multiple branches that evaluate without failure. The restriction of branches by events such as Assume, AssumeReg and so on will always ensure that only a single branch will remain viable. Thanks to this, instead of non-deterministically choosing which branch to evaluate, we can simply evaluate each branch in turn and choose the single branch that does not lead to the failure state $\top$. In most cases, branches can be pruned very early in the trace as each branch's first event is typically an Assume, which helps make such computation relatively inexpensive.

Additionally, valid traces will never make undefined references. If a valid trace contains a read from a register, it is guaranteed that the register is defined in the machine state Σ. Likewise, the PC will never be undefined.

From here on, we refer to traces that fulfill these properties as "valid".

With these properties in mind, the alternate deterministic semantics are given in figure 4.3. These semantics are a big-step semantics $\rightarrow$ for the entirety of a trace (a single instruction), and evolve the tuple $\langle t, \Sigma \rangle$ where t is the trace and Σ is the state to the resulting state Σ'.

Non-branching events are simply evaluated from left to right, with the rest of the trace being updated as constants are instantiated and substituted, and the state Σ is updated

$$\frac{\Sigma[r] = v \quad \langle t[v/n], \Sigma \rangle \to \Sigma'}{\langle \mathsf{ReadReg}(r, n) :: t, \Sigma \rangle \to \Sigma'} \qquad \frac{\Sigma[r] = \top}{\langle \mathsf{ReadReg}(r, n) :: t, \Sigma \rangle \to \top}$$

$$\frac{\langle t, \Sigma[r \mapsto v] \rangle \to \Sigma'}{\langle \mathsf{WriteReg}(r, v) :: t, \Sigma \rangle \to \Sigma'} \qquad \frac{\langle t[v/x], \Sigma \rangle \to \Sigma'}{\langle \mathsf{DefineConst}(n, v) :: t, \Sigma \rangle \to \Sigma'}$$

$$\frac{\langle t, \Sigma \rangle \to \Sigma'}{\langle \mathsf{Assert}(\mathsf{true}) :: t, \Sigma \rangle \to \Sigma'} \qquad \frac{}{\langle \mathsf{Assert}(\mathsf{false}) :: t, \Sigma \rangle \to \top}$$

$$\frac{\Sigma[r] = v \quad \langle t, \Sigma \rangle \to \Sigma'}{\langle \mathsf{AssumeReg}(r, v) :: t, \Sigma \rangle \to \Sigma'} \qquad \frac{(\Sigma[r] \neq v) \vee (\Sigma[r] = \top)}{\langle \mathsf{AssumeReg}(r, v) :: t, \Sigma \rangle \to \top}$$

$$\frac{\langle t_i, \Sigma \rangle \to \Sigma' \quad \forall j \neq i.\ \langle t_j, \Sigma \rangle \to \top}{\langle \mathsf{Cases}(t_1, \dots, t_n), \Sigma \rangle \to \Sigma'} \qquad \frac{\nexists i.\ \text{previous condition holds}}{\langle \mathsf{Cases}(t_1, \dots, t_n), \Sigma \rangle \to \top} \qquad \frac{}{\langle [\,], \Sigma \rangle \to \Sigma}$$

Figure 4.3: Alternative, big-step deterministic intra-trace semantics for Isla trace language. $\Sigma[\cdot] = \top$ is used to represent a lookup failing, and $\langle t, \Sigma \rangle \to \top$ represents termination in failure while $\langle t, \Sigma \rangle \to \Sigma'$ represents successful termination in state Σ'.

as writes are performed that modify registers in R or memory in M.

Reads from registers and memory are done by performing the relevant lookups on R or M, and then substituting the found value for the constant in the rest of the trace. Since these maps are partial, if the lookups fail, the evaluation of the trace also fails $\langle t, \Sigma \rangle \to \top$. However, any valid trace—state combination should not lead to this failure, as the state should have instantiated all registers/memory used by the trace.

Writes are performed by simply updating Σ with the relevant value.

We note that the semantics of memory events have been excluded, as within the scope of this project, we represented any memory accesses in terms of register accesses.

Defines are handled by substituting all later occurrences of the constant with the given value, much like how reads are handled.

Asserts are handled by evaluating the given value and proceeding if it evaluates to true, terminating in failure otherwise. AssumeReg in particular does the same, but reading from R as ReadReg does, failing if the lookup fails or if the lookup returns false. This class of

events should only fail within a Cases construct if the trace is valid.

Lastly, the branching Cases construct simply takes the semantics of the first branch in the list that does not fail—remember that our additional properties above say that we will have exactly one such branch.

4.2.1 Correctness

In this section, we will reason about the correctness of the deterministic semantics by justifying correspondence between the original semantics in 4.2 and our alternative semantics in 4.3 in the case of "valid" traces.

Firstly, for ReadReg, due to valid traces having type correctness and never having invalid references, there is never any worry for ReadReg having multiple possible values it can take. Σ will always contain the value for the desired register, so the rule going to $\top$ is really only useful for detecting an invalid trace. Due to this, our rules for ReadReg correspond to STEP-READ-REG-EQ, and STEP-READ-REG-NEQ need never be considered.

Our alternate semantics for WriteReg are exactly the same as STEP-WRITE-REG, except written as big-step instead of small-step.

The original semantics' STEP-DECLARE-CONST only performs restriction of constant names to specific types, but due to type correctness of valid traces, they become noops and can be excluded from our syntax and semantics—when parsing Isla traces into Isabelle, we can simply remove all occurrences of DeclareConst.

The same justification for ReadReg also applies to AssumeReg—we need not worry about multiple possible values nor an undefined register in Σ.

With that assumption in place, we can see that our semantics for AssumeReg are fundamentally the same as Assert. These constructs only make sense in valid traces when they are in a Cases construct—if they are outside of one, they must evaluate to true or the trace becomes invalid. Hence, for the case that they are outside a Cases construct,

our presented rule of moving to $\top$ when they are false is sufficient by saying the trace is invalid.

Finally, we look at the Cases construct. In the original semantics, it simply non-deterministically branches to each of its branches. For valid traces, however, we have that one and only one branch will not lead to $\top$, hence our given semantics correspond well. In the implementation, we evaluate each branch in turn until we find one that terminates successfully, which is sufficient due to, once again, the fact that one and only one branch will do so for valid traces.

The remaining rule $\langle [\,], \Sigma \rangle$ is trivial, and the corresponding rules in the original semantics are covered in the next section.

4.3 Inter-trace semantics

The above has been concerned with intra-trace semantics, but to reason about programs with > 1 instruction, we also need to consider inter-trace semantics. As shown in figure 4.2, the inter-trace semantics are relatively simple—when a trace is evaluated until it is empty, meaning we have reached $\langle [\,], \Sigma \rangle$, the instruction located at the address pointed to by the PC is loaded as the next trace.

In the original semantics given in [17], the rule STEP-NIL-END defines what happens when there is no instruction located at the PC. In our interpretation, we consider this as "successful termination", to better suit our purposes of writing proofs about small subroutines. In other words, we interpret a program by running the intra-trace semantics on the instruction located at the address given by PC, then continue with the trace located at the address given by PC *after* the current trace has evaluated, and repeat until we reach a state where the PC points at no instruction.

4.4 A note on special behaviours

There are some registers that behave differently from other registers on specific architectures. For example, on RISC-V, the x0 register is constantly zero, so writes to it should not change its value from 0.

There are no mechanisms in the Isla trace semantics, or our alternative semantics, to handle such special behaviour—rather, there is no *need* for such mechanisms. The reason is that this kind of behaviour is captured in the *generation* of traces rather than the execution of traces. All manners of special behaviours like the above are written in the Sail hardware model, and Isla will generate traces accordingly. In the example of writing to x0, Isla will simply not generate an event like WriteReg(x0, v) thanks to the Sail model.

This is where we can see the strength of writing for Isla instead of hand-writing ISA semantics—we only need to consider the semantics of Isla traces and everything else is covered by trace generation using the Sail model. Of course, when hand-writing Isla traces, we will need to hand-write such considerations ourselves.

Chapter 5. Constructing proofs

With the syntax and semantics of Isla defined in the previous section, we now need to cast them into the Isabelle proof assistant [25] in order to construct proofs. Isabelle allows us to define sum and product datatypes similar to in ML, as well as total functions, inductive definitions and proof methods. It supports proofs in the form of "apply-scripts", where sequences of proof methods/tactics can be specified to be applied, as well as structured proofs similar to human proofs, with steps and justifications for each step. Lemmas and theorems can also be defined and reused.

In this section, excerpts from the Isabelle theory files have been typeset where they are illustrative, and the appendix contains the theory files in their entirety. These excerpts and the appendix are typeset using Isabelle's document preparation system, so although they appear to be in pretty LaTeX maths notation, they correspond exactly to the actual code. For example, the code

```
datatype trace =
    EmptyTrace
  | Seq event trace (infixr ":::" 100)
  | Cases "trace list"
```

is typeset as

datatype *trace* =
  EmptyTrace
| *Seq event trace* (**infixr** ''::: '' 100)
| *Cases ''trace list''*

5.1 Isla traces in Isabelle

We begin by examining how traces are represented in Isabelle.

An event is represented by the following datatype.

datatype *event* =
  ReadReg reg name — *name* must be of type *bv*64 *val*.
  | *WriteReg reg "bv*64 *val"*
  | *ReadMem name "bv*64 *val" nat* — Value to read to, source address, size to read.
  | *WriteMem "bv*64 *val" "bv*64 *val" nat* — Value to write, destination address, size to write.
  | *Assert "bool val"* — Subsumes Assume events.
  | *AssumeReg reg "bv*64 *val"*
  | *DefineConstBV*64 *name "bv*64 *val"*
  | *DefineConstBool name "bool val"*
  | *DefineConstBoolFromBV*64 *name "bv*64 *val"*

As an aside, note the distinction between Isabelle syntax where we have oblique identifiers with space-separated, curried parameters such as *ReadReg r n*, as opposed to mathematical, "human" syntax as in the previous sections, like ReadReg(r, n).

Notably, we assume that all registers are 64-bit bit vectors, and have different variants of DefineConst for defining constants that are bit vectors and Booleans. We also add a variant for converting a bit vector to a Boolean. These variants are necessary due to our usage of strongly-typed values.

We also roll Assert and Assume into a single *Assert* event, as they behave the same in our deterministic semantics.

Within events, values are defined with the following datatype.

datatype *'a val* =
  Val 'a — Instantiated.
  | *Name name*
  | *Monop "'a* ⇒ *'a" "'a val"*
  | *Binop "'a* ⇒ *'a* ⇒ *'a" "'a val" "'a val"*

A value is strongly-typed, and can either be a full-instantiated, shallowly-embedded value (e.g. *Val* 3), or an expression that is either a single name of a constant (e.g. v1) or a shallowly-embedded operator and recursive value operands.

With events defined, we represent traces as below. Sequential composition of events :: and the empty trace [ ] are represented by ::: and *EmptyTrace*, while the *Cases* construct contains a list of traces $t_1, \ldots, t_n$.

```
datatype trace =
   EmptyTrace
 | Seq event trace (infixr ":::" 100)
 | Cases "trace list"
```

Next, we define the machine state $\Sigma = (R, I, M)$ in Isabelle using the in-built Isabelle map datatype *HOL.HOL.Map.map* [26] for each of R, I and M.

With the Isabelle representation of traces and state done, we next define the intra-trace semantics. We do so by defining the total function *step*, of which an excerpt is given below.

```
function step :: "trace ∗ state ⇒ state option" where
  "step (ReadReg r n ::: t, (stR, stI, stM))
     = (case stR r of Some x ⇒ step (substValueBV64 n x t, (stR, stI, stM)) | - ⇒ None)"
| "step (WriteReg r v ::: t, (stR, stI, stM))
     = step (t, (stR(r ↦ v), stI, stM))"
| "step (DefineConstBV64 n v ::: t, st)
     = step (substValueBV64 n v t, st)"
| "step (Assert (Val True) ::: t, st) = step (t, st)"
| "step (Assert (Val False) ::: t, st) = None"
| "step (AssumeReg r (Val v) ::: t, (stR, stI, stM))
     = (case stR r of
          Some (Val v') ⇒ if v = v' then step (t, (stR, stI, stM)) else None
        | - ⇒ None)"
| "step (Cases (c#cs), st)
     = (case step (c, st) of None ⇒ step (Cases cs, st) | res ⇒ res)"
| "step (Cases [], st) = None"
| "step (EmptyTrace, st) = Some st"
| ...
by pat-completeness auto — Proves totality of this function (other than termination).
```

Note that the function takes a tuple of a *trace* and a *state*, and optionally returns a *state*. It returns a state if it successfully executes the entirety of the trace, and returns

None if it fails for any of the reasons described in the previous section.

There are helper functions *substValueBool* and *substValueBV64* that perform the substitution $t[v/x]$ for the correct types. They do so by stepping through the remainder of the trace, replacing all occurrences of the constant x (technically, the name of the constant) with the value v.

Recalling the property of valid traces that expressions are always "fully evaluated" before usage, we defer evaluation of expressions to these helper functions—when they substitute a constant name for a value, if that expression subtree is now fully instantiated, it recursively instantiates up the expression tree until it meets an uninstantiated value. For example, when substituting the value 10 for v_3 in $v_4 + (v_3 \times 3)$, the helper function does $(v_4 + (v_3 \times 3))[10/v_3] = v_4 + (10 \times 3) = v_4 + 30$.

The effect of this is that when *step* reaches an event that uses a value, such as *DefineConst*, if the trace is valid then the expression will always be fully evaluated and in the form *Val v*. We can see that in the cases of *step* where it is *not* fully evaluated, *None* is returned and evaluation fails due to the trace being invalid.

Lastly, we define the inter-state semantics. We begin by lifting the machine state triple to the following datatype

datatype *multitraceState* = *Running state* | *Err* | *Ok state*

where successful termination and failure are represented as *Ok* and *Err* respectively. Successful termination is reached when a trace successfully terminates with the PC pointing at no further instruction. Failure is reached when the PC is undefined, or if a trace fails—neither will happen with valid traces.

We then define the (inductive) "evolve" relation that goes from *multitraceState* to *multitraceState*, represented by the operator $\longrightarrow$. It is a right-unique relation due to execution being deterministic, and we are mainly interested in its reflexive transitive

closure written as $\longrightarrow^*$. We first define $\longrightarrow$ as *evolve* as follows:

```
fun runInstr :: "state ⇒ multitraceState" where
  "runInstr (stR, stI, stM) = (case stR ''PC'' of None ⇒ Err |
      Some (Val pc) ⇒ (case stI pc of None ⇒ Ok (stR, stI, stM) |
          Some t ⇒ (case step (t, (stR, stI, stM)) of None ⇒ Err |
              Some st' ⇒ Running st')))"
inductive evolve :: "multitraceState ⇒ multitraceState ⇒ bool" (infix "⟶" 55) where
  "runInstr st = mtst ⟹ Running st ⟶ mtst"
```

To define $\longrightarrow^*$, we then define $\longrightarrow|n|$ as n steps of $\longrightarrow$, then declare $\longrightarrow^*$ as syntactic sugar as follows:

```
abbreviation evolveStar :: "multitraceState ⇒ multitraceState ⇒ bool"
   (infix "⟶*" 55) where
  "mtst ⟶* mtst' ≡ ∃ n. mtst ⟶|n| mtst'"
```

Implementation details of all of the above and more can be found in the full theory files in the appendix.

5.2 Properties of Isla traces

Due to Isabelle's logic being based on HOL, which is a logic of total functions, when defining this function we have to prove totality and termination. For most functions, such as the helper value substitution functions, Isabelle can do this automatically, but due to the *step* function's more involved recursion, we need to provide a termination proof.

To do so, we define a measure on traces

$$\mathsf{measureTrace}([\,]) = 0$$

$$\mathsf{measureTrace}(e :: t) = 1 + \mathsf{measureTrace}(t)$$

$$\mathsf{measureTrace}(\mathsf{Cases}(t_1, \dots, t_n)) = 1 + n + \sum_{i=1}^{n} \mathsf{measureTrace}(t_i)$$

and can prove that the helper functions do not increase the measure of the trace, while every recursive call to *step* strictly decreases the measure—hence *step* is total and always

terminates. This also leads to the nice, sensible result that our semantics for single traces always terminates.

The details of the termination proof can be located in the appendix.

In addition, we also prove many other useful lemmas and theorems about the inter-trace execution semantics. Notably, we prove that $\longrightarrow^*$ is reflexive and transitive, and is deterministic in reaching a terminating state (if it does); i.e. if it reaches successful termination or failure, there is no other terminating state it can reach. The former is a sanity check for our definition in terms of $\longrightarrow|n|$, showing that we have indeed defined the reflexive transitive closure.

The latter is especially useful when constructing proofs. Due to the inductive definition of $\longrightarrow^*$, it is not a total function like *step* was—and it shouldn't be; there are clearly programs that do not terminate, and $\longrightarrow^*$ should loop infinitely for those. With the fact that it is deterministic in reaching a terminating state, we know that if we can prove that a program reaches successful termination, that is the *only* termination result it can possibly reach.

Phrased differently in a proof-oriented way, this means that if wish to prove properties about the state a program reaches upon termination, finding a terminating state that it *can* reach and proving that this existential witness fulfills the properties is sufficient. (This is true because $\longrightarrow$ is right-unique.) This lemma is cast and proven in Isabelle as

lemma *evolveStarOkWitnessEnough*:
"$(\exists\, st'.\ mtst \longrightarrow^* Ok\ st' \land P(st')) \Longrightarrow (mtst \longrightarrow^* Ok\ st' \longrightarrow P(st'))$"
using *evolveStarDetermOk* **by** *blast*

Details of proofs, as well as the remaining proven lemmas can be found in the appendix. These lemmas can be invoked in proofs as necessary.

5.3 Constructing proofs about code

We are now ready to see concrete proofs about traces generated from code. Firstly, we look at the simplified trace for the instruction `add x1, x1, x2` which places the sum of registers `x1` and `x2` in `x1`.

```
abbreviation addAABTrace :: trace where
  "addAABTrace ≡
      ReadReg ''x1'' 1
  ::: ReadReg ''x2'' 2
  ::: DefineConstBV64 3 (Binop (+) (Name 1) (Name 2))
  ::: ReadReg ''PC'' 4
  ::: DefineConstBV64 5 (Binop (+) (Name 4) (Val 64))
  ::: WriteReg ''x1'' (Name 3)
  ::: WriteReg ''PC'' (Name 5)
  ::: EmptyTrace"
```

Since *step* is a total function, we can use Isabelle's **value** command to evaluate what this trace does on a simple state.

```
value "step (addAABTrace, (Map.empty
  ( ''x1'' ↦ Val x1
  , ''x2'' ↦ Val x2
  , ''PC'' ↦ Val pc
  ), Map.empty, Map.empty))"
```

which gives us the value

Some (λu. *if* u = ''PC'' *then Some* (*Val* ($pc + 64$)) *else if* u = ''x1'' *then Some* (*Val* ($x1 + x2$)) *else if* u = ''PC'' *then Some* (*Val pc*) *else if* u = ''x2'' *then Some* (*Val x2*) *else if* u = ''x1'' *then Some* (*Val x1*) *else None*, *Map.empty*, *Map.empty*)

Note that none of the registers are initialized to concrete values, but are variables in Isabelle—this shows us that symbolic execution of traces works as expected. We also see that `x1`'s value is set to `x1` + `x2` as expected.

More interestingly, we can write a theorem statement that says: given a machine state

where registers x1, x2 and PC are defined, stepping through the trace for `add x1, x1, x2` will always result in a state where x1 is set to $x1+x2$. Isabelle can prove this automatically using the `auto` method.

```
theorem
  "⟦ stR ''x1'' = Some (Val x1); stR ''x2'' = Some (Val x2); stR ''PC'' = Some (Val pc)
  ; step (addAABTrace, (stR, stI, stM)) = Some (stR', stI', stM')
  ⟧ ⟹ stR' ''x1'' = Some (Val (x1 + x2))" by auto
```

Next, we examine the same single instruction, but now using the inter-trace semantics $\longrightarrow^*$. We can write a theorem statement saying: given a machine state where the same registers are defined, the PC is set to 64, the instruction at address 64 is `add x1, x1, x2`, and there is no instruction at address 128, if the machine state reaches a terminating state successfully, that terminating state will have that x1 is set to $x1 + x2$ (henceforth referred to as property Q).

```
theorem
    "stR ''x1'' = Some (Val x1) ∧ stR ''x2'' = Some (Val x2) ∧ stR ''PC'' = Some (Val 64)
       ∧ stI 64 = Some addAABTrace ∧ stI 128 = None
  ⟹ Running (stR, stI, stM) ⟶* Ok st'
  ⟹ (fst st') ''x1'' = Some (Val (x1 + x2))"
    (is "?assms ⟹ ?mtst ⟶* Ok ?st' ⟹ ?Q(?st')")
```

This proof is a bit more involved as we need to explicitly appeal to the determinism of $\longrightarrow^*$ in reaching successful termination.

Since Isabelle's default automated methods do not know that expanding it with existential quantification is the correct way to use determinism to construct the proof, we use the structured Isar proof functionality to explicitly tell Isabelle the steps we wish to take.

Using pattern matching, we first prove the lemma that there *exists* a successful termination state that is reached, and that that state fulfills Q.

proof −
  let "?P(st′)" = "?mtst ⟶* Ok st′"
  have "?assms ⟹ ∃ st′. ?P(st′) ∧ ?Q(st′)"
    by (*rule exI*, *auto*, (*rule evolveStarStepLeft*, *rule evolve.intros*, *auto*)+)

Then, appealing to determinism, with the lemma we proved earlier from the previous section, *evolveStarOkWitnessEnough*, we can prove that for *any* successful termination state that can be reached, that state fulfills predicate Q. We can then finally prove the original theorem statement using the *blast* method, which is a capable first-order logic reasoning method—just what we need.

  then have "?assms ⟹ ∀ st′. ?P(st′) ⟶ ?Q(st′)" **using** *evolveStarDetermOk* **by** *blast*
  then show "?assms ⟹ ?P(?st′) ⟹ ?Q(?st′)" **by** *blast*
qed

We now move to a multiple instruction program, corresponding to

```
128: add x1, x1, x2
192: mov x1, x2
256: add x1, x1, x2
320:
```

We want to prove that after successful termination, x1 is set to $2 \times$ x1 $+ 2 \times$ x2 and x2 is set to x1 + x2. This theorem's proof can be completed in the exact same way as the previous theorem's, so we present the theorem statement but leave the proof text to the appendix—thanks to pattern matching, "the exact same way" is no exaggeration.

theorem
  "stR ″x1″ = Some (Val x1) ∧ stR ″x2″ = Some (Val x2) ∧ stR ″PC″ = Some (Val 128)
    ∧ stI 128 = Some addAABTrace ∧ stI 192 = Some movABTrace
    ∧ stI 256 = Some addAABTrace ∧ stI 320 = None
  ⟹ Running (stR, stI, stM) ⟶* Ok st′
  ⟹ (fst st′) ″x1″ = Some (Val (2 ∗ x1 + 2 ∗ x2))
    ∧ (fst st′) ″x2″ = Some (Val (x1 + x2))"

We begin to see a pattern that statements of the form

$$assms \Longrightarrow Running\ mtst \longrightarrow^{*} Ok\ st' \Longrightarrow Q(st')$$

can be proved using this same strategy. This form is extremely useful—it encapsulates theorem statements that say "given some assumptions, if the program terminates, it ends up in a state that fulfills some conditions Q".

Refactoring the structured Isar proof as an apply script by reversing the proof process and nudging the solver gently in the right direction, we obtain a proof automation method as given by

method *step-evolve* = (*rule evolveStarStepLeft*, *rule evolve.intros*, *auto*)
method *prove-exists* = (*rule exI*, *auto*, *step-evolve+*)
method *proof-automation* =
 (*unfold atomize-imp*, *rule impI*, *rule evolveStarOkWitnessEnough*, *prove-exists*)

which unfolds the second meta implication $\Longrightarrow$ into logical implication $\longrightarrow$, uses determinism to convert the goal into an existential statement, then steps through $\longrightarrow^{*}$repeatedly.

We can test this proof method on the following program that squares `x1` in place assuming `x2` and `x3` are zeroed.

```
        addi x2, x1, 0
        addi x3, x1, -1
loop:
        beq x0, x3, end
        add x1, x1, x2
        addi x3, x3, -1
        beq x0, x0, loop
end:
```

Indeed, the proof automation method we derived works perfectly:

theorem
$''x1 = \mathit{Val}\ 3$
$\Longrightarrow \mathit{Running}\ (\mathit{initState}\ x1) \longrightarrow^{*} \mathit{Ok}\ st'$
$\Longrightarrow (\mathit{fst}\ st')\ ''x1'' = \mathit{Some}\ (\mathit{Val}\ 9)''$
by *proof-automation*

We can expect that a great number of such proofs can be completed using just the proof-automation method, but having cast it in Isabelle together with a large selection of helpful auxiliary lemmas means that more complicated proofs can be constructed flexibly, while using the proof methods as sub-components.

All the definitions and proofs in this section can be found in full in the appendix.

Chapter 6. Analysis

This approach has proved to be useful on small, "provable" procedures—if a hand-written proof was possible for the program in the first place, it can be done now using this method while utilizing Isabelle to make sure the proofs are correct. In addition, instead of leaving instruction semantics up to humans, we can rely on Isla and Sail where possible, and hand-write traces as and when necessary.

This stopgap hand-writing can be incrementally and compositionally phased out as Isla and Sail model development begin adding support for instructions/architectures that our proofs use, with minimal change to the proofs' overall structure.

This combination of Isla-generated and hand-written traces allows easy expansion to unsupported instructions and platforms, which is especially important given the benefits that small, incremental formal verification on top of CHERI architectures can give us, and the relative immaturity of tooling for new CHERI architectures compared to conventional, decades-old architectures like x86. For completely unsupported platforms, using Isla to generate traces for analogous instructions in a supported platform can also be a good starting point for hand-writing traces.

There are also disadvantages to this method—it is not fully automated and requires manual work. When proofs need to be written in forms not supported by the proof automation method, it once again becomes necessary to write Isabelle proofs by hand, which has a steep learning curve. Even when proof automation works, hand-writing Isla traces is still a significant task, which does not scale that well to larger procedures. At the same time, this effort is unlikely to be significantly higher than hand-writing ISA semantics, which also loses the advantages of chaining with Isla and Sail models.

Another disadvantage is the fact that we have used an alternate Isla semantics. We reasoned about the correctness of the alternate semantics by appealing to their equivalence to the original semantics under the properties of "valid" traces, but these are hand-written paper proofs that are not formally done with a proof assistant or equivalent.

Despite these disadvantages, this approach is still valuable for bridging the gap between proof automation and manual proofs for small programs on new architectures, which the author believes is of significant value, especially in the CHERI scene where incremental guarantees are extremely useful and new platforms are aplenty.

Chapter 7. Conclusion and prospects

In summary, given the practical usefulness of formally verifying incrementally higher-level guarantees on top of platforms like CHERI, as well as the relative immaturity of proof tooling on these new platforms, this project has presented a valuable middle ground by adding flexibility, alternate semantics, and proof automation, enabling proofs to be written about critical programs that can be chained with tooling as it matures.

7.1 Future work

Firstly, there are opportunities for automation to be added to the proof process. The translation from Isla traces to their Isabelle embeddings are currently done by hand, even for those that are generated by the Isla tool—there is potential for scripts to automate the parsing of these traces, and perhaps even parsing of Isla hardware configuration files to initialize machine states.

It would also be extremely useful for the proofs of correctness of the alternate semantics to be cast in a proof assistant to be machine-checked, as it would greatly increase the confidence in the correctness of proofs written using the alternate semantics. This would require embedding the original semantics in Isabelle, writing a predicate on validity of traces, and then proving equivalence under that predicate.

Now that the proof methodology and automation has been completed, it would definitely be interesting for future work to be done in applying it to verification of the CheriOS nanokernel, or other such small assembly programs for which correctness is critical. The semantics for memory, and assumptions like size of registers and semantics of bit vector arithmetic can be easily modified as required for these applications.

References

[1] D. Price, "Pentium FDIV flaw-lessons learned," *IEEE Micro*, vol. 15, no. 2, pp. 86–88, Apr. 1995, ISSN: 02721732. DOI: 10.1109/40.372360. [Online]. Available: http://ieeexplore.ieee.org/document/372360/ (visited on 04/29/2023).

[2] N. Leveson and C. Turner, "An investigation of the Therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, Jul. 1993, ISSN: 0018-9162. DOI: 10.1109/MC.1993.274940. [Online]. Available: http://ieeexplore.ieee.org/document/274940/ (visited on 04/29/2023).

[3] G. Le Lann, "An analysis of the Ariane 5 flight 501 failure-a system engineering perspective," in *Proceedings International Conference and Workshop on Engineering of Computer-Based Systems*, Monterey, CA, USA: IEEE Computer. Soc. Press, 1997, pp. 339–346, ISBN: 978-0-8186-7889-9. DOI: 10.1109/ECBS.1997.581900. [Online]. Available: http://ieeexplore.ieee.org/document/581900/ (visited on 04/29/2023).

[4] C. Baier and J.-P. Katoen, *Principles of Model Checking*. Cambridge, Mass: The MIT Press, 2008, 975 pp., ISBN: 978-0-262-02649-9.

[5] V. D'Silva, D. Kroening, and G. Weissenbacher, "A Survey of Automated Techniques for Formal Software Verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, Jul. 2008, ISSN: 0278-0070. DOI: 10.1109/TCAD.2008.923410. [Online]. Available: http://ieeexplore.ieee.org/document/4544862/ (visited on 04/29/2023).

[6] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: A survey," *ACM Transactions on Design Automation of Electronic Systems*, vol. 4, no. 2, pp. 123–193, Apr. 1999, ISSN: 1084-4309, 1557-7309. DOI: 10.1145/307988.307989. [Online]. Available: https://dl.acm.org/doi/10.1145/307988.307989 (visited on 04/29/2023).

[7] H. Amjad, "Combining model checking and theorem proving," University of Cambridge, UCAM-CL-TR-601, Sep. 2004.

[8] G. Klein, K. Elphinstone, G. Heiser, *et al.*, "seL4: Formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, Big Sky Montana USA: ACM, Oct. 11, 2009, pp. 207–220, ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596. [Online]. Available: https://dl.acm.org/doi/10.1145/1629575.1629596 (visited on 11/04/2022).

[9] G. Klein, J. Andronick, K. Elphinstone, *et al.*, "Comprehensive formal verification of an OS microkernel," *ACM Transactions on Computer Systems*, vol. 32, no. 1, pp. 1–70, Feb. 2014, ISSN: 0734-2071, 1557-7333. DOI: 10.1145/2560537. [Online]. Available: https://dl.acm.org/doi/10.1145/2560537 (visited on 11/04/2022).

[10] D. Cofer, A. Gacek, J. Backes, *et al.*, "A Formal Approach to Constructing Secure Air Vehicle Software," *Computer*, vol. 51, no. 11, pp. 14–23, Nov. 2018, ISSN: 0018-9162, 1558-0814. DOI: 10.1109/MC.2018.2876051. [Online]. Available: https://ieeexplore.ieee.org/document/8625938/ (visited on 11/12/2022).

[11] I. Kuz, Y. Liu, I. Gorton, and G. Heiser, "CAmkES: A component model for secure microkernel-based embedded systems," *Journal of Systems and Software*, vol. 80, no. 5, pp. 687–699, May 2007, ISSN: 01641212. DOI: 10.1016/j.jss.2006.08.039. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S016412120600224X (visited on 11/12/2022).

[12] J. Woodruff, R. N. Watson, D. Chisnall, *et al.*, "The CHERI capability model: Revisiting RISC in an age of risk," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 457–468, Oct. 16, 2014, ISSN: 0163-5964. DOI: 10.1145/2678373.2665740. [Online]. Available: https://dl.acm.org/doi/10.1145/2678373.2665740 (visited on 10/10/2022).

[13] R. N. Watson, J. Woodruff, P. G. Neumann, *et al.*, "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization," in *2015 IEEE Symposium on Security and Privacy*, San Jose, CA: IEEE, May 2015, pp. 20–37, ISBN: 978-1-4673-6949-7. DOI: 10.1109/SP.2015.9. [Online]. Available: https://ieeexplore.ieee.org/document/7163016/ (visited on 10/10/2022).

[14] K. Nienhuis, A. Joannou, T. Bauereiss, *et al.*, "Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process," in *2020 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA: IEEE, May 2020, pp. 1003–1020, ISBN: 978-1-72813-497-0. DOI: 10.1109/SP40000.2020.00055. [Online]. Available: https://ieeexplore.ieee.org/document/9152777/ (visited on 10/10/2022).

[15] A. Armstrong, B. Campbell, B. Simner, C. Pulte, and P. Sewell, "Isla: Integrating Full-Scale ISA Semantics and Axiomatic Concurrency Models," in *Computer Aided Verification*, A. Silva and K. R. M. Leino, Eds., vol. 12759, Cham: Springer International Publishing, 2021, pp. 303–316, ISBN: 978-3-030-81684-1 978-3-030-81685-8. DOI: 10.1007/978-3-030-81685-8_14. [Online]. Available: https://link.springer.com/10.1007/978-3-030-81685-8_14 (visited on 04/25/2023).

[16] A. Armstrong, T. Bauereiss, B. Campbell, *et al.*, "ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS," *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1–31, POPL Jan. 2, 2019, ISSN: 2475-1421. DOI: 10.1145/3290384. [Online]. Available: https://dl.acm.org/doi/10.1145/3290384 (visited on 05/29/2023).

[17] M. Sammler, A. Hammond, R. Lepigre, *et al.*, "Islaris: Verification of machine code against authoritative ISA semantics," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, San Diego CA USA: ACM, Jun. 9, 2022, pp. 825–840, ISBN: 978-1-4503-9265-5. DOI: 10.1145/3519939.3523434. [Online]. Available: https://dl.acm.org/doi/10.1145/3519939.3523434 (visited on 01/02/2023).

[18] T. Bauereiss, B. Campbell, T. Sewell, *et al.*, "Verified Security for the Morello Capability-enhanced Prototype Arm Architecture," in *Programming Languages and Systems*, I. Sergey, Ed., vol. 13240, Cham: Springer International Publishing, 2022, pp. 174–203, ISBN: 978-3-030-99335-1 978-3-030-99336-8. DOI: 10.1007/978-3-030-99336-8_7. [Online]. Available: https://link.springer.com/10.1007/978-3-030-99336-8_7 (visited on 05/02/2023).

[19] L. Esswood, "CheriOS: Designing an untrusted single-address-space capability operating system utilising capability hardware and a minimal hypervisor," University of Cambridge, Jul. 2020.

[20] T. A. L. Sewell, M. O. Myreen, and G. Klein, "Translation validation for a verified OS kernel," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 471–482, Jun. 23, 2013, ISSN: 0362-1340, 1558-1160. DOI: 10.1145/2499370.2462183. [Online]. Available: https://dl.acm.org/doi/10.1145/2499370.2462183 (visited on 05/25/2023).

[21] K. E. Gray, G. Kerneis, D. Mulligan, C. Pulte, S. Sarkar, and P. Sewell, "An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors," in *Proceedings of the 48th International Symposium on Microarchitecture*, Waikiki Hawaii: ACM, Dec. 5, 2015, pp. 635–646, ISBN: 978-1-4503-4034-2. DOI: 10.1145/2830772.2830775. [Online]. Available: https://dl.acm.org/doi/10.1145/2830772.2830775 (visited on 05/29/2023).

[22] S. Flur, K. E. Gray, C. Pulte, *et al.*, "Modelling the ARMv8 architecture, operationally: Concurrency and ISA," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg FL USA: ACM, Jan. 11, 2016, pp. 608–621, ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837615. [Online]. Available: https://dl.acm.org/doi/10.1145/2837614.2837615 (visited on 05/24/2023).

[23] A. Fox and M. O. Myreen, "A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture," in *Interactive Theorem Proving*, M. Kaufmann and L. C. Paulson, Eds., red. by D. Hutchison, T. Kanade, J. Kittler, *et al.*, vol. 6172, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 243–258, ISBN: 978-3-642-14051-8 978-3-642-14052-5. DOI: 10.1007/978-3-642-14052-5_18. [Online]. Available: http://link.springer.com/10.1007/978-3-642-14052-5_18 (visited on 05/29/2023).

[24] D. Greenaway, J. Andronick, and G. Klein, "Bridging the Gap: Automatic Verified Abstraction of C," in *Interactive Theorem Proving*, L. Beringer and A. Felty, Eds., red. by D. Hutchison, T. Kanade, J. Kittler, *et al.*, vol. 7406, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 99–115, ISBN: 978-3-642-32346-1 978-3-642-

32347-8. DOI: 10.1007/978-3-642-32347-8_8. [Online]. Available: http://link.springer.com/10.1007/978-3-642-32347-8_8 (visited on 01/19/2023).

[25] T. Nipkow, M. Wenzel, and L. C. Paulson, Eds., *Isabelle/HOL* (Lecture Notes in Computer Science), red. by G. Goos, J. Hartmanis, and J. Van Leeuwen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, vol. 2283, ISBN: 978-3-540-43376-7 978-3-540-45949-1. DOI: 10.1007/3-540-45949-9. [Online]. Available: http://link.springer.com/10.1007/3-540-45949-9 (visited on 12/15/2022).

[26] T. Nipkow. "Theory Map." (1997–2003), [Online]. Available: https://isabelle.in.tum.de/library/HOL/HOL/Map.html (visited on 12/15/2022).

Appendix A. Full Isabelle theories

In this Appendix, the document preparation feature of Isabelle is used to typeset the entirety of the Isabelle theories written for the project.

The contents of the theories correspond exactly to the code in the theory files, and the order of sections in this appendix are based on the order that these theories must be loaded and evaluated. The ROOT file that defines the entire session is as follows:

```
session proj4isabelle = HOL +
  options [document = pdf, document_output = "output"]
  sessions
    "HOL-Eisbach"
  theories [document = false]
    "HOL-Eisbach.Eisbach"
  theories
    Syntax
    State
    Semantics
    Multitrace
    Automation
    Examples
  document_files
    "root.tex"
```

A note on word count. Although the appendix appears to be full of natural language, they are comments within program text, and hence have not been included in the word count. The author considered the entirety of the appendix, other than this preamble, to be part of the allowed 40 additional pages of material.

A.1 Syntax definitions in `Syntax.thy`

```
theory Syntax
  imports Main
begin
```

In this theory, we define the syntax of Isla traces *trace* and events *event*, with relevant helper datatypes. There are a number of assumptions/approximations made as well that are outlined in this theory that can be edited as necessary for the proof goals at the time.

Registers are referred to by their names as *string*s. For example, register `x1` would be represented by the string ''*x*1'', and the program counter (PC) as ''*PC*''.

```
type-alias reg = string
```

Next, we define values/constants that are strongly typed. The "type" of the value is a type parameter to the datatype, so constants that are 64-bit bit vectors would be *bv*64 *val* while those that are Booleans would be *bool val*.

Names for values (constants) are represented just by the corresponding number, for example, v38 is represented simply by 38::*nat*.

```
type-alias name = nat
```

Values themselves can already be instantiated with a shallowly-embedded value in *Val 'a*, or can be an expression to be calculated. In the latter case, they are either a single named constant *Name name*, or an expression tree with non-leaf nodes as unary operators (*Monop op operand*) or binary operators (*Binop op operand*1 *operand*2). The operators *op* are shallowly embedded as Isabelle operators, and operands are values.

```
datatype 'a val =
    Val 'a — Instantiated.
  | Name name
  | Monop "'a ⇒ 'a" "'a val"
  | Binop "'a ⇒ 'a ⇒ 'a" "'a val" "'a val"
```

For this project, we only have to consider values that are either booleans, or bit vectors

of 64-bit length. We represent 64-bit bitvectors simply as *int*s for now, but this type can be swapped out as necessary for proofs involving e.g. signed/unsigned arithmetic.

type-alias *bv64* = *int*

Next is the type of Isla trace events. We make the assumption here that all registers take values of type *bv64*. Notably, due to our use of strongly-typed values as opposed to the original syntax, we have to distinguish between DefineConst events of differing types, with *DefineConstBV64*, *DefineConstBool* and *DefineConstBoolFromBV64*. The last case takes a *bv64 val* and puts it in a name of type *bool val*, which is necessary for marshalling from comparisons to Boolean cases.

datatype *event* =
  ReadReg reg name — *name* must be of type *bv64 val*.
  | *WriteReg reg "bv64 val"*
  | *ReadMem name "bv64 val" nat* — Value to read to, source address, size to read.
  | *WriteMem "bv64 val" "bv64 val" nat* — Value to write, destination address, size to write.
  | *Assert "bool val"* — Subsumes Assume events.
  | *AssumeReg reg "bv64 val"*
  | *DefineConstBV64 name "bv64 val"*
  | *DefineConstBool name "bool val"*
  | *DefineConstBoolFromBV64 name "bv64 val"*

Lastly, we define the type of traces. Sequencing is done in the usual functional list way, with the empty list *EmptyTrace* and cons operator *Seq*, with syntactic sugar :::. The syntactic sugar allows us to write $e :: t$ in mathematical notation as $e ::: t$ in Isabelle. We use a triple colon instead of the original double colon as :: is used in Isabelle as the "type of" operator.

The Cases($t_1, \dots, t_n$) construct is represented with *Cases*, which takes $t_1, \dots, t_n$ as an Isabelle list. With "valid" traces, only one branch should successfully terminate.

datatype *trace* =
  EmptyTrace
  | *Seq event trace* (**infixr** ":::" 100)
  | *Cases "trace list"*

end

A.2 Machine state definition in `State.thy`

theory *State*
imports *Main Syntax*
begin

This short theory simply defines the types of machine states we use. States are of type *state*, which is a tuple $\Sigma = (R, I, M)$:

- R, a register state mapping register names to their corresponding values,
- I, an instruction store mapping addresses to traces of instructions,
- and M, a memory state mapping addresses to bytes.

Each of the three maps are implemented as *map*s, which are simply partial functions with the type synonym given by (reproduced from inbuilt theory *HOL.Map*)

type-synonym ('a, 'b) "map" = "'a ⇒ 'b option" (**infixr** "⇀" 0)

The definitions of the three maps are respectively as follows:

type-synonym *regState* = "(*reg*, *bv*64 *val*) *map*"
type-synonym *instrs* = "(*bv*64, *trace*) *map*"

type-synonym *byte* = *char* — Helper type synonym.
type-synonym *memState* = "(*bv*64 *val*, *byte*) *map*"

which finally gives us the machine state type definition:

type-synonym *state* = "*regState* ∗ *instrs* ∗ *memState*"

end

A.3 Deterministic semantics in `Semantics.thy`

theory *Semantics*
  imports *Main Syntax State*
begin

This theory is concerned with defining the deterministic semantics for Isla traces that we introduced in the main text. We first begin with defining several helper functions before the main *step* function which corresponds to the actual big-step intra-trace semantics.

A.3.1 Value substitution

This subsection defines helper functions for evaluating value substitution for the rest of the trace; namely, it defines $t[v/x]$ for value v, constant name x and trace t.

This function *substValue n v e* performs the substitution $e[v/n]$ on value e. It evaluates subexpressions as and when all operands are fully instantiated, so that in valid traces, values are always fully instantiated before they are reached in the trace evaluation. It can do so for any type of value.

```
fun substValue :: "name ⇒ 'a val ⇒ 'a val ⇒ 'a val" where
  "substValue n v (Name n') = (if n = n' then v else Name n')"
| "substValue n v (Val v') = Val v'"
| "substValue n v (Monop op v1) = (case substValue n v v1 of
      Val v1' ⇒ Val (op v1') | v1' ⇒ Monop op v1')"
| "substValue n v (Binop op v1 v2) = (case (substValue n v v1, substValue n v v2) of
      (Val v1', Val v2') ⇒ Val (op v1' v2') | (v1', v2') ⇒ Binop op v1' v2')"
```

This function *substValueBV64 n v t* performs the substitution $t[v/n]$ on *trace* t, where the value in question is a 64-bit bit vector :: *bv64*.

```
fun substValueBV64 :: "name ⇒ bv64 val ⇒ trace ⇒ trace" where
  "substValueBV64 - - EmptyTrace = EmptyTrace"
| "substValueBV64 n v (WriteReg r v' ::: t) =
    WriteReg r (substValue n v v') ::: substValueBV64 n v t"
| "substValueBV64 n v (ReadMem n' v' s ::: t) =
    ReadMem n' (substValue n v v') s ::: substValueBV64 n v t"
```

```
| "substValueBV64 n v (WriteMem v1 v2 s ::: t) =
   WriteMem (substValue n v v1) (substValue n v v2) s ::: substValueBV64 n v t"
| "substValueBV64 n v (AssumeReg r v' ::: t) =
   AssumeReg r (substValue n v v') ::: substValueBV64 n v t"
| "substValueBV64 n v (DefineConstBV64 n' v' ::: t) =
   DefineConstBV64 n' (substValue n v v') ::: substValueBV64 n v t"
| "substValueBV64 n v (DefineConstBoolFromBV64 n' v' ::: t) =
   DefineConstBoolFromBV64 n' (substValue n v v') ::: substValueBV64 n v t"
| "substValueBV64 n v (e ::: t) = e ::: substValueBV64 n v t"
| "substValueBV64 n v (Cases cs) = Cases (map (substValueBV64 n v) cs)"
```

This next function performs the same, but for Booleans instead.

```
fun substValueBool :: "name ⇒ bool val ⇒ trace ⇒ trace" where
  "substValueBool - - EmptyTrace = EmptyTrace"
| "substValueBool n v (Assert v' ::: t) =
   Assert (substValue n v v') ::: substValueBool n v t"
| "substValueBool n v (DefineConstBool n' v' ::: t) =
   DefineConstBool n' (substValue n v v') ::: substValueBool n v t"
| "substValueBool n v (e ::: t) = e ::: substValueBool n v t"
| "substValueBool n v (Cases cs) = Cases (map (substValueBool n v) cs)"
```

A.3.2 Big-step semantics

Now, we define the big-step deterministic semantics for Isla traces, which takes a tuple of a trace and a machine state (t, st) and "executes" the entirety of t. If it successfully terminates, it returns the resulting state *Some st′*, and returns *None* if the execution failed.

We assume that all traces being processed are "valid", meaning that *Cases* will only have exactly one trace that terminates successfully, and that all types are consistent, and that all values are instantiated by prior substitutions before they are used.

Thanks to the assumption on *Cases*, we can evaluate it by simply evaluating each branch until we find one that terminates successfully, then use that branch's result.

```
function step :: "trace * state ⇒ state option" where
  "step (ReadReg r n ::: t, (stR, stI, stM))
     = (case stR r of Some x ⇒ step (substValueBV64 n x t, (stR, stI, stM)) | - ⇒ None)"
| "step (WriteReg r v ::: t, (stR, stI, stM))
     = step (t, (stR(r ↦ v), stI, stM))"
| "step (DefineConstBV64 n v ::: t, st)
     = step (substValueBV64 n v t, st)"
| "step (DefineConstBool n v ::: t, st)
     = step (substValueBool n v t, st)"
| "step (DefineConstBoolFromBV64 n (Val bv) ::: t, st)
     = step (substValueBool n (Val (if bv = 0 then True else False)) t, st)"
| "step (DefineConstBoolFromBV64 n (Name n') ::: t, st) = None"
| "step (DefineConstBoolFromBV64 n (Monop op v) ::: t, st) = None"
| "step (DefineConstBoolFromBV64 n (Binop op v1 v2) ::: t, st) = None"
| "step (Assert (Val True) ::: t, st) = step (t, st)"
| "step (Assert (Val False) ::: t, st) = None"
| "step (Assert (Name n) ::: t, st) = None"
| "step (Assert (Monop op v) ::: t, st) = None"
| "step (Assert (Binop op v1 v2) ::: t, st ) = None"
| "step (AssumeReg r (Val v) ::: t, (stR, stI, stM))
     = (case stR r of
          Some (Val v') ⇒ if v = v' then step (t, (stR, stI, stM)) else None
        | - ⇒ None)"
| "step (AssumeReg r (Name n) ::: t, st) = None"
| "step (AssumeReg r (Monop op v) ::: t, st) = None"
| "step (AssumeReg r (Binop op v1 v2) ::: t, st) = None"
| "step (Cases (c#cs), st)
     = (case step (c, st) of None ⇒ step (Cases cs, st) | res ⇒ res)"
| "step (Cases [], st) = None"
| "step (EmptyTrace, st) = Some st"
  — Memory semantics not yet implemented.
| "step (ReadMem - - - ::: -, -) = None"
| "step (WriteMem - - - ::: -, -) = None"

 by pat-completeness auto — Proves totality of this function (other than termination).
```

A.3.3 Totality and termination of semantics

Finally, in this section, we prove that *step* is a total terminating function. We begin by defining a measure on trace sizes, that should be strictly decreased with each recursive call of *step*.

```
fun measureTrace :: "trace ⇒ nat" where
  "measureTrace EmptyTrace = 0"
| "measureTrace (e ::: t) = 1 + measureTrace t"
| "measureTrace (Cases []) = 1"
| "measureTrace (Cases (c#cs)) = 1 + measureTrace c + measureTrace (Cases cs)"
```

We then prove the following lemmas that say that the functions *substValueBool* and *substValueBV64* do not change the measure of the trace when performing the substitution. In other words, we prove that measureTrace($t[v/n]$) = measureTrace(t).

```
lemma substMeasureBoolEqualCons [simp]:
    "measureTrace (substValueBool n v (e ::: t))
       = 1 + measureTrace (substValueBool n v t)"
  by (induction e) auto

lemma substMeasureBoolEqualCases [simp]:
    "(⋀c. c ∈ set cs ⟶ measureTrace (substValueBool n v c) = measureTrace c)
     ⟹ measureTrace (Cases (map (substValueBool n v) cs)) = measureTrace (Cases cs)"
  by (induction cs) auto

lemma substMeasureBoolEqual [simp]:
    "measureTrace (substValueBool n v t) = measureTrace t"
  by induction auto

lemma substMeasureBV64EqualCons [simp]:
    "measureTrace (substValueBV64 n v (e ::: t))
       = 1 + measureTrace (substValueBV64 n v t)"
  by (induction e) auto

lemma substMeasureBV64EqualCases [simp]:
    "(⋀c. c ∈ set cs ⟶ measureTrace (substValueBV64 n v c) = measureTrace c)
     ⟹ measureTrace (Cases (map (substValueBV64 n v) cs)) = measureTrace (Cases cs)"
  by (induction cs) auto

lemma substMeasureBV64Equal [simp]: "measureTrace (substValueBV64 n v t) = measure-
Trace t"
  by induction auto
```

With these lemmas proved, we can finally use *measureTrace* as the measure function to prove termination of *step*.

termination *step*
  by (*relation "measure* (λ(t, -). *measureTrace* t)") *auto*

end

A.4 Inter-trace semantics in `Multitrace.thy`

```
theory Multitrace
  imports Main Semantics "HOL−Eisbach.Eisbach"
begin
```

This theory is now concerned with defining the inter-trace semantics to allow proofs about programs with more than 1 instruction.

A.4.1 Defining $\longrightarrow$

We begin by lifting the machine state datatype *state* to an extended state datatype that supports failure (*Err*) and successful termination (*Ok state*). Successful termination is defined as finishing a trace with the PC pointing a no further trace. The *Running st* case is the final case where execution is still ongoing.

```
datatype multitraceState = Running state | Err | Ok state
```

We can then define the total function that takes a *state* and runs the trace pointed to by the PC, returning a *multitraceState* with several cases. If the PC points at no further trace, it successfully terminates with *Ok*. If the PC is undefined, it fails with *Err*. If the trace execution by *step* fails, it also fails with *Err*. Lastly, if *step* succeeds and returns a new state *st′*, it returns *Running st′*.

```
fun runInstr :: "state ⇒ multitraceState" where
  "runInstr (stR, stI, stM) = (case stR ''PC'' of None ⇒ Err |
     Some (Val pc) ⇒ (case stI pc of None ⇒ Ok (stR, stI, stM) |
        Some t ⇒ (case step (t, (stR, stI, stM)) of None ⇒ Err |
           Some st′ ⇒ Running st′)))"
declare runInstr.cases[simp]
```

The inter-trace semantics are then given inductively as *evolve*, or $\longrightarrow$, which takes *multitraceState*s that are *Running* to the next *multitraceState*. This is an inductive relation and not a total function as it does not take *Ok* or *Err* to any other state.

```
inductive evolve :: "multitraceState ⇒ multitraceState ⇒ bool" (infix "⟶" 55) where
```

```
"runInstr st = mtst ⟹ Running st ⟶ mtst"
```

We can then easily prove that $\longrightarrow$ is deterministic. In other words, it is a right-unique relation.

```
theorem evolveDeterm: "⟦ mtst ⟶ mtst'; mtst ⟶ mtst'' ⟧ ⟹ mtst' = mtst''"
  using evolve.simps by fastforce
```

A.4.2 Multiple steps of $\longrightarrow$

We can then define $\longrightarrow|n|$ as performing $\longrightarrow$ n times.

```
inductive evolveN :: "multitraceState ⇒ nat ⇒ multitraceState ⇒ bool"
   ("- ⟶|-| -" [100, 55, 100] 55) where
  evolveN0Def: "mtst ⟶|0| mtst"
| evolveNStepRight: "⟦ mtst ⟶|n| mtst'; mtst' ⟶ st'' ⟧ ⟹ mtst ⟶|n+1| st''"
```

The following trivial lemmas can then be shown where $n = 0$ or 1.

```
lemma evolveN0 [simp]: "mtst ⟶|0| mtst' ⟷ mtst = mtst'"
  using evolveN.cases evolveN.intros(1) by auto

lemma evolveN1 [simp]: "mtst ⟶|1| mtst' ⟷ mtst ⟶ mtst'"
  using evolveN.cases evolveN.intros(2) by fastforce
```

Since our inductive definition was given by performing each additional $\longrightarrow$ on the right, we now show that it's equivalent to if we defined it as additional $\longrightarrow$s on the left.

```
lemma evolveNStepLeft [intro]:
   "⟦ mtst ⟶ mtst'; mtst' ⟶|n| st'' ⟧ ⟹ mtst ⟶|n+1| st''"
proof (induction n arbitrary: mtst mtst' st'')
  case 0 then show ?case using evolveN0 evolveN1 by auto
next
  case (Suc n)
  obtain mtst3 where "mtst' ⟶|n| mtst3 ∧ mtst3 ⟶ st''"
    by (metis Suc.prems(2) Suc-neq-Zero add-diff-cancel-right' diff-Suc-1 evolveN.cases)
  moreover hence "mtst ⟶|n+1| mtst3"
    using Suc.IH Suc.prems(1) by blast
  then show ?case using Suc-eq-plus1 calculation evolveN.intros(2) by presburger
qed
```

This also means that we can unpack $\longrightarrow$s from the left of $\longrightarrow|n|$ when $n > 0$.

```
lemma evolveNUnstepLeft:
  "mtst ⟶|Suc n| mtst' ⟹ ∃ st''. (mtst ⟶ st'' ∧ st'' ⟶|n| mtst')"
proof (induction n arbitrary: mtst mtst')
  case 0 then show ?case using evolveN0 evolveN1 by auto
next
  case (Suc n)
  then show ?case by (metis Suc-eq-plus1 Zero-not-Suc add-right-imp-eq evolveN.simps)
qed
```

Multiple ⟶|n|s can also be chained together, or split apart.

```
lemma evolveNChain:
  "⟦ mtst ⟶|m| mtst'; mtst' ⟶|n| st'' ⟧ ⟹ mtst ⟶|m+n| st''"
proof (induction n arbitrary: m mtst mtst' st'')
  case 0
  then have "mtst' = st''" by simp
  then show ?case using "0.prems" by force
next
  case (Suc n)
  obtain mtst3 where "mtst' ⟶ mtst3"
    using Suc.prems(2) evolveNUnstepLeft by blast
  then show ?case
    by (metis Suc.IH Suc.prems(1) Suc.prems(2) Suc-eq-plus1 add-Suc-shift
        evolveNStepRight evolveNUnstepLeft)
qed

lemma evolveNSplit:
  "mtst ⟶|n| mtst' ⟹ ∀ m ≤ n. ∃ st''. mtst ⟶|m| st'' ∧ st'' ⟶|n−m| mtst'"
proof (induction n arbitrary: mtst mtst')
  case 0 then show ?case by simp
next
  case (Suc n)
  show ?case (is "∀ m. ?P(m)")
  proof
    fix m
    show "?P(m)"
    proof (induction m)
      case 0 then show ?case by (simp add: Suc.prems)
    next
      case (Suc m) then show ?case
        by (metis Suc-diff-le Suc-eq-plus1 Suc-le-mono diff-Suc-Suc
            evolveNStepRight evolveNUnstepLeft le-SucI)
    qed
  qed
```

qed

For a fixed n, we can show that $\longrightarrow|n|$ is deterministic/right-unique just like $\longrightarrow$ is.

```
lemma evolveNDeterm: "⟦ mtst ⟶|n| mtst'; mtst ⟶|n| mtst'' ⟧ ⟹ mtst' = mtst''"
  apply (induction n arbitrary: mtst mtst' mtst'')
   apply simp
  apply (metis evolveDeterm evolveNUnstepLeft)
  done
```

A.4.3 Reflexive transitive closure $\longrightarrow^*$

We can then define the reflexive transitive closure $\longrightarrow^*$ in terms of $\longrightarrow|n|$.

```
abbreviation evolveStar :: "multitraceState ⇒ multitraceState ⇒ bool"
    (infix "⟶*" 55) where
  "mtst ⟶* mtst' ≡ ∃ n. mtst ⟶|n| mtst'"
```

For terminated *multitraceState*s, the only *multitraceState* they can go to is the very same state.

```
lemma noEvolveOk: "Ok st ⟶* mtst ⟹ mtst = Ok st"
 by (metis evolve.cases evolveN.cases evolveN1 evolveNSplit le-add2 multitraceState.simps(6))

lemma noEvolveErr: "Err ⟶* mtst ⟹ mtst = Err"
 by (metis evolve.cases evolveN.cases evolveN1 evolveNSplit le-add2 multitraceState.simps(4))
```

Now, we show that if a *multitraceState* evolves into successful termination, this is necessarily done with a fixed number of steps, and the terminating state it reaches is deterministic.

```
lemma evolveNFixedOk: "⟦ mtst ⟶|m| Ok st; mtst ⟶|n| Ok st' ⟧ ⟹ m = n ∧ st
= st'"
proof(rule conjI)
  assume assms: "mtst ⟶|m| Ok st"
                "mtst ⟶|n| Ok st'"
  show m-eq-n: "m = n"
  proof (rule ccontr)
    assume contrAssm: "m ≠ n"
    then show False
    proof cases
      assume lt: "m < n"
      then obtain mtst'' where mtst'': "mtst ⟶|m| mtst'' ∧ mtst'' ⟶|n−m| Ok st'"
```

```
      using evolveNSplit assms(2) by fastforce
    then have "mtst'' = Ok st" using assms(1) evolveNDeterm by blast
    then have "¬(mtst'' ⟶|n−m| Ok st')"
      by (metis lt diff-is-0-eq evolve.cases evolveN.cases linorder-not-less
          multitraceState.simps(6) noEvolveOk)
    then show ?thesis using mtst'' by blast
  next
    assume "¬(m < n)"
    then have gt: "m > n" using contrAssm by auto
    then obtain mtst'' where mtst'': "mtst ⟶|n| mtst'' ∧ mtst'' ⟶|m−n| Ok st"
      using evolveNSplit assms(1) by fastforce
    then have "mtst'' = Ok st'" using assms(2) evolveNDeterm by blast
    then have "¬(mtst'' ⟶|m−n| Ok st)"
      by (metis gt diff-is-0-eq evolve.cases evolveN.cases linorder-not-less
          multitraceState.simps(5) noEvolveOk)
    then show ?thesis using mtst'' by blast
  qed
 qed
 show "st = st'" using assms m-eq-n evolveNDeterm by blast
qed
```

We can then show that $\longrightarrow^*$ is deterministic/right-unique as long as it reaches a terminating state.

```
lemma evolveStarDetermOk: "⟦ mtst ⟶* Ok st; mtst ⟶* Ok st' ⟧ ⟹ st = st'"
 using evolveNFixedOk by blast

lemma evolveStarOkNoErr: "mtst ⟶* Ok st ⟹ ¬(mtst ⟶* Err)"
proof
 assume assms: "mtst ⟶* Ok st"
               "mtst ⟶* Err"
 obtain n where evolveNErr: "mtst ⟶|n| Err" using assms(2) by blast
 then have "∀ m < n. ¬(mtst ⟶|m| Ok st)"
  by (metis evolveNDeterm evolveNSplit less-or-eq-imp-le
      multitraceState.distinct(5) noEvolveOk)
 moreover then have "∀ m ≥ n. mtst ⟶|m| mtst' ⟹ mtst' = Err"
  using evolveNDeterm evolveNErr by auto
 then have "∀ m. ¬(mtst ⟶|m| Ok st)"
  by (metis calculation evolveNDeterm evolveNErr evolveNSplit
      linorder-not-le multitraceState.distinct(5) noEvolveErr)
 then have "¬(mtst ⟶* Ok st)" by auto
 then show False using assms(1) by blast
qed
```

```
lemma evolveStarErrNoOk: "mtst ⟶* Err ⟹ ¬(mtst ⟶* Ok st)"
  using evolveStarOkNoErr by blast

theorem evolveStarDeterm:
    "⟦ mtst ⟶* mtst; mtst ⟶* mtst'
    ; ∀st. mtst ≠ Running st; ∀st. mtst' ≠ Running st
    ⟧ ⟹ mtst = mtst'"
  by (metis multitraceState.exhaust noEvolveOk noEvolveErr)
```

As a sanity check, we ensure that our definition of $\longrightarrow^*$ is indeed reflexive and transitive.

```
lemma evolveStarRefl [simp, intro]: "mtst ⟶* mtst"
  using evolveN0Def by blast

lemma evolveStarTrans [simp, intro]:
    "⟦ mtst1 ⟶* mtst2; mtst2 ⟶* mtst3 ⟧ ⟹ mtst1 ⟶* mtst3"
  using evolveNChain by blast
```

Finally, we prove that it is the closure of $\longrightarrow$ by showing it can be achieved by adding $\longrightarrow$ on the left and right.

```
lemma evolveStarStepLeft [simp, intro]:
    "⟦ mtst1 ⟶ mtst2; mtst2 ⟶* mtst3 ⟧ ⟹ mtst1 ⟶* mtst3"
  by blast

lemma evolveStarStepRight:
    "⟦ mtst1 ⟶* mtst2; mtst2 ⟶ mtst3 ⟧ ⟹ mtst1 ⟶* mtst3"
  using evolveNStepRight by blast

end
```

A.5 Proof automation and methods in `Automation.thy`

theory *Automation*
  imports *Multitrace "HOL−Eisbach.Eisbach"*
begin

This theory provides the proof automation methods for our semantics.

These are common, reusable methods. The first performs a single step of $\longrightarrow$ on the left side of $\longrightarrow^*$, and the latter helps with proving statements of the form:

$$\exists\, st'.\ mtst \longrightarrow^* Ok\ st' \wedge P\ st'$$

method *step-evolve* = (*rule evolveStarStepLeft*, *rule evolve.intros*, *auto*)
method *prove-exists* = (*rule exI*, *auto*, *step-evolve+*)

We then prove the following lemma in order to achieve full proof automation for specific classes of proofs. This lemma says that, to prove that any terminating state reached from our current state fulfills some property P, it suffices to prove that there *exists* a state that can be reached from our current state, and that that state fulfills P. This is true due to $\longrightarrow^*$ being deterministic for terminating states.

lemma *evolveStarOkWitnessEnough*:
  $"(\exists\, st'.\ mtst \longrightarrow^* Ok\ st' \wedge P(st')) \Longrightarrow (mtst \longrightarrow^* Ok\ st' \longrightarrow P(st'))"$
  using *evolveStarDetermOk* **by** *blast*

We can then achieve proof automation for statements of the form:

$$[\![assms;\ mtst \longrightarrow^* Ok\ st']\!] \Longrightarrow P\ st'$$

by using the above lemma and proof methods.

method *proof-automation* =
  (*unfold atomize-imp*, *rule impI*, *rule evolveStarOkWitnessEnough*, *prove-exists*)

end

A.6 Examples in `Examples.thy`

```
theory Examples
  imports Automation
begin
```

This theory contains the example programs and proofs that were presented in the main body, illustrating the defined semantics, proof methods, and automation..

This trace corresponds to `add x1, x1, x2`.

```
abbreviation addAABTrace :: trace where
  "addAABTrace ≡
      ReadReg ''x1'' 1
    ::: ReadReg ''x2'' 2
    ::: DefineConstBV64 3 (Binop (+) (Name 1) (Name 2))
    ::: ReadReg ''PC'' 4
    ::: DefineConstBV64 5 (Binop (+) (Name 4) (Val 64))
    ::: WriteReg ''x1'' (Name 3)
    ::: WriteReg ''PC'' (Name 5)
    ::: EmptyTrace"
```

We can show the effect of *step* on the above trace using the following command

```
value "step (addAABTrace, (Map.empty
  ( ''x1'' ↦ Val x1
  , ''x2'' ↦ Val x2
  , ''PC'' ↦ Val pc
  ), Map.empty, Map.empty))"
```

which gives us the value

Some (λu. *if* u = ''PC'' *then Some* (*Val* (*pc* + 64)) *else if* u = ''x1'' *then Some* (*Val* (*x*1 + *x*2)) *else if* u = ''PC'' *then Some* (*Val pc*) *else if* u = ''x2'' *then Some* (*Val x*2) *else if* u = ''x1'' *then Some* (*Val x*1) *else None*, *Map.empty*, *Map.empty*)

showing it has correctly updated `x1` to `x1` + `x2`, and incremented the PC by 64.

Since *step* is total, Isabelle has no difficulty proving that `x1` is set correctly by *addAAB-Trace*, which is specified by the following theorem.

```
theorem
  "⟦ stR ''x1'' = Some (Val x1); stR ''x2'' = Some (Val x2); stR ''PC'' = Some (Val pc)
  ; step (addAABTrace, (stR, stI, stM)) = Some (stR', stI', stM')
  ⟧ ⟹ stR' ''x1'' = Some (Val (x1 + x2))" by auto
```

Writing (effectively) the same proof, but involving inter-trace semantics is more involved. The theorem statement now uses $\longrightarrow^*$, and makes the statement about any state *st'* that is reached with successful termination from the initial state.

```
theorem
     "stR ''x1'' = Some (Val x1) ∧ stR ''x2'' = Some (Val x2) ∧ stR ''PC'' = Some (Val 64)
        ∧ stI 64 = Some addAABTrace ∧ stI 128 = None
  ⟹ Running (stR, stI, stM) ⟶* Ok st'
  ⟹ (fst st') ''x1'' = Some (Val (x1 + x2))"
     (is "?assms ⟹ ?mtst ⟶* Ok ?st' ⟹ ?Q(?st')")
proof -
  let "?P(st')" = "?mtst ⟶* Ok st'"

  — First, we find an existential witness of such a terminating state.
  have "?assms ⟹ ∃ st'. ?P(st') ∧ ?Q(st')"
    by (rule exI, auto, (rule evolveStarStepLeft, rule evolve.intros, auto)+)

  — We can then generalize to all possible terminating states using the determinism
  — of ⟶* on terminating states.
  then have "?assms ⟹ ∀ st'. ?P(st') ⟶ ?Q(st')" using evolveStarDetermOk by blast

  — Lastly, we can prove the theorem statement.
  then show "?assms ⟹ ?P(?st') ⟹ ?Q(?st')" by blast
qed
```

This is another trace, which corresponds to `mov x1, x2`.

```
abbreviation movABTrace :: trace where
  "movABTrace ≡
      ReadReg ''x1'' 1
   ::: WriteReg ''x2'' (Name 1)
   ::: ReadReg ''PC'' 2
   ::: DefineConstBV64 3 (Binop (+) (Name 2) (Val 64))
   ::: WriteReg ''PC'' (Name 3)
   ::: EmptyTrace"
```

We can then write a theorem about a two-instruction program involving the two traces we've defined. This theorem specifies exactly what the final state should be, so a simple apply script is sufficient to prove its correctness.

```
theorem
    "stR ''x1'' = Some (Val x1) ∧ stR ''x2'' = Some (Val x2) ∧ stR ''PC'' = Some (Val 128)
      ∧ stI 128 = Some addAABTrace ∧ stI 192 = Some movABTrace
      ∧ stI 256 = Some addAABTrace ∧ stI 320 = None
  ⟹ Running (stR, stI, stM) ⟶* Ok (stR(
          ''x1'' ↦ Val (2 ∗ x1 + 2 ∗ x2),
          ''x2'' ↦ Val (x1 + x2),
          ''PC'' ↦ Val 320), stI, stM)"
  apply (auto)
  apply (rule evolveStarStepLeft, rule evolve.intros, auto)+
  apply (simp add: fun-upd-twist)
  done
```

As before with the single-instruction proof, we can use the same proof strategy to prove a more general statement about terminating states reached without necessarily specifying the entire resulting state. This theorem statement asserts that `x1` and `x2` are set correctly, without making any statements about the rest of the state.

```
theorem
    "stR ''x1'' = Some (Val x1) ∧ stR ''x2'' = Some (Val x2) ∧ stR ''PC'' = Some (Val 128)
      ∧ stI 128 = Some addAABTrace ∧ stI 192 = Some movABTrace
      ∧ stI 256 = Some addAABTrace ∧ stI 320 = None
  ⟹ Running (stR, stI, stM) ⟶* Ok st'
  ⟹ (fst st') ''x1'' = Some (Val (2 ∗ x1 + 2 ∗ x2))
      ∧ (fst st') ''x2'' = Some (Val (x1 + x2))"
   (is "?assms ⟹ ?mtst ⟶* Ok ?st' ⟹ ?Q(?st')")
```

proof −
  let "?P(st′)" = "?mtst $\longrightarrow^*$ Ok st′"
  have "?assms $\Longrightarrow$ $\exists$ st′. ?P(st′) $\wedge$?Q(st′)"
    by (rule exI, auto, (rule evolveStarStepLeft, rule evolve.intros, auto)+)
  then have "?assms $\Longrightarrow$ $\forall$ st′. ?P(st′) $\longrightarrow$?Q(st′)" **using** evolveStarDetermOk **by** blast
  then show "?assms $\Longrightarrow$?P(?st′) $\Longrightarrow$?Q(?st′)" **by** blast
qed

This shared proof method applies to a wide variety of statements, as long as they are of the same form. This form is in fact the one we defined earlier in *Automation*, so we can apply *proof-automation* to perform a push-button automated proof of the same statement.

theorem
    "stR ″x1″ = Some (Val x1) $\wedge$ stR ″x2″ = Some (Val x2) $\wedge$ stR ″PC″ = Some (Val 128)
      $\wedge$ stI 128 = Some addAABTrace $\wedge$ stI 192 = Some movABTrace
      $\wedge$ stI 256 = Some addAABTrace $\wedge$ stI 320 = None
  $\Longrightarrow$ Running (stR, stI, stM) $\longrightarrow^*$ Ok st′
  $\Longrightarrow$ (fst st′) ″x1″ = Some (Val (2 $*$ x1 + 2 $*$ x2))
      $\wedge$ (fst st′) ″x2″ = Some (Val (x1 + x2))"
  by proof-automation

Our next example deals with the multiple instruction program including branching that corresponds to the following program:

```
  addi x2, x1, 0
  addi x3, x1, -1
loop:
  beq x0, x3, end
  add x1, x1, x2
  addi x3, x3, -1
  beq x0, x0, loop
end:
```

It squares x1 assuming that x2 and x3 are initially zeroed.

We begin by defining general traces for addi, beq and add instructions.

```
abbreviation addiTrace :: "reg ⇒ reg ⇒ bv64 ⇒ trace" where
  "addiTrace rd rs imm ≡
      ReadReg rd 0
    ::: ReadReg rs 1
    ::: DefineConstBV64 2 (Val imm)
    ::: DefineConstBV64 3 (Binop (+) (Name 1) (Name 2))
    ::: WriteReg rd (Name 3)
    ::: ReadReg ''PC'' 4
    ::: DefineConstBV64 5 (Binop (+) (Name 4) (Val 64))
    ::: WriteReg ''PC'' (Name 5)
    ::: EmptyTrace"

abbreviation beqTrace :: "reg ⇒ reg ⇒ bv64 ⇒ trace" where
  "beqTrace rs1 rs2 addr ≡
      ReadReg rs1 0
    ::: ReadReg rs2 1
    ::: DefineConstBV64 2 (Binop (λx y. (if x = y then 0 else 1)) (Name 0) (Name 1))
    ::: DefineConstBoolFromBV64 3 (Name 2)
    ::: Cases [ Assert (Name 3)
                ::: WriteReg ''PC'' (Val addr)
                ::: EmptyTrace
              , DefineConstBool 4 (Monop (λx. ¬x) (Name 3))
                ::: Assert (Name 4)
                ::: ReadReg ''PC'' 5
                ::: DefineConstBV64 6 (Binop (+) (Name 5) (Val 64))
                ::: WriteReg ''PC'' (Name 6)
                ::: EmptyTrace
```

```
]"

abbreviation addTrace :: "reg ⇒ reg ⇒ reg ⇒ trace" where
  "addTrace rd rs1 rs2 ≡
       ReadReg rd 0
    ::: ReadReg rs1 1
    ::: ReadReg rs2 2
    ::: DefineConstBV64 3 (Binop (+) (Name 1) (Name 2))
    ::: WriteReg rd (Name 3)
    ::: ReadReg ''PC'' 4
    ::: DefineConstBV64 5 (Binop (+) (Name 4) (Val 64))
    ::: WriteReg ''PC'' (Name 5)
    ::: EmptyTrace"
```

We can then define the instruction store *instrs* that corresponds to the program beginning at address 0, noting the addresses carefully written to correspond to the labels.

```
abbreviation programInstructions :: instrs where
  "programInstructions ≡ Map.empty
   (   0 ↦ addiTrace ''x2'' ''x1'' 0
   ,  64 ↦ addiTrace ''x3'' ''x1'' (−1)
   , 128 ↦ beqTrace ''x0'' ''x3'' 384
   , 192 ↦ addTrace ''x1'' ''x1'' ''x2''
   , 256 ↦ addiTrace ''x3'' ''x3'' (−1)
   , 320 ↦ beqTrace ''x0'' ''x0'' 128
   )"
```

We then define the initial register values with x1 taking an arbitrary value, the PC starting at address 0, and x2 and x3 set to 0. Of course, x0 is also set to 0, as it is permanently zero on RISC-V.

```
abbreviation regInits :: "bv64 val ⇒ regState" where
  "regInits x1 ≡ Map.empty
   ( ''x0'' ↦ Val 0
   , ''x1'' ↦ x1
   , ''x2'' ↦ Val 0
   , ''x3'' ↦ Val 0
   , ''PC'' ↦ Val 0
   )"
```

Our proof automation is then enough to prove this program is correct for specific initial values of x1—i.e. we show that it terminates and squares x1.

abbreviation *initState* :: "*bv64 val* ⇒ *state*" **where**
"*initState x1* ≡ (*regInits x1*, *programInstructions*, *Map.empty*)"
theorem
"$x1 = Val\ 3$
$\Longrightarrow Running\ (initState\ x1) \longrightarrow^{*} Ok\ st'$
$\Longrightarrow (fst\ st')\ ''x1'' = Some\ (Val\ 9)$"
by *proof-automation*

end