Practical Verification of Safety-Critical Systems

Stuart Pernsteiner

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2018

Reading Committee:

Zachary Tatlock, Chair

Michael D. Ernst, Chair

Maya Cakmak

Program Authorized to Offer Degree:
Computer Science

University of Washington

Abstract

Practical Verification of Safety-Critical Systems

Stuart Pernsteiner

Co-Chairs of the Supervisory Committee:
Assistant Professor Zachary Tatlock
Department of Computer Science

Professor Michael D. Ernst
Department of Computer Science

Software-based control systems operate scientific equipment worth millions of dollars and even safety-critical medical devices, making them good targets for strong formal verification techniques. However, these systems are rarely verified in practice. We identify three key challenges hindering the application of verification to real-world control systems and present solutions to each.

First, safety properties of control systems often rely on correct operation and interaction of several heterogeneous hardware and software components. No single analysis tool can reason about all types of components. We present techniques, based on the established practice of safety case construction, for building a machine-checkable safety case that combines concrete evidence about the system implementation derived from multiple analysis tools. Using these techniques, we uncovered safety-critical flaws in a prerelease version of control software for the Clinical Neutron Therapy System (CNTS), a radiotherapy installation.

Second, software components of control systems are often developed using proprietary or domain-specific languages for which no formal semantics yet exist. We present a methodology for rapidly developing language semantics, allowing application of formal verification techniques in languages that have received little previous study. We used this methodology

to develop semantics for Python and for the EPICS dataflow language, suitable for analyzing components of the CNTS control software.

Third, for control system software written in specialized languages, often no verified language implementations are available. We present a new technique for developing verified compilers that combines a verified denotation function with a verified extraction procedure to achieve high run-time performance with low verification effort. We demonstrate the effectiveness of this technique by developing a verified compiler for a fragment of the EPICS dataflow language and using it to compile portions of the CNTS control software.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

First, many thanks to my wonderful and patient advisors Zachary Tatlock and Michael Ernst for their encouragement and guidance throughout my years in graduate school. Thanks also to my collaborators Calvin Loncaric, Eric Mullen, James Wilcox, Emina Torlak, Dan Grossman, and Jon Jacky, without whom the work presented here would never have been possible. And finally, thanks to everyone from the PLSE group for keeping things interesting.

Chapter 1

INTRODUCTION

Control systems built with proprietary and domain-specific languages operate scientific equipment worth millions of dollars and even safety-critical medical devices, making them good targets for the strongest formal verification techniques. However, these systems are rarely verified in practice. In this work, we identify three key challenges hindering the application of verification to real-world control systems and present solutions to each.

First, critical safety properties of control systems can involve several types of hardware components and multiple pieces of software written in different languages. No single tool can analyze such a diverse set of system components. Combining results of multiple independent analyses is essential, but in prior approaches, such as safety case construction, the high-level safety argument takes the form of a natural-language document, not suited for automated checking or formal reasoning.

We propose the use of *machine-checkable safety cases*, consisting of an abstract system model linked to concrete evidence about the behavior of individual components. The system model, developed in a formal and machine-checkable modeling language, captures the high-level safety argument, while the links to concrete evidence, such as results of automated program analyses, ensure that the model is not grounded in faulty assumptions about component behavior. The resulting safety case can be checked automatically not only for logical consistency of the safety argument but also for correspondence with the system implementation it is intended to model.

In Chapter 2, we describe machine-checkable safety case construction and its application to the Clinical Neutron Therapy System, a safety-critical radiation therapy installation.

Second, software components of safety-critical control systems may be written in pro-

prietary or domain-specific languages for which no formal (or even informal) specification exists. Rewriting the control system in a language more amenable to formal analysis is typically not an option—aside from the usual cost of reimplementing a substantial piece of software, safety-critical control systems are further burdened with extensive testing and other validation requirements necessary to ensure safe operation. But applying formal verification and reasoning to software in previously unstudied languages first requires the development of formal semantics for the implementation language, which itself is often a complex and time-consuming affair.

We present a methodology, which we call *marquer*, for rapidly developing language semantics that are narrowly focused on a specific program and property. The marquer methodology results in semantics that match the behavior of a reference language implementation "bug-for-bug," allowing reasoning about the behavior of a program under the language implementation actually used in the production system. Focusing the semantics on a concrete program and property of interest allows the verification engineer to avoid modeling complex language features that are irrelevant to the verification task at hand. This reduces the cost of semantics development, making it feasible to apply formal verification even in languages for which no semantics currently exists.

In Chapter 3, we describe the marquer methodology in detail, along with its application to safety-critical software components in CNTS.

Third, the relative obscurity of domain-specific languages for control systems means that suitable verified language implementations are rarely available. And prior techniques for building verified implementations present an unfortunate dilemma. Verified interpreters are relatively easy to construct, but mandate use of a complex, trusted runtime system; verified compilers, meanwhile, can avoid the need for complex runtimes, but their development is extraordinarily labor-intensive.

We present a new approach to constructing verified compilers that combines the ease of development and verification of an interpreter and the performance and low TCB of a compiler. A compiler built with our approach, which we call a *denotational compiler*, operates

in two phases. First, within the context of an automated theorem prover, a verified denotation function transforms a deeply-embedded AST of the source language into a function in the theorem prover's internal language. This denotation has a type such as `program` $\rightarrow$ `state` $\rightarrow$ `state`, so evaluating `denote p` (of type `state` $\rightarrow$ `state`) produces a pure function whose behavior is equivalent to executing a step of input program `p`. Second, a verified extraction procedure compiles the denotation of the input program from the prover's internal language to executable code. Combining the verified denotation with verified extraction produces a verified compilation pipeline from source programs to executables.

In Chapter 4, we elaborate on the design of denotational compilers, and we present Œuf, a verified Gallina compiler designed to be well suited for use as the verified extraction portion of a denotational compiler.

Chapter 2

AUTOMATED SAFETY CASE CHECKING

2.1 Introduction

Formal techniques for guaranteeing software correctness have made tremendous progress in recent decades. However, applying these techniques to real-world safety-critical systems remains challenging for three reasons. First, using general-purpose tools to formally prove deep properties of a system component (e.g., functional correctness of a cryptographic primitive [15]) requires substantial expertise and manual effort. Second, many real systems contain components for which effective formal analysis is still an active research topic (e.g., formally guaranteeing liveness for a consensus protocol within a distributed system [57]), and thus in practice, these components can only be analyzed by weaker techniques such as testing or expert review. Third, even when deep properties can be established for individual system components, their composition may not add up to overall system safety, leading to catastrophic failures [63].

This chapter reports on a large-scale case study in applying modern verification techniques to check the safety of a radiotherapy system in current clinical use: the Clinical Neutron Therapy System (CNTS) at the University of Washington Medical Center. We describe how to practically address the above challenges with a combination of techniques that reason at the system and component levels, and that provide guarantees of varying strength, from automatic proof to manual review. To check system-level properties (such as "the beam shuts off if physical settings of the machine do not match the prescription"), we developed and analyzed a formal model of CNTS in Alloy [64, 1]. Using this model, we obtained partial specifications of critical component-level properties (such as "the therapy control software sends a shut-down message if it receives an out-of-tolerance sensor reading"). To check the

resulting component properties, we built a suite of custom tools, ranging from an SMT-based verifier for properties of the control software to a manual review processor for properties of the physical components. These custom tools plug into our *safety case checker* (SCC), which combines the system model with the results of the component checks into a mechanized argument—a form of a *safety case* [66, 127]—that CNTS satisfies its critical safety properties.

To the best of our knowledge, this case study presents the *first formal, machine-checked safety case for a real system*. A typical safety case takes the form of a structured argument, expressed in natural language or graphical notation [38, 78], that a system satisfies a desired critical property. This argument decomposes the system property into a set of component properties, each of which is justified with concrete *evidence* (e.g., the results of verification, manual review, or testing). As massive informal artifacts, however, typical safety arguments suffer from logical fallacies [52] and are difficult to audit for the presence and sufficiency of evidence. For these reasons, prior work [113, 112] has called for mechanization of safety case checking. Our work shows, for the first time, that such mechanization is not only possible, but that it is both practical and useful. Building on existing verification frameworks [122, 1], our safety case (including the system model and the component checkers) consists of just 2700 lines of code, yet its checks uncovered safety-critical flaws in pre-release versions of the CNTS control software.

Our work differs from prior efforts [31] at safety case mechanization in that *the safety argument is embedded into a formal model of the system*. In prior work, properties of system components are abstracted by uninterpreted predicates, and a mechanized tool ensures that the logical structure of the argument is sound. With this approach, missing properties or assumptions about the environment are discovered through manual auditing. We take a different approach, which enables us to both check the logical soundness of the argument and mechanically discover (some kinds of) missing properties. In particular, our safety argument takes the form $S \Rightarrow P$, where S is a model of the system and P is the desired safety property. The model S specifies the behavior of the system in terms of the properties $s_1, \ldots, s_n$ of individual components and their interactions. When expressed in Alloy, such a

model enables both bounded simulation and checking of abstract executions of the system. Bounded simulation helps ensure that the model S includes all desirable treatment scenarios (thus guarding against vacuous fulfillment of the safety argument $S \Rightarrow P$). The checking, on the other hand, helps detect missing component properties (e.g., "the Ethernet network does not drop messages") that are necessary to establish the safety property P.

A component property s_i serves as a (partial) specification for the implementation of the component c_i in the underlying system. Ideally, each component c_i would be mechanically verified to satisfy s_i, but such verification is currently infeasible for many properties (e.g., the reliability of an Ethernet network). Our safety case therefore employs a pragmatic approach that allows weaker evidence to be used for these properties. In particular, each property s_i is guarded by an uninterpreted predicate, written as $\mathit{evidence}(\mathit{tool}, \mathit{args}) \Rightarrow s_i$, which states that the given external tool establishes the property s_i for the component c_i. When checking a safety case, SCC invokes the specified external tools to determine the truth of these predicates, but it does not reason about the strength of the evidence provided by the tool: the *sufficiency* of evidence is subject to manual auditing. Our safety case aids this manual part of the process by making explicit, and precise, all links from a safety argument to evidence.

The rest of this chapter is organized as follows. Section 2.2 provides an overview of the CNTS system. Section 2.3 describes our machine-checked safety case for CNTS. Section 2.4 presents SCC and other tools we built as part of the case study. Section 2.5 discusses the flaws that these tools helped us find and correct. Section 2.6 surveys related work, and Section 2.7 concludes the chapter.

2.2 Overview of CNTS

The Clinical Neutron Therapy System (CNTS) is a radiotherapy machine that uses neutron radiation to treat tumors resistant to conventional radiotherapy. Due to the high installation and maintenance costs of neutron-based systems, CNTS has been in service for over 30 years, and it is one of only three systems of this kind in the United States. As such, it depends heavily on custom software developed by CNTS engineers, who have achieved a remarkable

safety record, with no serious misadministrations due to machine or control system problems.

CNTS engineers have re-implemented its therapy control software twice since 1984 [68, 73]. The latest controller [69] is implemented in a subset of the EPICS (Experimental Physics and Industrial Control System) dataflow language [11], which is widely used for controlling scientific instruments. This development effort aims to support new therapies, integrate new software and hardware, and adapt to changes in the hospital's systems—with continued safety as the foremost concern.

CNTS as a whole is designed [72] to enforce a number of critical safety properties, including *prescription safety*, which is the focus of our work:

> **Prescription Safety Property (P_{rx}).** *During treatment, the beam will turn off if any physical machine setting moves outside the tolerances specified by the prescription and the operator has not issued the manual override command.*[1]

This property is enforced by a *control subsystem*, consisting of hardware and software components, that monitors and drives the system's physical components. Our safety case for P_{rx} (Section 2.3) spans the control subsystem of CNTS, as well as the physical components involved in a treatment prescription.

Physical Components. The key physical components of CNTS include the cyclotron, the leaf collimator, the gantry, and the treatment couch. The cyclotron generates a broad beam of particles, which passes through the leaf collimator in the gantry head on its way to the patient. The collimator consists of forty steel leaves that control the shape of the beam. The gantry head also contains a set of wedges and filters that can be inserted to further adjust the beam's shape and intensity. The gantry rotates 360 degrees around the treatment couch so that the beam can enter the patient from any angle. The couch itself has five degrees of motion freedom. A treatment prescription specifies settings for all of these components, and

[1] The manual override exists to enable treatment to continue if a sensor for a machine setting generates a false alarm.

the CNTS control subsystem ensures that each setting remains within prescribed tolerances during treatment.

Control Subsystem. The CNTS control subsystem is a collection of hardware and software components, which communicate by exchanging messages through a private Ethernet network. The components relevant to our safety case include:

Embedded Single-Board Computers interface with motors and sensors for controlling the physical components of the system. For example, the Treatment Motion Controller (TMC) monitors the orientation of the couch and the gantry. Separate computers exist for shaping the neutron beam and for monitoring the patient's radiation exposure. These embedded computers were installed by the original system vendor and are treated as black boxes by CNTS engineers.

Therapy Control (TC) Software displays the user interface on the operator console, accepts commands from the operator, coordinates the activity of the embedded computers, and helps enforce CNTS safety properties (such as P_{rx}). The TC, now re-implemented in EPICS, runs on a general-purpose Linux computer.

Programmable Logic Controller (PLC) serves as an interface between the networked components and the cyclotron control hardware. The PLC is electrically connected to the Hardwired Safety Interlock System (HSIS), which consists of a series of mechanical relays. These relays carry power to the radio frequency amplifiers that accelerate particles in the cyclotron. If any relay is opened, the cyclotron stops receiving power and the neutron beam shuts off. For example, if a machine setting moves out of the prescribed tolerances, the TC sends a signal to the PLC to open an HSIS relay, thus shutting off the beam. The PLC functionality is implemented using ladder logic [74], and the CNTS staff modifies and maintains this implementation in addition to the TC.

2.3 Building a Case for Prescription Safety

This section describes our approach to building a formal, machine-checkable safety case for a key property of CNTS—prescription safety (P_{rx}). The presentation focuses on the aspects of case design that enabled us to practically and effectively check a deep property of a complex system. The tools we developed, and the results we obtained, are discussed in the following sections. All artifacts comprising the case can be found on the project's web page [4].

2.3.1 Structuring the Case: A System Model with Pluggable Checkers

Our safety case for P_{rx} consists of two parts: (1) a system model S that specifies partial properties of the components spanned by P_{rx} (Section 2.2), and (2) a set of custom tools for checking if those components satisfy the specified properties. The tools indicate either success or failure, depending on whether a given property can be established (through some means) for a given component. Our safety case checker (SCC) connects the model and the tools through uninterpreted predicates of the form $\mathit{evidence}(\mathit{tool}, \mathit{args})$, which guard component properties specified by S (see, e.g., Figure 2.2). These predicates tell SCC which tools to invoke, and how, to establish a given property. The SCC collects pass/fail results of running the tools and uses an off-the-shelf counterexample finder [1] to check that, assuming the obtained results, $S \Rightarrow P_{rx}$. A safety case constructed in this way provides a high-level safety argument (i.e., the system model S, the system property P_{rx}, and the predicates describing how the evidence is obtained), which aids manual auditing, as well as a formal artifact, which enables automatic checking for (some classes of) safety-critical regressions as the system evolves.

2.3.2 Making the Case: Diagrams, Models, and Tools

We developed the safety case for P_{rx} in three steps. First, we worked with CNTS engineers to identify the system components and interactions that are relevant to P_{rx}, producing an informal, diagrammatic model of the system. This step was crucial for determining the level of

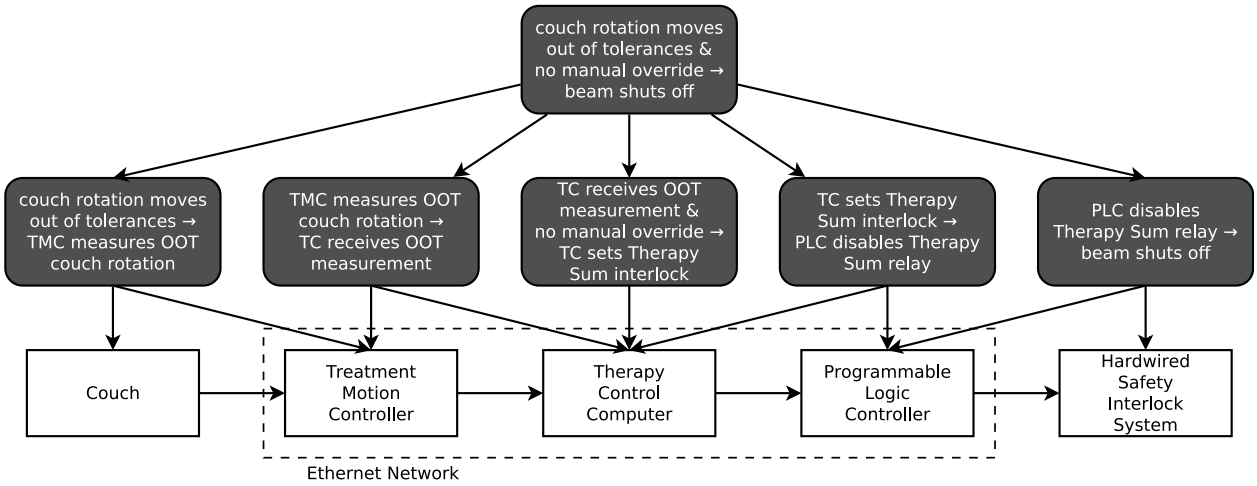

Figure 2.1: A fragment of the property-part diagram for a part of the P_{rx} property. Boxes represent system components, and rounded boxes represent properties. Edges originating from a property indicate that the property depends on the target components or properties for its fulfillment. Edges originating from a component indicate interaction via messages. Components within the dashed box are connected via Ethernet.

detail at which to model the system, and to understand the engineers' concerns—specifically, which properties mattered most and where to focus our analysis effort. Next, we formalized and refined the system model in Alloy [64, 1], a widely-used specification language that extends first-order logic with transitive closure and relational algebra. Throughout the formalization process, we relied heavily on Alloy's counterexample finder to detect errors in logical reasoning, as well as component properties that were missing from our informal diagrams. Finally, we used the resulting Alloy model—and feedback from CNTS engineers—to determine where and how to focus our tool-building effort. We describe each of these steps in more detail below.

Drafting an informal case. To build a safety case for P_{rx}, we first drafted an informal safety argument, expressed as a property-part diagram [65]. This kind of diagram shows how components—or parts—of the system relate to each other and how their individual

properties collectively satisfy the desired system property. Figure 2.1 shows a fragment of the property-part diagram for P_{rx} that covers the rotation angle of the treatment couch. The diagram shows how the top-level property and its sub-properties are established by a combination of other properties and components. For example, the Programmable Logic Controller (PLC) and the Hardwired Safety Interlock System (HSIS) jointly ensure that the beam is turned off when the Therapy Sum relay is opened (disabled). The informal case for P_{rx} includes similar diagrams for other machine parameters that are specified by a treatment prescription.

Formalizing the case. To formalize the P_{rx} case, we developed an Alloy model of the major components of CNTS. These include the therapy control (TC) software, the Treatment Motion Controller (TMC), the Programmable Logic Controller (PLC), the Hardwired Safety Interlock System (HSIS), the beam, and the machine settings involved in a prescription. Our model specifies the internal states of the components (e.g., whether a given HSIS relay is open or closed) and their external interactions (e.g., the messages exchanged between the TC and the embedded computers) in sufficient detail to capture the key behaviors of the system, such as the beam shutting off when a machine setting enters an out-of-tolerance state, or the beam staying on due to manual override. The lead CNTS engineer audited the model to ensure that it accurately describes the partial properties of the system relevant to the argument.

Figure 2.2 shows a snippet of our formalization. Lines 7–13 specify a property of the PLC that is relevant to the case. We model the changes in the state of the PLC relays and coils as events ordered by the `next` relation. Relays can be either open or closed, and coils can be either energized or de-energized. Relay 2754 of the PLC corresponds to the Therapy Sum Interlock relay in Figure 2.1. Whenever this relay is in the open state (line 7), PLC coil 1623 enters a de-energized state, which, in turn, electrically signals an HSIS relay to open and cause the beam to shut off (lines 8–13). Other parts of the system are modeled at a similar level of detail.

While formalizing the case, we relied heavily on the Alloy Analyzer [1] to discover soundness

```
1  evidence[PLC_Analysis,
2           "--mode" -> "all-paths-to-coil-contain-relay" +
3           "--network-file" -> "plc-code/cyclotron/mod1.stu" +
4           "--coil" -> "%M1623" +
5           "--relay" -> "%M2754",
6           Proof] =>
7  all relayOpen: Relay2754.state & RelayOpen |
8    some coilState: Coil1623.state & CoilDeenergized,
9         coilChangeSignal : PLC.sentMsgs & CoilChange |
10     coilState in relayOpen.next and
11     coilChangeSignal in coilState.next and
12     coilChangeSignal.coil = Coil1623 and
13     coilChangeSignal.state = coilState
```

Figure 2.2: A property of the PLC, guarded by an evidence predicate. The property states that PLC coil 1623 is de-energized when relay 2754 is open. Coil 1623 controls the neutron beam and relay 2754 is written to by the Therapy Controller (TC)—this is the mechanism by which the TC shuts off the beam when a machine setting is out of its prescribed tolerance. The property (lines 7–13) is a formula in the Alloy language [64, 1]. The evidence predicate (lines 1–6) states that the external `PLC_Analysis` tool (Section 2.4) must be called to establish that the PLC satisfies this property.

```
1 check PrescriptionSafetyCase {
2   all ms: MachineState |
3     (properties and
4     some badSetting[ms] and
5     not badSettingOverriden[ms]) =>
6     some off: Beam.state & BeamOff |
7       happensBefore[ms, off]
8 }
9 for 3 but 10 Event, 2 int
```

Figure 2.3: The Alloy formulation of the P_{rx} property. For all machine states, if all component properties (e.g., Figure 2.2) hold, a machine setting is out of prescribed tolerances in that state, and the manual override has not been enabled for that machine setting, then a "beam off" event must occur after the given state.

and vacuity errors in our argument. Checking that the model is sound (permits no behavior that violates P_{rx}) let us detect missing component properties (e.g., the Ethernet network does not drop messages), and checking that the model is not vacuous (permits some behaviors that satisfy P_{rx}) let us detect accidental contradictions in the formalization. To check for soundness errors, we asked the Analyzer to verify that our model of CNTS implies P_{rx} for all event sequences of length up to ten, as shown in Figure 2.3. The bound of ten encompasses all known treatment scenarios and is thus large enough to prevent P_{rx} from being vacuously fulfilled. To check this, we asked the Analyzer to verify that a bound of ten events is sufficient to simulate treatment scenarios in which (1) the beam remains on because the machine settings remain within tolerance; (2) a setting goes out of tolerance, but the beam shut off is manually overridden; and (3) the beam shuts off due to an out-of-tolerance setting and the absence of manual override. All of these checks explore massive potential state spaces (on the order of 2^{2800} states), and all pass in seconds.

Building Tools. Having formalized the case, we used it to decide what tools to build in order to establish the specified component properties. In particular, we determined the tool interfaces by going through the model and systematically annotating all component properties with *evidence* predicates, as shown in Figure 2.2. An evidence predicate evaluates to true if and only if the specified tool indicates success when invoked with the given arguments. For example, lines 1–6 in Figure 2.2 state that the evidence (a proof) for the PLC property is produced by the PLC analysis tool (Section 2.4), invoked with the specified parameters. We determined what kind of evidence was practically sufficient for each component (e.g., a proof, passing tests, or manual review) based on the feedback from CNTS engineers. This process of connecting properties with evidence was crucial in focusing our analysis effort (Section 2.5): building a tool that proves a narrow class of properties is much easier than building a general-purpose verifier. Our safety case both articulated these properties and helped guide the construction of tools for checking them.

2.4 Tools and Analyses

This section surveys the tools that we built to produce and check evidence for the P_{rx} safety case described in Section 2.3. We focused the analysis effort on the components that are directly modified by the CNTS engineers: the Therapy Control (TC) software and the Programmable Logic Controller (PLC). In particular, we built a linter and an SMT-based verifier for the subset of EPICS in which TC is written, a static analyzer for (a narrow class of) properties of PLC ladder logic code, and a static analyzer for the EPICS-PLC interface code. We relied on expert approval and manual review of documentation to establish the relevant properties of the embedded computers, the Hardwired Safety Interlock System (HSIS), the Ethernet network, and the physical components. The presence of this expert evidence was checked using a simple text-processing tool. We also built a safety case checker (SCC) to connect our formal model of CNTS (Section 2.3) with the results produced by the tools. We describe each of these tools in more detail below, highlighting how the safety case guided our choice of analyses, their construction, and their application.

Safety Case Checker (SCC). Our safety case checker (SCC) automates the integration of external evidence-generating tools into a formal safety argument, expressed with respect to a model of the system. We chose the Alloy language and the Alloy Analyzer as our toolset for system-level reasoning. However, a similar checker could also be created for other formal frameworks (e.g., [98, 101, 39]).

The SCC consists of two parts: (1) an extension to the Alloy language, and (2) a checker for this language extension that is based on the Alloy Analyzer. The language is extended with a small library that provides the uninterpreted `evidence` predicate. In particular, the formula `evidence[tool, args, kind]` represents the presence (or absence) of the given kind of evidence, produced by invoking the specified tool on the provided arguments. The checker, as shown in Figure 2.4, provides an interpretation for each `evidence` predicate in a safety case by performing the specified tool invocation and recording the result in Alloy. This invocation is performed through a simple plug-in interface, designed for easy addition of

```
1 procedure SCCCheck(S, P, universe_size):
2   I[evidence] ← {} // partial interpretation
3   for evidence[tool, args, kind] in FindEvidencePredicates(S):
4     result ← InvokePluggableChecker(tool, args)
5     if result = true:
6         I[evidence] ← I[evidence] ∪ {⟨ tool, args, kind ⟩}
7     return AlloyCheckSAT(S ∧ ¬ P, I, universe_size)
```

Figure 2.4: Overview of the algorithm used by our safety case checker.

new tools to the checker. The resulting interpretation I gives meaning only to the `evidence` predicates, and, as such, it is a *partial interpretation* [123, 120] for the safety case $S \Rightarrow P$, which includes additional relations (e.g., the `next` relation in Figure 2.2). The checker passes I and $S \wedge \neg P$ to the Alloy Analyzer, which checks that I cannot be extended into a finite counterexample to the safety case—i.e., an interpretation $I' \supseteq I$ that satisfies $S \wedge \neg P$ in a finite universe of discourse. The results of SCC verification are sound up to the bound on the universe size, modulo any bugs in the implementation of SCC and the pluggable checkers.

EPICS Verifier We focused our most advanced analysis, a fully automated verifier for a subset of the EPICS language, on the newest component of the system, the Therapy Control (TC) software. This tool uses an SMT solver to verify safety properties of programs written in a subset of EPICS that includes all code from CNTS. Because the Therapy Control software is finite-state, with bounded execution length and memory consumption, our EPICS verifier is both sound (it does not miss defects) and complete (it does not report false positives) for CNTS. The P_{rx} case uses the verifier to prove that the therapy control software initiates beam shut-off whenever it receives an out-of-tolerance reading from a sensor (see, e.g., Figure 2.5).

The verifier builds on Rosette [121], a language for constructing verification and synthesis tools based on SMT solvers. It takes as input an EPICS dataflow program and a safety property, symbolically interprets the program on an arbitrary state and input event, and

then invokes the Z3 solver [34] to check that the property holds in the resulting state. The output is either a guarantee that the property holds, or a concrete state and input event (a counterexample) that causes the program to violate the property.

An EPICS dataflow program is a graph consisting of edges, called *links*, and nodes, called *records*, along with configuration settings and initial values for each record. At run time, the EPICS interpreter behaves as a reactive system, responding to *events* (such as arrival of input from external devices or expiration of a timer) by updating the state of the graph and possibly sending output to external devices. In contrast to traditional dataflow systems, which automatically update all dependent values when an input value changes, the EPICS language gives the programmer explicit control over all data flow and control flow in the program. Moreover, EPICS code can modify the structure of the dataflow graph at run time, adding or removing edges, as well as modifying record configurations, such as the expression evaluated in a record. These dynamic features make it challenging to verify general EPICS code.

Unlike general EPICS programs, however, CNTS code is amenable to verification because it does not use the dynamic features of EPICS. Since EPICS forbids dynamic memory allocation and guarantees termination of event handlers, all EPICS programs with no dynamic features are finite-state, and their executions have bounded length. As a result, our verifier can automatically prove deep properties of CNTS code, without requiring loop invariants or imposing artificial bounds on heap size and execution length. Our verifier uncovered a safety-critical bug in a production version of the CNTS therapy control software, in addition to a subtle dependence of a pre-release version of the software on a bug in the EPICS runtime, as described in Section 2.5.1.

EPICS Linter The EPICS linter establishes a basic well-formedness property assumed by our verifier: all record links (i.e., dataflow edges) in a program refer to valid records. EPICS does not otherwise report broken links, instead assuming that missing records will be provided by other EPICS instances on the same local network. Ensuring that all links are valid implies

```
1 (define (couch-wrong-implies-beam-off)
2   (process_IsoGantryCouchTurntableActual)
3   (assert (=>
4     (and
5       (> (abs (- prescribed actual)) tolerance)
6       (= couch-override 0)
7       (= mode 0))
8     (= beam-interlock 0))))
```

Figure 2.5: A property of the CNTS therapy control software verified with the EPICS verifier. It states that, after a couch turntable angle reading is processed (line 2), the beam interlock is triggered ("beam off") if the couch turntable's actual rotation differs from the prescription by more than the tolerance, the manual override is disabled, and the machine is in therapy mode.

that a given EPICS program contains all code relevant to the analysis. Our linter uncovered several issues in the therapy control code, described in Section 2.5.1.

EPICS-PLC Interface Checker The EPICS-PLC interface checker ensures that a given EPICS record is connected to a particular PLC relay. This requires analyzing both the EPICS program and a separate startup script that initializes communication with the PLC. The P_{rx} safety case, for example, uses the interface checker to ensure that the EPICS Therapy Sum Interlock record is properly connected to the PLC Therapy Sum Interlock relay.

PLC Checker The PLC checker analyzes the graph of connections between coils, relays, and the power source in a ladder logic program. This checker provides two analyses that are useful for the P_{rx} case. First, it can check that a named relay's state is not updated by any other element within the PLC. Second, it can check that all paths from the power source to a named coil pass through a named relay, thus guaranteeing that the coil is energized only when the relay is closed. In the P_{rx} safety case, these checks establish that the Therapy Sum

Interlock relay in the PLC is modified only as a result of messages from the therapy control software, and that opening the Therapy Sum Interlock relay must de-energize the PLC coil connected to the Hardwired Safety Interlock System (HSIS).

Expert Evidence The expert evidence tool allows an expert to assert that a component property holds based on manual inspection of some part of the system. After examining the property in question, the expert creates a text document explaining in prose the nature of the inspection and the evidence that supports the property, along with the expert's name and the current date to support future auditing. For example, the configuration of the CNTS HSIS is defined by a non-machine-readable circuit diagram, so the P_{rx} safety case relies on expert inspection to establish properties of the HSIS. In general, our case relies on expert evidence only for claims that would be impractical to support otherwise.

2.5 Results and Discussion

This section presents the results of developing a mechanically-checkable safety case for P_{rx}, and the lessons learned from our experience. Developing the case uncovered several issues in the new therapy control software for CNTS, including two safety-critical defects. We also found that structuring the case as a system model with pluggable checkers focused our analysis effort, enabling us to perform deep checks of a complex system, while writing only 2700 lines of code.

2.5.1 Issues Uncovered

Through the construction of the P_{rx} case, we found several previously unknown issues in a pre-release version of the CNTS therapy control software, and we rediscovered an issue that the CNTS engineers found during a production test run. We discuss these issues first, and conclude by briefly describing a problem that we found in the system model itself.

Array Semantics While developing the part of the case related to the setting of the couch rotation angle, we discovered a serious issue in the therapy control code and in a major

component of the EPICS runtime. The issue concerns array calculations, which are performed using EPICS records of type `acalcout`. An `acalcout` record performs calculations over arrays of a statically specified length. All intermediate values in the computation are truncated or padded with zeros to match this length.

The affected calculation in the therapy control software uses an "in-place slice" operator that retains elements between two given indices and zeroes the rest. In the documentation [44], the bounds on this operator are both inclusive, but in the version of `acalcout` used in CNTS, the upper bound is erroneously treated as exclusive. The therapy control software behaves correctly under the exclusive semantics, but upgrading to a version of `acalcout` that correctly implements the inclusive semantics would introduce subtle errors into several calculations in the therapy control software. Our EPICS verifier, initially implemented using the inclusive semantics, detected one such error in the computation of a flag sent to the PLC to control the neutron beam, for which *sending the wrong value may cause the beam to fail to turn off when a sensor reading is out of prescribed tolerances.* Due to this issue, the CNTS engineers cannot safely upgrade this essential library—correct behavior of the software depends upon the library bug. The CNTS engineers were unaware of this problem until our EPICS verifier revealed it.

Gantry Rotation Early in the first production run of the new therapy control software, the CNTS engineers identified a safety-critical flaw in the checking of the gantry rotation angle. The gantry angle measurement ranges from $-0.5°$ to $+360.5°$, and the intent when developing the gantry rotation checks was to treat pairs of angles separated by $360°$ as equivalent, so that a rotation measurement of $360°$ would satisfy a prescription for $0°$. However, an error in the arithmetic used in the check caused the system to treat as equivalent any pair of angles equally distant from $180°$; for example, a measurement of $200°$ would satisfy a prescription for $160°$. *This error could have allowed the beam to turn on and remain on with the gantry rotation set to an incorrect angle.*

Before we had completed the relevant portion of the P_{rx} case, the CNTS engineers notified us that the code contained a bug involving the gantry rotation angle, but provided no other

details. When we completed the case, SCC detected an error: the EPICS verifier, when asked to prove that the therapy control software triggered the Therapy Sum Interlock upon receiving an out-of-tolerance gantry rotation measurement, instead produced a counterexample containing a rotation measurement and prescription value that erroneously failed to trigger the interlock. We investigated the relevant EPICS code to identify the root cause of the bug and confirmed the details with the EPICS engineers. After applying their fix to the CNTS EPICS code, SCC processed the P_{rx} safety case without errors.

Broken Links Our EPICS linter uncovered a total of 59 references to nonexistent records in the therapy control software. Three of these represented serious problems: they were caused by a misspelled record name, which would *prevent the operator from being informed of certain error conditions*. The CNTS engineers fixed these issues promptly. Another three turned out to be harmless remnants from an earlier code removal. The remaining 53 links refer to records that should have been annotated as nonlocal, since they exist in a separate EPICS installation that is accessed transparently over the network at runtime. The CNTS engineers are investigating how to annotate these nonlocal links.

Case Error Our initial formalization of the P_{rx} case contained an error, arising from a misunderstanding about the design of the system: the case claimed that an invalid couch rotation would cause the therapy control software to open the PLC's Gantry/Couch Subsystem Interlock relay, and the PLC would then internally open its Therapy Sum Interlock relay. But while trying to formulate the corresponding `evidence` invocation, we were unable to find the PLC relay number for the Gantry/Couch Subsystem Interlock. In fact, no such relay exists. The therapy software, not the PLC, combines Gantry/Couch Subsystem and other interlocks to compute the Therapy Sum, and it sends the combined Therapy Sum Interlock state to the PLC. Our strategy of connecting every component property to concrete evidence directly led to our discovery of this modeling error.

Analysis	Time (s)
EPICS verifier	3183.9
EPICS linter	0.9
EPICS-PLC interface checker	0.4
PLC ladder logic checker	0.4
Alloy Analyzer	4.1
Total (including overhead)	3190.0

Codebase	Size
PLC	5222 nodes, 10870 edges
TC	5448 lines
P_{rx} case	645 lines
SCC library	85 lines
EPICS tools	1666 lines
PLC checker	298 lines

Figure 2.6: Total analysis time for the P_{rx} case (left), sizes of the CNTS software components analyzed as part of the case (right, above the line), and sizes of the system model and tools that make up the case (right, below the line). The performance data was collected on a Debian 8 laptop with an Intel Core i7-4900MQ CPU running at 2.80GHz with 16GB RAM.

2.5.2 Performance

As shown in Figure 2.6, end-to-end checking of our safety case takes less than an hour. The figure also shows the sizes of the codebases processed by our analysis tools. The PLC software is developed in a non-textual representation, so we report its size in terms of nodes (relays, coils, and other logic elements) and edges (wires) in the ladder logic graph. The size of the therapy control codebase includes both the EPICS dataflow graph definitions and the startup script used to load the graph and initialize communication between the therapy software and external hardware. The system model for the P_{rx} case is specified in Alloy, extended with the SCC `evidence` library. The EPICS tools (the linter, verifier, and connection checker) share a common codebase, so they are reported together. Most of the common codebase is related to parsing; only 57 lines are specific to the linter, 441 are specific to the EPICS verifier, and 61 are specific to the EPICS-PLC interface checker. The case, in total, consists of 2700 lines of code.

2.5.3 Lessons Learned

Developing our safety case for CNTS led to three primary insights.

1. System Model as Safety Case Our decision to base the case on a detailed system model rather than a propositional formula (with component properties represented by uninterpreted predicates) enabled us to detect errors in the case with minimal auditing effort. A missing property in an Alloy model results in a concrete counterexample (an execution of the system that violates the safety property), whereas a missing premise in a propositional argument can be detected only through manual auditing. For example, our safety case for P_{rx} initially failed to specify that the ethernet network does not drop any packets. The Alloy Analyzer produced a counterexample, showing a scenario in which a dropped packet led to a violation of P_{rx}. Using a detailed Alloy model helped us not only ensure that relevant component properties are stated, but also that our argument is not vacuous (because the system model has no executions).

2. Simple Safety Case Checking The decision to implement our safety case checker (SCC) on top of Alloy significantly eased the development burden while providing us with ready-made automated analysis and visualization facilities. Initially, we had planned to implement SCC using a custom language to handle reasoning about the model and external evidence, as proposed in previous work [46, 31, 48, 36]. However, as part of the development of SCC, we chose to first prototype it directly in Alloy. From this prototype, the general design pattern for evidence predicates emerged, allowing us to easily connect additional external checkers to SCC, leading to a suite of lightweight but effective tools for the P_{rx} case.

3. Deep and Narrow Custom Tools We found case-guided tool development to be highly effective since it focused our efforts on the important properties of each component. With only 1964 lines of code, we built four custom tools for CNTS that check a narrow class of properties each, as specified by our system model. The integration of these tools into SCC was eased by its simple plug-in architecture. To add a new tool, the tool developer simply registers a plugin, consisting of a small Python script that can invoke the tool and interpret its output to

determine success or failure. In particular, the need for the EPICS-PLC connectivity checker became apparent only late in the development of the P_{rx} case, but because we had already implemented the plug-in architecture for SCC, we were able to both develop and integrate this checker in less than a day.

2.6 Related Work

There is a large body of literature on ensuring safety of critical systems (see, e.g., [87, 66] for a survey). This section surveys the most closely related work, focusing on previous safety efforts at CNTS, methodologies for ensuring safety of complex systems, languages for expressing safety cases, and symbolic techniques for checking properties of software components.

CNTS Safety The CNTS engineering staff has, over the years, produced a large collection of heterogeneous evidence [67] in support of CNTS safety properties, including P_{rx}. This includes a 200-page document detailing system requirements, developed in consultation with physicists and clinicians; a 2,100 line Z specification [68] of the Therapy Control software; a 16,000 LOC reference implementation of the Z specification in C (in use until July 2015); a 240-page reference manual [70]; a 43-page therapist guide [71]; and an extensive set of end-to-end testing protocols that are executed on a daily, weekly, monthly, and yearly basis. However, none of these prior efforts produced an explicit, mechanically checked safety argument. Our P_{rx} safety case is the first such argument to be created for CNTS, and its creation has already led to improvements in the new CNTS software.

Approaches to Safety Traditional approaches to system safety are *process-based*. Systems like CNTS or the Mars rover [60] are developed according to strict best practices, by highly skilled engineers. At the system level, these practices involve detailed requirements, documentation, hazard analysis, and formalization of key parts of the system design. At the code level, they include adherence to stringent coding conventions (see, e.g., [59]), manual code reviews, use of static analysis, and extensive testing. Process-based approaches are highly effective at producing low-defect code. However, they provide no explicit argument that the system as a

whole satisfies critical properties.

Our work builds on *case-based* approaches to safety (e.g., [66, 63, 95, 78]). These approaches aim to produce an explicit argument [63] that links claims about component behavior to concrete evidence in the form of tests, proofs, manual reviews, etc. Our approach to developing the P_{rx} case is most closely related to that of Near et al. [95]. We also used property-part diagrams [65] to first develop an informal case, which we then formalized in Alloy [1, 64]. But Near et al. do not connect their system-level argument to evidence; instead they use an interactive analysis to produce evidence obligations only for the software component of their target system. Our approach, in contrast, uses SCC to connect the system model to evidence generated by a variety of tools, including a fully automatic code verifier. To our knowledge, our case study is also the first to analyze low-level Programmable Logic Controllers as well as the system's software.

Languages for Expressing Safety Cases Existing languages and tools for developing dependability cases (e.g., [78, 37]) focus on managing the structure of a case. In these languages, a safety case takes the form of a semi-formal argument expressed in a graphical notation [78], with safety claims linked to evidence from heterogeneous sources. SCC, in contrast, focuses on expressing the safety argument with respect to a detailed formal model of the system, linked to tool-generated evidence. This approach enables automated reasoning about the logical correctness and non-vacuity of the safety argument.

In terms of automation, SCC is most closely related to the Evidential Tool Bus (ETB) [31]. The ETB is a general-purpose framework for tool integration, for scripting distributed workflows, and for connecting claims with supporting evidence. Its input language is a variant of Datalog. SCC shares with the ETB the idea of a semantics-neutral connection between claims and evidence, using uninterpreted predicates. In contrast to the ETB, however, SCC provides a more expressive formal language (with quantifiers and transitive closure), suitable for system modeling. SCC also provides, via Alloy, automatic soundness and vacuity checking, as well as facilities for counterexample visualization. We made heavy use of these features

while developing the P_{rx} safety case.

Software Verification There is a wide variety of verification tools (e.g., [24, 3, 29, 40, 49, 129, 12, 13]) for general-purpose programming languages. These tools are hard to build, requiring significant effort and expertise. As a result, they are rarely created for more specialized languages, such as EPICS. Our EPICS verifier is, to our knowledge, the first of its kind. Its implementation leverages Rosette [121, 122], a language designed for easy creation of domain-specific verification and synthesis tools based on SMT. Our verifier scales to real EPICS programs and is capable of finding subtle flaws that cannot be found without symbolic reasoning.

2.7 Conclusion

This chapter reported on a case study in applying modern verification techniques to construct the first mechanically-checked safety case for a real safety-critical system, the Clinical Neutron Therapy System (CNTS). Our safety case includes a detailed formal model of CNTS and a set of tools for establishing component properties specified by the model. Leveraging existing formal tools, Alloy and Rosette, we built the entire case by writing just 2700 lines of code. The construction of the case revealed serious flaws in the CNTS therapy control software and in the implementation of the EPICS language, which we reported to the CNTS staff. Our results demonstrate that formal, checkable safety cases can provide significant practical benefits by focusing analysis effort on deep properties of system components that matter for the safety of the system as a whole.

Chapter 3

RAPID DEVELOPMENT OF LANGUAGE SEMANTICS

3.1 Introduction

Mechanized verification promises strong correctness guarantees for safety-critical software systems where errors can damage equipment or endanger lives. However, mechanized verification is rarely applied to real-world safety-critical systems. This chapter identifies and addresses two obstacles to the adoption of mechanized verification: lack of verification infrastructure and resistance to changing critical system components.

First, mechanized verification requires verification infrastructure, such as formal semantics, program analyses, and verified compilers or interpreters for the programming languages used in the system. But real-world systems often use domain-specific or proprietary languages for which no such verification infrastructure exists. Developing infrastructure is very costly, often requiring *decades* of effort by experts [77], and this effort has been expended only for a few widely used languages, *e.g.*, C [84], Java [22], and Python [55].

Second, engineers who maintain real-world systems are justifiably resistant to disruptive implementation changes. Current methodologies for full formal verification typically require rewriting the system software, replacing the underlying language implementation, or both. These changes are costly, often reduce performance, and may destabilize the system (which likely depends on properties beyond those in the formal specification).

We address these problems by providing a methodology for rapidly developing a formal model of an existing language implementation, focused to support a specific program and property of interest. Applying the methodology produces a formal semantics of the target language that precisely matches the observed behavior of a reference implementation "bug-for-bug". This enables proofs about a program as it will run in production, even when

the creation or adoption of a verified language implementation is infeasible. Keeping the semantics focused on a specific program and property, rather than attempting to fully specify the semantics of every language construct and support reasoning about arbitrary properties, reduces the effort required to construct new verification infrastructure. *This makes it feasible to develop machine-checked safety proofs for critical systems regardless of implementation language.*

Our semantics development methodology, which we call the *marquer* methodology, has three main goals. First, the semantics it produces should be *faithful*: programs should be mapped to the behaviors they exhibit under the reference language implementation. A program that has been verified with respect to an unfaithful semantics might fail under real-world usage, making the verification effort worse than useless: it would give a false sense of security about a potentially unsafe system. Second, the semantics should be *focused*: it should capture only the language features necessary to reason about the target system. Coding standards for high-assurance systems often forbid the use of language features that are considered error-prone or difficult to reason about, making development of complete semantics an unnecessary burden on the verification effort. Third, the semantics should be *flexible*: the verification engineer should be able to maintain and extend the semantics with low effort. The semantics itself, like any software artifact, will likely contain bugs in its initial versions. Flexibility reduces the cost of fixing those bugs and makes it easier to focus the semantics to the target program and property of interest.

The marquer methodology achieves these goals using three key techniques. (1) For flexibility, we propose structuring the semantics around a specialized *intermediate representation* (IR), decomposing the language semantics into a translation from the source language to the IR plus an operational semantics for the IR. This design separates high-level and low-level concerns and reduces the complexity of adding new language features or bug fixes, which is often necessary to maintain focus and faithfulness of the semantics. (2) For focus, we prescribe *incremental development*: the verification engineer should model new language features only as they are found to be necessary, and only to the level of detail required to

prove the property of interest. It is sound to implement less important features conservatively (*i.e.* as a "havoc" operation that changes program state arbitrarily) or to omit them entirely. (3) For faithfulness, we recommend the use of *differential testing* throughout the development process to detect discrepancies between the semantics and the reference implementation. We also describe a technique for testing nondeterministic language semantics, as use of havoc complicates traditional testing approaches.

By applying the marquer methodology, we were able to construct mechanized semantics for two languages (EPICS and Python 2.7), build and verify analyses for these languages, and mechanically prove safety-critical invariants of the control software for a working radiation therapy installation. These successful outcomes were achieved collaboratively with the installation's engineering staff to ensure that the resulting tools both address important safety issues and complement their existing workflows. Our work was carried out in the Coq proof assistant [16], but we believe the marquer methodology would apply equally well in the context of other verification frameworks like Isabelle [99], Rosette [121], Lean [35], or Dafny [82] and within other safety-critical domains.

In short, this chapter contributes:

- a methodology for developing focused semantics for programming languages, targeted at a particular program and property of interest (Section 3.3)

- an approach to ensuring correspondence between a new language semantics and an existing reference implementation through random differential testing (Section 3.3.3)

- a case study applying these techniques to develop semantics and verify application properties for safety-critical components of CNTS, a real-world safety critical system (Sections 3.4 to 3.6).

3.2 Background

To motivate our testing-based approach to formalizing reference implementations, this section explains mechanized formal verification and describes the real-world, safety-critical context in which we developed and applied the marquer methodology.

3.2.1 Mechanized Verification

Mechanized verification refers to verification techniques that produce proofs of correctness in a machine-readable and machine-checkable format. This reduces the need to trust traditional "pen and paper" proofs, which are no more immune to errors than the system implementation in question [33]. Using an automated proof checker reduces the question of trust in the proof to the question of trust in a small, heavily vetted checker. Empirical evidence has demonstrated that mechanically verified software is extremely reliable: for instance, sophisticated differential testing techniques that uncovered thousands of bugs in unverified compilers found no bugs in the mechanically verified components of the CompCert C compiler [130, 81].

Such a high degree of reliability is clearly attractive in safety-critical contexts like radiotherapy. However, modern mechanized verification techniques require heroic effort. Verification engineers must either (1) replace the system software and its supporting language implementation with a rewrite of the software in the language of a mechanized theorem prover and an implementation of the prover's internal language or (2) develop a formalization of the semantics of the system's implementation language, mechanically prove correctness of the existing system, and then replace "only" the language implementation with a formally verified substitute. We describe these traditional approaches in Section 3.2.3.

Neither option is feasible for existing safety-critical systems, which have typically undergone extensive testing and validation, and whose engineers are therefore justifiably resistant to disruptive infrastructure changes. The problem is exacerbated by the extreme difficulty of developing complete semantics for a new language. Semantics development efforts have thus far focused on the most commonly used programming languages, such as C [84] and Java [22],

for which the payoff of defining complete semantics is highest; however, many safety-critical systems are implemented using specialized DSLs or proprietary frameworks which are less common and thus less attractive targets for semantics development.

Due to these obstacles, mechanized verification is very rarely applied to existing safety-critical systems despite its great potential benefits. The goal of the marquer methodology is to make it feasible to develop machine-checked safety proofs for critical systems regardless of implementation language.

3.2.2 The CNTS Radiation Therapy Machine

The Clinical Neutron Therapy System (CNTS) at the University of Washington Medical Center is a radiation therapy machine that uses neutron radiation to treat certain types of cancer. Its control system is complex, and malfunctions in similar systems have led to fatal radiation overdoses [86, 62], making the CNTS software a good candidate for formal verification. We used the marquer methodology to develop semantics and verify critical properties of two components of the CNTS control system.

Therapy Control Software

We first analyzed the CNTS therapy control software, which ensures that radiation is applied only in accordance with a patient's prescription. This software monitors the positions and angles of the CNTS installation's major hardware components, such as the gantry from which the neutron beam is emitted, and permits the beam to turn on only when all hardware is positioned as specified in the prescription for the current patient. It is essential for safe operation that the beam does not turn on if any hardware component is out of position; otherwise, radiation could be applied to the wrong part of the patient's body.

Our goal was to verify the integrity of the control logic in the therapy control software. The therapy control software internally manipulates dozens of status flags, each indicating a condition such as a particular hardware component being correctly positioned, and combines these flags to determine whether the CNTS installation as a whole is properly configured to

begin radiation treatment. However, the framework used to implement the therapy control software uses floating point for most numeric values and calculations, which means faults in this software could let status flags assume values such as `0.5` or `nan`. Such an error could have widespread, hard-to-predict effects on downstream safety checks due to floating-point arithmetic invalidating common integer reasoning principles such as "either $x = y$ or $x \neq y$" and "$1 - x \neq x$". Thus, we analyzed the therapy control software to prove that all status flag variables, despite having floating-point types, contain only integer values. Establishing this property enables soundly reasoning about the control logic as if it used integers instead of floats, thus addressing a class of safety concerns gathered from the CNTS engineers and complementing their existing workflows.

The CNTS therapy control software is built upon EPICS [11], a dataflow framework originally developed for controlling experimental particle accelerators. Section 3.4 describes the EPICS framework in more detail. EPICS has received little coverage in the formal verification literature, and to our knowledge, no formal semantics for the EPICS dataflow language existed prior to this work. We were thus forced to develop our own semantics to conduct our analysis of the therapy control software.

Prescription Loading Script

The second CNTS component we analyzed was a script for processing and loading patient prescriptions into the running therapy control software. CNTS uses a database to store patient prescription details, such as the intended position and intensity of the neutron beam. The prescription loading script retrieves prescription details for a given patient, converts them from the coordinate systems and units used in the prescription to those used by the CNTS hardware, and loads the converted values into the therapy control software.

We considered two critical safety properties of the prescription loading script. First, the script must *correctly convert the values* retrieved from the database to the format expected by the therapy control software. Erroneous conversion could cause the therapy control software to allow activation of the neutron beam while the hardware is improperly positioned, applying

Table 3.1: Verification tradeoffs between approaches.

	Costs			Benefits
	Replace Source	Replace Language	Verification Effort	Formal Guarantee
Testing	–	–	Lower	–
Extraction	Yes	Yes	High	Strong
VST	–	Yes	Highest	Strongest
Marquer	–	–	Highest	Strong

radiation to the wrong part of the patient's body. Second, the script must *indicate successful completion*—by outputting a result checksum and signaling termination to the EPICS control system—only when all database queries have succeeded and all prescription values have been loaded into the therapy control software. A failure partway through prescription loading could leave that software in an inconsistent state with a previous patient's prescription details still present in some fields, with similarly catastrophic results. This second safety property ensures that any failure of the script can be detected by other control system components and reported to the treatment operator.

The CNTS prescription loading script is written in Python, a widely-used imperative language. While previous work produced mechanized semantics for Python [55], we chose to develop new Python semantics as part of our analysis of the prescription loading script to benefit from the focus and flexibility of the marquer methodology, and to gauge its effectiveness on a different style of language than EPICS.

3.2.3 Marquer and Alternatives for CNTS

Table 3.1 summarizes how the marquer methodology compares to other approaches for proving that a software system satisfies a property. Each approach strikes a different tradeoff between

reimplementation effort (what components of the system must be replaced), verification effort (what new system components must be produced), and guarantees provided as a result. Software systems like the one at CNTS can be very large and complex, so reimplementation is expensive. Furthermore, system engineers are likely to be unwilling or unable to replace software components—even with a substitute that has been formally verified to satisfy certain properties—since the existing systems that have already been subject to intensive validation.

Testing Testing increases confidence in system correctness by checking that a property holds for a finite number of different inputs. It avoids the need for reimplementation by considering the behavior of the program and its supporting language implementation together, and it can typically be applied to any language implementation currently in use. However, while testing is intuitive and conceptually simple, it makes no guarantees for untested inputs. Testing is the primary method that CNTS engineers traditionally used to gain confidence in their code.

Extraction "Extraction-style" verification enables verification engineers to write machine-checked proofs about a system's behavior. This approach has been successfully used to verify large software systems like the CompCert C compiler [84]. With this technique, the software system must be (re-)implemented in the language of a proof assistant such as Coq [16]. The proof assistant then provides a way to "extract" the source code to another language (*e.g.* OCaml) where it can be compiled and executed. Extraction and compilation are typically not verified. Extraction-style verification is impractical for existing systems like CNTS because it requires both rewriting the software in the language of the proof assistant and replacing the language implementation with one for the target language of the extraction.

VST The Verified Software Toolchain (VST) is a set of tools for writing formal proofs of correctness for C programs [8]. Unlike extraction-style verification, "VST-style" verification does not require reimplementing the entire system in a new language. However, it requires extensive verification infrastructure for the language in use. Unfortunately, such infrastructure exists only for a handful of widely-used languages; for others, employing VST-style verification requires language experts to build a new formal semantics and verified language

implementation (*e.g.* a verified compiler). Furthermore, the switch to a verified language implementation required for strong formal guarantees is infeasible at CNTS and similar installations.

Marquer The marquer methodology offers strong formal guarantees without the need to reimplement any system components or produce a complete language semantics. Instead, we propose rapidly developing a *partial* semantics of a given language implementation, narrowly focused to support a specific program and property of interest, and extensively tested against an existing language implementation. The result is a semantics that matches the observed behavior of a reference implementation and is suitable for reasoning about the program of interest running atop that implementation. Focusing the development process in such a manner avoids the massive investment of time and expertise required by traditional approaches to developing language semantics, even when the target language lacks a specification for use as a starting point. And there is no need for system engineers to switch to a verified language implementation, since correspondence between the semantics and the implementation ensures that correctness proofs under the semantics will still apply to real-world executions under the existing reference implementation. Through extensive testing for correspondence between the developed semantics and the reference implementation, marquer offers similar guarantees to extraction-style verification, which similarly relies on an unverified language implementation, but fits within the constraints of real-world critical systems.

3.3 The marquer methodology

The goal of the marquer methodology is to produce, with minimal effort, a language semantics that is *flexible* with respect to extensions or corrections of the semantics, that is *focused* on supporting reasoning about a specific program and property, and that *faithfully* models a particular reference implementation of the language. The marquer methodology has three key components.

1. We propose designing the language semantics around a specialized *intermediate represen-*

tation, making the semantics flexible.

2. We present a general strategy for *incremental development* of language semantics, letting the verification engineer focus their development efforts on only features that support the program and property of interest. The flexibility of marquer-based semantics minimizes effort required to add additional features as they prove necessary.

3. We prescribe *differential testing* to ensure a faithful correspondence between the semantics and the reference implementation. Testing throughout development helps the verification engineer find bugs quickly as they add features incrementally, and having flexible semantics makes many types of bugs easy to fix.

The marquer methodology is not an automated framework: verification engineers still need to make design decisions during the process. In this respect, it is the same as any other formal verification methodology. Adhering to the design process described in this section, however, promotes focus, faithfulness, and flexibility.

3.3.1 Intermediate Representation

To construct semantics for a new language, we propose that the verification engineer separate the design into two major components: an intermediate representation (IR) with operational semantics, and a translation function that converts programs in the source language into programs in the IR. The design of the IR captures low-level details of the source language and its execution model, while the translation captures the high-level behavior of each construct of the source language.

A key advantage of this approach is that it gives the verification engineer the freedom to design an IR that is amenable to verification. The syntax and semantics of the source language are dictated by the reference implementation, but there are no such constraints on the design of the IR. By choosing simple and orthogonal IR primitives, the verification engineer can reduce the complexity of the downstream proofs and analysis tools. Furthermore,

a well-designed IR lets the verification engineer reuse existing IR primitives, which are already handled in downstream proofs and analyses, when adding new source-language features. Both properties improve flexibility of the semantics by reducing follow-on changes needed when adding or modifying features.

IR Design

We recommend using a modular design for the intermediate representation. The verification engineer should select a simple, well-understood language, such as IMP [128], to serve as a basis for the IR, then extend the IR with additional primitive operations that reflect common behaviors in the source language.

Choosing a well-understood base language frees the verification engineer to focus on the selection of primitive operations, which is crucial to achieving flexibility. In addition, a base language that is similar in control flow structure to the source language will simplify the translation of source programs to IR and minimize the overall complexity of the source language semantics. In case of doubt, a simple imperative language like IMP is often a good choice, as nearly any form of control flow can be expressed in terms of sequences of statements with explicit branching.

To select primitive operations for the IR, the verification engineer should consider the effect that each supported language construct has on the program state, factor those operations into finer-grained steps, and introduce IR primitives corresponding to each step. The goal is to identify common behaviors between language constructs that can share IR primitives. This reduces duplication of effort in downstream proofs and analyses and might be reusable for future source-language features. For example, Python's variable assignment (`x = expr...`) and function definition (`def x(): code...`) constructs respectively begin by evaluating an expression and constructing a closure, but both end by using the same logic to store the result value into the variable `x`. The same primitive operation can be reused for both destructuring assignment (`x, y, z = ...`) and for-loop iteration (`for x in collection: ...`), if support for these features proves necessary.

Translation

The translation function maps each source-language construct to IR code implementing its behavior. Given an IR that is tailored to the source language, the translation function is usually simple: in developing semantics for EPICS and Python, we used straightforward recursive traversals of the input abstract syntax tree. The design of the translation, however, is secondary to the design of the IR, as most downstream proofs and analyses will operate over the post-translation IR program.

In some cases, it may be useful to defer to "run time" (i.e., the IR operational semantics) certain decisions that could alternatively be performed at "compile time" (in the translation function). For example, our Python semantics distinguishes between local, captured, and global variables at run time, not at compile time. This is unlike the CPython interpreter, which distinguishes variable scopes during compilation to its internal IR and generates different opcodes depending on scope. Deferring scope handling to run time keeps the translation simple and adds minimal complexity to the IR step relation, as the program state used by the step relation must already track which variables are present in each scope. If static information is needed (for instance, to enable efficient compilation of the input program), later verified passes could analyze and transform the IR to remove uses of the dynamic constructs.

3.3.2 Incremental Development

We propose developing semantics incrementally, spending development effort on a language feature only if the verification engineer finds it is needed for verifying the program and property of interest. This keeps the semantics focused and avoids spending time on unnecessary features. The cost of semantics development is overhead, because the primary goal is to establish that a specific real-world program satisfies a critical safety specification. Incremental development is well-suited for language semantics because many language features are orthogonal.

Analysis of a program requires that the semantics supports every language feature used in that program. However, the semantics of a language feature need not be precise: rather than

fully understanding the feature and providing a complete specification of its behavior, the verification engineer can instead implement a simple conservative approximation. For example, the verification engineer might approximate a conditional branch as a nondeterministic choice between the "then" and "else" cases, instead of specifying the program states that trigger execution of each case. Conservatively implementing a language feature usually requires minimal effort, and the verification engineer can later refine the specification if the language feature proves relevant to the property of interest.

3.3.3 Differential Testing

Language semantics produced using the marquer methodology are useful only to the extent they accurately capture all behaviors that the reference implementation can exhibit while running the program of interest. However, formally proving a correspondence between the semantics and the reference implementation would be prohibitively time-consuming, as the reference implementation is usually large, complex, and full of implementation details (such as performance optimizations) that are not visible at the level of language semantics. Instead, we propose using *differential testing* [91], in which a testing framework provides the same program as input to both the reference implementation and an implementation based on the new semantics, then checks for discrepancies between the two executions. If the program running under the reference implementation exhibits a behavior that is not permitted by the semantics, then there is a bug in the semantics—it does not faithfully model all behaviors of the reference implementation. But if the two executions do match, the successful test provides evidence that the semantics correctly models the constructs used in the test program. Running a large number of differential tests can provide an acceptable degree of confidence in the correctness of the semantics.

The style of incremental development we recommend as part of the marquer methodology creates an additional challenge not typically present in differential testing: a semantics that conservatively models certain language features using nondeterministic choice may admit multiple distinct behaviors for a given program. These behaviors are all equally valid from

the point of view of the semantics, so a differential testing run is successful as long as the single behavior exhibited by the reference implementation is in the set of admitted behaviors. This means, however, that the verification engineer cannot simply implement a verified interpreter based on the semantics and compare its execution results to those of the reference implementation. In cases of nondeterminism, the verified interpreter might make a different choice than the reference implementation, resulting in a false-positive test failure.

The marquer methodology proposes applying differential testing to nondeterministic semantics by developing an *trace-directed interpreter*, which takes as input not only the program to execute but also a trace of the test program under the reference implementation. The trace-directed interpreter proceeds normally in cases where the semantics is deterministic. But when the semantics admits nondeterminism, rather than choosing arbitrarily from among the valid program behaviors, the trace-directed interpreter mimics the behavior in the trace, so long as it is permitted by the semantics. If the trace-directed interpreter never diverges from the trace, then the semantics admits the behavior of the reference implementation, and the test succeeds.

3.4 Developing Semantics for EPICS

This section describes how we applied the marquer methodology (Section 3.3) to develop semantics for the EPICS dataflow language in the Coq proof assistant. The section first gives a brief overview of the EPICS programming model and the property we set out to verify, then describes and justifies the decisions we made in designing an intermediate representation, incrementally adding features, and testing for correspondence to the EPICS reference implementation.

The EPICS Programming Model An EPICS dataflow program is a collection of *records*. Each record has a *record type* and a set of *fields* that each store a scalar or array value. A record's type determines the names and types of the fields it can contain, as well as the behavior of its *processing routine*. At run time, the EPICS framework handles *events*, such as the arrival

of new sensor data or the expiration of a timer, by invoking the processing routine of any record associated with the event. Depending on the values of each record's fields, the record's processing routine might perform some combination of reading and writing the record's own fields, reading and writing fields of other records, and invoking other records' processing routines. Invoking another record's processing routine behaves like a typical function call: the target record's processing routine runs to completion, then the source record's processing routine resumes execution.

Our EPICS semantics distinguishes between *configuration* and *state* fields. Values in configuration fields primarily affect the control flow of the record's processing routine, and our semantics does not support run-time changes to configuration values. Our semantics treats values in state fields as data, which can change at any time during execution. This distinction is not present in the EPICS reference implementation, but it reflects the coding standards of the CNTS engineers and helps simplify the design of the semantics.

3.4.1 Intermediate Representation

The IR in our EPICS semantics is a simple imperative language (Figure 3.1). This allows us to keep our representation of each record processing routine in close correspondence with the pseudocode provided in the reference manual [45]. Using an imperative IR also makes it straightforward to reconcile IR record processing routines with the C++ code of the reference implementation.

As described in Section 3.3.1, we selected primitives for our EPICS IR by identifying common operations in the record processing pseudocode for the records used in the program of interest. These common primitives include ReadLink and WriteLink for moving data between records, Process for invoking a record's processing routine, and HwRead and HwWrite for accessing hardware devices. Our EPICS IR also includes several specialized primitives, each used in only a single type of record, for cases where we could not see a sensible way of decomposing the relevant portion of the record processing routine into reusable pieces. An example of a specialized primitive is Calculate, used in the `calc`

$$
\begin{array}{lcl}
\mathit{db} & ::= & \mathit{rec}* \\
\mathit{rec} & ::= & \mathit{rname}, \mathit{rty}, (\mathit{field}, \mathit{value})* \\
\mathit{rname} & \in & \text{record names} \\
\mathit{rty} & ::= & \text{calc} \mid \text{ai} \mid \text{ao} \mid \text{seq} \mid \ldots \\
\mathit{field} & ::= & \text{VAL} \mid \text{INP} \mid \text{OUT} \mid \text{OMSL} \mid \ldots \\
\mathit{value} & ::= & \mathit{double} \mid \mathit{enum} \mid \mathit{string} \mid \mathit{link} \\
\mathit{link} & ::= & \mathit{rname}, \mathit{field}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{dbprog} & ::= & \mathit{rprog}* \\
\mathit{rprog} & ::= & \mathit{rname}, \mathit{rty}, \mathit{op}* \\
 & \mid & \textsc{ReadLink}\ \mathit{link}\ \mathit{field} \\
 & \mid & \textsc{WriteLink}\ \mathit{field}\ \mathit{link} \\
 & \mid & \textsc{Process}\ \mathit{rname} \\
 & \mid & \textsc{Calculate}\ \mathit{expr}\ \mathit{field} \\
 & \mid & \ldots
\end{array}
$$

Pop

$$\frac{}{\mathit{dbp} \vdash (\mathit{dbs}, (\mathit{rn}, []) :: \mathit{frames}) \rightarrow (\mathit{dbs}, \mathit{frames})}$$

SetConst

$$\frac{\mathit{setfield}(\mathit{dbs}, \mathit{rn}, \mathit{fn}, v) = \mathit{dbs}'}{\mathit{dbp} \vdash (\mathit{dbs}, (\mathit{rn}, \textsc{SetConst}\ \mathit{fn}\ v :: \mathit{ops}) :: \mathit{frames}) \rightarrow (\mathit{dbs}', (\mathit{rn}, \mathit{ops}) :: \mathit{frames})}$$

Figure 3.1: Portions of the source language syntax, IR syntax, and IR step relation definitions from our EPICS semantics

record to evaluate an arithmetic expression over fields of the current record and store the result into another field. Finally, we added several "havoc" primitives to support incremental development: HavocUpdate sets all fields of the current record to nondeterministic values, HavocWrite sets a field of another record to a nondeterministic value, and HavocProcess nondeterministically either processes a specific record or does nothing. We use these to provide conservative semantics for partially unimplemented record types.

Translation

The translation function in our semantics generates IR code implementing the processing routine for each record in the EPICS dataflow program. Translation of a record takes as input the type of the record and the values of its configuration fields, and outputs a copy of the IR implementation of the processing routine for the relevant record type, specialized to the record's configuration values. While the EPICS manual and reference implementation define record processing routines as branching and iterating at run time based on the values of various fields, our semantics, which treats these configuration fields as immutable, evaluates these branch conditions and unrolls loops statically during the translation to IR. The resulting IR code branches only on values from mutable state fields, and contains no looping constructs, which simplifies the development of both analyses and proofs.

The entire definition of translation is only 360 lines of code, and the longest single case (for handling the `seq` record type) consists of only only 25 lines. This is due in large part to our IR design, which avoids unnecessary subdivision of primitive operations.

3.4.2 Incremental Development

We developed our EPICS semantics incrementally, beginning with support for only the most essential record types based on our reading of the EPICS documentation and the CNTS therapy control software. We used the requirements of the calculation (`calc`) and analog output (`ao`) records to guide our design for the core features of our EPICS semantics, such as basic control flow and movement of data along dataflow links. We then added further

record types, eventually including all record types used in the therapy control software, and support for language features such as implicit conversions. In total, we implemented 19 record types out of the 31 available in EPICS Base and the EPICS extensions used by the therapy control software. Of these, 5 were not essential to our verification goals, so we modeled them conservatively, replacing most or all of the record processing logic with havoc.

Our approach to incremental development let us avoid significant work in some areas of the semantics design. For example, our semantics initially did not support automatic conversion between value types: a program that attempted to copy an integer or string value into a floating-point field would simply get stuck. However, we discovered that the CNTS therapy control software makes use of such conversions, so we could not run the therapy control software under such restrictive semantics. Not wanting to fully specify EPICS's complex conversion behavior, we next extended our semantics with a sound conservative approximation: converting a value to its own type (for example, converting an integer to an integer) leaves the value unchanged, and converting to any other type produces an arbitrary value of the target type. This is a valid, though imprecise, model of the behavior of the EPICS reference implementation, but again we found it insufficient. The therapy control software includes a number of float-to-integer-to-float conversions, with the intermediate integer value being used to select a message to display to the treatment operator. Specifying the float-to-integer and integer-to-float conversion steps to produce arbitrary output left us unable to prove the desired property of the high-level control logic in the therapy control software. Thus, we further refined our specification of conversions, but only in the two cases that are relevant to our proof: integer-to-float conversions, and float-to-integer conversions where the conversion is exact. This let us complete our proof of the control logic integrity property without specifying the rounding and truncation behavior of the float-to-integer conversion or the details of conversions between numeric types and strings.

3.4.3 Differential Testing

As described in Section 3.3.3, we checked for correspondence between our EPICS semantics and the reference implementation by applying differential testing between an instrumented copy of the reference implementation and a trace-directed interpreter based on our semantics. We chose instrumentation points in the reference implementation partially based on the needs of our semantics. In addition to recording events at the start and end of each record processing routine for comparison with the trace-directed interpreter, the instrumented reference implementation also records events after each movement of a value across a dataflow link, as this operation sometimes involves a type conversion that our semantics models as havoc, and following evaluation of a string expression, which our semantics again models as havoc. Instrumenting these operations makes it easy for our trace-directed interpreter to choose appropriate values for nondeterministic operations, reducing false positives from differential testing.

We employed randomized differential testing throughout the development process, checking each new feature as we added it to the semantics. This testing often uncovered discrepancies, due both to simple implementation bugs and to lack of detail in the EPICS documentation. For example, after reading the documentation, we wrongly assumed that EPICS restricted fields of `ENUM` type to contain only numeric values less than the number of valid enumerators, with coercions occurring on out-of-bounds writes. But in fact, an `ENUM` field may contain any integer in the range $[0, 2^{16})$, regardless of the number of enumerators defined for the field. Random differential testing quickly uncovered this bug by generating cases where a `DOUBLE` value flowed into an `ENUM` field: our semantics declared that the conversion of even out-of-range values (*e.g.* 123.0) should be capped to produce an in-range `ENUM`, but the reference implementation produced larger values (*e.g.* 123).

3.5 Developing Semantics for Python

This section describes the process of applying the marquer methodology to develop semantics for Python in the Coq proof assistant and to verify the CNTS prescription loading script described in Section 3.2.2. Specifically, we modeled a subset of Python 2.7 sufficient to prove that the prescription loading script correctly converts prescription data between the coordinate systems used in the prescription database and in the CNTS therapy control software, and that it signals successful completion only if all read, conversion, and write operations have actually succeeded.

A notable property of the prescription loading script is its use of two third-party libraries, in addition to parts of the Python standard library. It uses `psycopg2` [124], a database adapter library, to communicate with the Postgres database that stores the prescription data, and it uses the `pyepics` library [97] to interface with the EPICS-based therapy control software. Our Python semantics includes formal models of the behavior of the relevant components of `psycopg2`, `pyepics`, and the Python standard library.

3.5.1 Intermediate Representation

Our Python IR is a stack-based bytecode language, similar to the bytecode used in the CPython reference implementation [107]. However, while a few of our IR operations, such as like Dup and Return, are analogous to CPython bytecodes, others, like Iterate, are higher-level primitives that simplify proofs written against our semantics. Where the CPython bytecode implements loops and conditionals iteration using jumps, making for an efficient implementation, our IR supports only structured control flow so as not to require reasoning about an explicit program counter.

The prescription loading script uses only a very small subset of Python. In particular, it does not handle exceptions with try/catch and it does not use any of Python's object-oriented features. It does not make use of special methods [108] such as `__add__` (colloquially called "magic" methods). Furthermore, it interacts with Postgres via six hardcoded queries, each

$$
\begin{array}{lcl}
\mathit{state} & ::= & \mathit{frame}^*, \mathit{fname} \mapsto \mathit{funcDef}, \mathit{addr} \mapsto \mathit{obj}, \mathit{world} \\
\mathit{frame} & ::= & \mathit{scope}^*, \mathit{locals}, \mathit{stack}, \mathit{op}^* \\
\mathit{fname}, \mathit{addr} & \in & \mathbb{N} \\
\mathit{funcDef} & ::= & \mathit{op}^*, \mathit{locals} \\
\mathit{scope} & \in & \mathit{name} \mapsto \mathit{val} \\
\mathit{locals} & \in & \mathit{name}^* \\
\mathit{stack} & \in & \mathit{val}^* \\
\mathit{world} & ::= & \mathit{pgState}, \mathit{epicsState} \\
\mathit{pgState} & \in & \mathit{Auth} \mapsto \mathit{Query} \mapsto \mathit{Row}^* \\
\mathit{epicsState} & \in & \mathit{name} \mapsto \mathit{val} \\
\mathit{val} & ::= & \text{None} \mid \text{Int } i \mid \text{Str } s \mid \text{Tuple } \mathit{val}^* \\
 & \mid & \text{Ptr } \mathit{addr} \mid \text{Func } \mathit{fname}\ \mathit{scope}^* \mid \ \ldots \\
\mathit{obj} & ::= & \text{List } \mathit{val}^* \mid \text{Dict } (\mathit{val}, \mathit{val})^* \mid \text{PGCursor } \mathit{row}^* \mid \ \ldots \\
\mathit{op} & ::= & \textsc{Dup} \mid \textsc{PushLit}\ \mathit{val} \mid \textsc{Load}\ \mathit{name} \mid \textsc{Store}\ \mathit{name} \\
 & \mid & \textsc{UnOp}\ \mathit{uop} \mid \textsc{BinOp}\ \mathit{bop} \mid \textsc{MkFunc}\ \mathit{fname} \\
 & \mid & \textsc{If}\ \mathit{op}^*\ \mathit{op}^* \mid \textsc{Iterate}\ \mathit{op}^* \mid \textsc{Call} \mid \textsc{Return} \mid \ldots \\
\mathit{uop} & ::= & - \mid \ \sim \ \mid \text{not} \\
\mathit{bop} & ::= & + \mid - \mid * \mid \ldots
\end{array}
$$

$$
\textsc{IfTrue}\quad \frac{\mathit{as\_bool}(h, v) = \text{True}}{((s, l, v :: \mathit{stk}, \textsc{If}\ o_1\ o_2 :: \mathit{ops}) :: \mathit{fs}, c, h, w) \rightarrow ((s, l, \mathit{stk}, o_1 +\!\!+ \mathit{ops}) :: \mathit{fs}, c, h, w)}
$$

$$
\textsc{IterateNext}\quad \frac{h(p) = i \qquad \mathit{iter\_next}(i, i', v) \qquad h' = h[p \mapsto i']}{((s, n, \text{Ptr } p :: \mathit{stk}, \textsc{Iterate}\ o :: \mathit{ops}) :: \mathit{fs}, c, h, w) \rightarrow ((s, n, v :: \text{Ptr } p :: \mathit{stk}, o +\!\!+ \textsc{Iterate}\ o :: \mathit{ops}) :: \mathit{fs}, c, h', w)}
$$

Figure 3.2: Our Python IR and selected semantic rules.

of which takes a small number of string or numeric parameters, and uses only the basic `caget` and `caput` functions from the `pyepics` library, which respectively read and write a global EPICS variable.

Our semantics reflects this subset. Our semantics models Postgres as a set of databases identified by *Auth* strings containing username, password, and database name. Each database maps string queries to results, *i.e.* lists of rows. Our semantics models EPICS as a map from global EPICS variable names to values.

Our semantics models the state of a running Python program (*state* in Figure 3.2) as four parts: the call stack, the set of all function declarations in the program, the heap, and the state of the outside world. Each call stack frame includes a list of enclosing variable scopes, a list of local variable names, a stack of intermediate computation results, and a list of IR instructions remaining to execute. To ease reasoning, our semantics divides Python objects into immutable "values" and mutable, heap-allocated "objects". Values include integers, function closures, and pointers to mutable objects, while objects include collections like lists and dictionaries, iterators, and Postgres result cursors.

Translation

Translating Python source to the IR in Figure 3.2 amounts to a standard conversion of statements and expressions into stack machine language. For example, the expression $x + 1$ becomes $[\textsc{LoadVar}\ x; \textsc{PushLit}\ 1; \textsc{BinOp}\ +]$.

There are two less-trivial tasks that the translation also handles. First, while Python allows function definitions to appear at any statement location in the AST, our IR semantics relies on a flat collection of function bodies (*code* in Figure 3.2) to simplify the handling of closures. The translation must therefore extract function bodies to build this collection and replace declarations with appropriate closure construction operations. Second, variable lookup behavior in Python varies depending on whether the variable is classified as "local". The set of local variable names is a global, static property of the function: absent an explicit declaration, a variable is local if it appears as the target of an assignment in any statement of

the function, regardless of when (or if) those statements actually execute. The translation in our semantics makes an extra pass over the function to collect the names of all local variables and stores this with the generated IR for the function.

3.5.2 Incremental Development

We began developing our Python semantics by determining which Python language features the prescription loading script uses. This process yielded a list of 7 library modules, 12 statement types, and 17 expression types we would need to support. This is in contrast to Python's several-hundred-module standard library, 23 statement types, and 22 expression types [109]. Even for supported language features, our semantics does not fully define behavior for every case. For instance, while the multiplication operator is defined for many Python types, including strings, and tuples, we only define its behavior on ints and floats.

We implemented features incrementally, starting with a working implementation that supported functions, variable assignment, printing, and integer expressions; we then worked toward more complex features like lists, dictionaries, and floating-point numbers.

3.5.3 Differential Testing

To test our Python semantics against the reference interpreter, our differential testing framework runs both and compares their output on `stdout`. Compared to the EPICS trace-directed interpreter (Section 3.4.3), this simpler design is possible since our semantics does not rely on nondeterminism for any features in the subset of Python used by the prescription loading script.

We used differential testing on every commit during development to ensure that new features were correctly implemented. Section 3.6.2 further discusses our process.

3.6 Evaluation

We evaluated the EPICS and Python semantics developed using the marquer methodology to ensure that they are *focused* on the program and property of interest, *faithful* to the

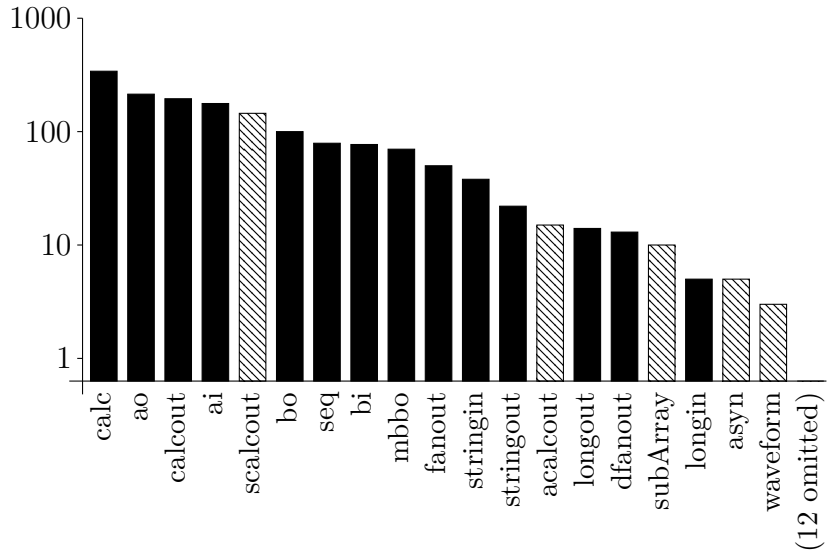

Figure 3.3: EPICS record types and implemented in our semantics. Bar height is the number of times the record or field appears in the CNTS therapy control software. Filled bars are fully implemented; hatched bars are conservatively approximated.

behaviors of the reference implementation, and *flexible* in the face of changing requirements. We evaluated focus by comparing the feature sets of our semantics with the features used by the program and property of interest, we evaluated faithfulness by analyzing the results of thorough differential testing of the semantics, and we evaluated flexibility by considering the effort required to add new features to the semantics.

3.6.1 EPICS

Focus

A language semantics developed using the marquer methodology could fail to achieve focus in two ways: by omitting features that are essential to the property of interest or by supporting unnecessary features. Omissions lead to useless semantics. They are easy to avoid: we simply continued expanding the semantics until we succeeded in proving the property of interest.

Unnecessary features indicate wasted development effort. To check that our EPICS semantics does not include many unnecessary features, we compared its feature set to the features used in the CNTS therapy control software. Figure 3.3 shows the result of this comparison: each record type supported by our EPICS semantics is used at least once in the therapy control software, and many of the less commonly used record types are implemented using simplified conservative approximations. Twelve record types are omitted from the graph because they never occur in the CNTS therapy control software and are unimplemented in our semantics.

Faithfulness

We established confidence in the faithful correspondence of our EPICS semantics and the EPICS reference implementation by applying differential testing. We used differential testing runs of 10^5 to 10^6 randomized tests throughout the development process to quickly detect newly-introduced bugs. While these were effective at finding even rare bugs—for example, one differential testing run uncovered a bug in our parsing of floating-point infinities, which caused only a single test failure in a run of over 1.2 million—the frequent changes to the semantics between runs means that the results of these tests do not provide full assurance that the final version is correct.

To gain higher confidence in the correspondence between our semantics and the reference implementation, we ran extensive differential testing on the final implementation. Specifically, we ran over 238 million tests obtained by exhaustive enumeration of all programs up to a size bound. This took roughly 5,000 CPU-hours, running primarily on a cluster of ten 36-core Amazon EC2 virtual machines. We bounded program size to a maximum of two records and four initialized fields, as this kept the number of tests within our budget while still providing coverage of nontrivial interactions between every pair of record types.

We chose bounded exhaustive testing for testing the final semantics to provide a strong and clear guarantee of correctness over an entire region of the input space. Randomized testing covers a larger space of inputs, but only sparsely, and it is easy to inadvertently bias test generation away from parts of the input space. Furthermore, in our experience, the

vast majority of bugs in the EPICS semantics result from either a single record type making erroneous updates to internal record state or from permitting one record to interact with another record's fields in an unexpected manner. Neither type of bug requires a large number of records to uncover, mitigating the limitations of exhaustive testing. Randomized testing still proved useful during development, where it helps to quickly cover a wide variety of inputs for new language constructs.

Exhaustive testing uncovered one bug: the logic in our semantics for taking action when a field value changes relied on ordinary IEEE-754 floating point comparison to detect changes, while the same logic in EPICS includes a special case, treating floating-point infinities as unequal to themselves. We had previously discovered and corrected an identical bug, but reintroduced it during subsequent edits to the semantics. We reapplied the fix, which involved only a small change to a single function, and reran the failing tests to ensure we had corrected the issue. As the exhaustive testing process uncovered no other discrepancies between our semantics and the reference implementation, we have confidence that the semantics faithfully models the implementation.

Flexibility

To evaluate the flexibility of semantics produced using the marquer methodology, we measured the time and code changes required to extend our EPICS semantics with a new feature: *alarm propagation*. As a control, we also extended a previous version of our EPICS semantics, which we designed and implemented using a more traditional methodology. We had not implemented alarm propagation previous to running this experiment because it is very rarely used in the CNTS control program.

The alarm system in EPICS is used to propagate status codes throughout the dataflow graph alongside ordinary data movement. For example, if an input record reads an out-of-range value from a hardware sensor, it may set an alarm status, which will propagate downstream to calculation nodes that use the input value and from there to additional calculation or output nodes. The application can use alarm status values directly, as inputs to calculations,

Table 3.2: Effort required to add alarm propagation to the marquer and traditional versions of our EPICS semantics.

	Traditional	**Marquer**	Change
Time spent	2h21	1h05	-54%
Lines added	938	640	-32%

or indirectly, by automatically disabling output records to prevent driving actuators using erroneous values. We implemented only the propagation of alarms, not the input and output components, because we judged propagation to be the most complex aspect of alarms.

Table 3.2 summarizes the results of our experiment: the changes to the marquer semantics modified 32% fewer lines of code and took 54% less time to implement. We identify two main ways that the marquer methodology contributed to the flexibility of the marquer EPICS semantics.

First, the marquer methodology promotes defining only the intermediate language semantics using a step relation, which allows the number and complexity of relations involved to be kept low. In contrast, our traditional EPICS semantics used relational definitions for the semantics of the entire language. The complexity of these definitions contributes directly to the number and complexity of cases in many proofs about the semantics, and keeping the complexity low makes it easier to update proofs after adding or extending language features. For comparison, the traditional semantics defines its step relation using 13 relations with a total of 34 cases and an average of 8.2 lines per case, whereas the marquer semantics uses only 3 relations, 22 cases, and 6.9 lines per case.

Second, the marquer methodology's approach of targeting most analyses and proofs to the intermediate language reduces the coupling between definitions. Propagating an alarm status from one record to another requires knowing the names of both records, but in our traditional EPICS semantics, the relations defining the effect of reading or writing across

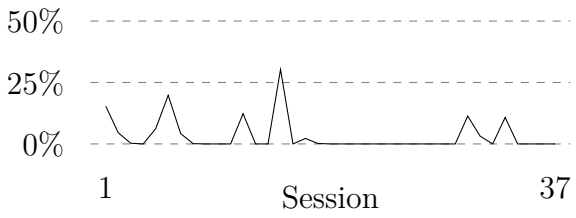

Figure 3.4: Error rates for differential testing runs of Python semantics during project development.

a dataflow link had access only to one of the two record names. Adding alarm propagation as an effect of data access required adding a new input to these relations, and the effects of this change cascaded throughout the codebase, requiring new arguments in the definitions and proofs of the well-formedness checker, the proofs of preservation, and the interpreter. In contrast, when adding alarm propagation to the marquer EPICS semantics, we added a new "propagate alarm" operation in the intermediate language, independent of the existing data access operations. Aside from adding cases handling the new operation, there was little need to restructure existing definitions or proofs. We still needed to thread an extra record name argument through the parts of the denotation function that handle source-level dataflow links, but these changes were contained to only a few functions in a single file.

3.6.2 Python

Focus

Our Python semantics are useful: we used them to implement a lightweight symbolic interpreter for Python, proved the symbolic interpreter correct with respect to our semantics, and used it to check the safety of the CNTS prescription loading script. Given the input `argv` and a symbolic Postgres database state, the symbolic interpreter outputs the resulting EPICS state as a function of the input.

The symbolic interpreter outputs the symbolic EPICS state after the script terminates. A

CNTS engineer manually inspected the generated formulas to verify that the conversion was done correctly. We also verified that all execution paths ending in errors do not write the final checksum to EPICS.

The symbolic interpreter extends *val* in Figure 3.2 to include symbolic integers and floats. The *state* type is also extended to include its path condition as a symbolic boolean. Execution proceeds exactly as in the usual step relation, but symbolic values are not reduced. When execution branches on a symbolic value the symbolic interpreter simply forks execution and runs each side separately.

To help alleviate the resulting exponential path explosion, symbolic *state*s can be merged when they are *structurally identical*: every part of the state is identical except integers and floats. When an integer or float value v differs between two states, in the merged state it has the symbolic value $ite(pc_1, v_1, v_2)$, where pc_1 is the path condition of the first state. In practice state merging is usually possible; most branches in the prescription loading script conditionally set float variables to zero without affecting the heap.

Faithfulness

During development of the Python semantics, a differential testing process ran in the background continuously running tests against the latest commit. Each new feature typically introduced new failures, as it was either incorrect or had an unexpected interaction with other language features. Figure 3.4 illustrates the outcome: spikes in error rates result from new features being incorrectly implemented, and are fixed over one or two subsequent commits. In the course of development we ran over 41 million tests on randomly generated Python programs. The programs varied in size up to 100 AST nodes. As with EPICS, we performed bounded exhaustive testing for Python on programs up to AST size 3. This resulted in 98080 distinct test programs. All tests passed without modifying the semantics, demonstrating the effectiveness of the marquer methodology and providing confidence in the faithfulness of our Python semantics.

Flexibility

Our Python semantics were developed by two different people: one set up the initial IR and translation passes, and the second implemented new features until the semantics covered all features used by the prescription loading script. Out of 2300 lines of code in the original implementation, only 200 needed to be modified to add all the required features. Many more lines of code were added to the initial core; the final semantics is 4100 lines of code. Of the lines that needed to be modified, most were simple code formatting and organization (*e.g.* moving related definitions to their own files). The rest—roughly two dozen—were bug fixes to correct errors revealed by differential testing.

3.7 Related Work

Semantics Development Recognizing the value of mechanized semantics, researchers have attempted to ease the task with tools. In particular, the K Framework [111] aims to enable modular development so that new language features can be added without changing the existing semantic rules [115]. K has been used to write complete mechanized semantics for C [43, 42, 56], Python 3 [55], Java [22], JavaScript [102], and others.

Instead of a specific tool, we have presented a methodology: a way of developing semantics that is agnostic to the underlying tool. The marquer methodology could be carried out in K just as easily as in Coq. Furthermore, efforts with K have been focused on *complete* semantics that perfectly capture the intent of the language designers. Complete semantics are suitable for developing verified compilers [84]. In contrast, the marquer methodology is focused on *partial* semantics that perfectly capture the real-world behavior of a reference implementation. Our style of semantics is more useful for proving properties about specific programs without investing the full effort to construct and validate a complete semantics.

Partial Semantics Not all semantics development efforts aim for completeness. In developing the verified C compiler CompCert [84], full formal semantics were developed for Clight [20, 19], an intermediate language that is a subset of C. Semantics for the complete C language are

defined as the composition of the translation pass from C to Clight plus the semantics for Clight—similar to our decomposition into translation and bytecode (Section 3.3).

Other developments model portions of a language in order to prove properties about the language itself, such as Featherweight Java [61] or recent work on C++ constructor semantics [110].

Testing Language Implementations Modern compiler implementations typically try to ensure correctness through use of thorough, manually-constructed test suites. The GCC compiler, for example, ships with a test suites composed of tens of thousands of individual tests [47]. Commercial vendors sell similar test suites intended for validating conformance of a C language implementation to the standard [105]. However, such manually-constructed test suites are often incomplete [102].

The aim of random differential testing is to apply random test generation to achieve better coverage of the analyses and transformation used in a compiler or interpreter. Csmith [130] has applied this technique to great effect: Csmith detected over 300 bugs in production-quality C compilers, which had not been discovered using manually-written tests. LangFuzz [58] uses a language grammar and a suite of example programs to generate interesting programs to find bugs in language runtimes. Researchers have even used random testing to synthesize semantics for x86 assembly [50]; in the future, we might be able to synthesize parts of our semantics from tests. Bishop et. al. used differential testing to validate their TCP semantics with respect to existing implementations [17, 18], and Owens similarly validated semantics for a subset of OCaml [100].

3.8 Conclusion

This chapter introduced the marquer methodology for developing faithful, focused, and flexible language semantics. Such semantics enable constructing machine-checked safety proofs for existing safety-critical systems *regardless of implementation language, and without requiring disruptive infrastructure changes.* For flexibility, the marquer methodology decomposes the

language semantics into a core intermediate representation (IR) and a translation from program syntax to the IR. For faithfulness, it prescribes differential testing to ensure agreement with a reference implementation. For focus, it models a subset of the target language, just enough to prove safety for the program of interest. By applying the marquer methodology collaboratively with the engineering staff at a working radiotherapy installation, we were able to construct mechanized semantics for two languages (EPICS and Python 2.7), build and verify analyses for these languages, and mechanically prove safety-critical invariants of the system's control software.

Chapter 4

RAPID CONSTRUCTION OF VERIFIED COMPILERS

4.1 Introduction

Formal verification provides rigorous guarantees of software safety and correctness. But flaws in the underlying interpreter, compiler, or runtime system can compromise those guarantees. Verifying the language implementation itself rules out such flaws, but constructing verified compilers or interpreters with adequate run-time performance remains a substantial engineering task, and thus far has been done only for a few languages [83, 80].

This chapter presents a new design for verified compilers, which we call a *denotational compiler*. A denotational compiler for a given language consists of a verified denotation, which maps programs in that language to functions in some host language, and a verified extraction procedure, which translates functions in the host language to executable code. Typically, the host language will be the built-in language of a theorem prover, such as the Gallina language used in Coq. This approach substantially reduces verification effort: the denotation component maps source programs to shallowly-embedded functions, much as an interpreter does, and is easy to verify thanks to most provers' extensive built-in support for reasoning about their own built-in languages. Verification of the extraction procedure is more difficult, but can be done only once and reused to build verified denotational compilers for any number of languages. And compared to verified interpreters, which have similar verification cost, a denotational compiler offers better performance by eliminating the overhead of explicit AST manipulations and allowing more optimization opportunities during extraction.

Denotational compilers benefit greatly from using a verified extraction procedure with a verified frontend, a minimal runtime system, and customizable code generation. As the extraction procedure serves as the backend of the overall denotational compiler, an unverified

frontend would introduce unverified steps into the middle of the compilation process, where it may be hard to check correctness through other means. Runtime systems are particularly challenging to verify, and if an unverified runtime must be used, then keeping it small helps to limit the trusted computing base (TCB) of compiled programs. Finally, customizing code generation allows optimizing the extraction procedure's handling of common or performance-critical elements of the denotation function's output, which is essential to achieving high performance for generated code.

To meet the verified extraction requirements of denotational compilers in Coq, we present Œuf, a verified compiler from Gallina, Coq's functional programming language, to executable x86 assembly. First, Œuf avoids relying on an unverified parser or grammar by using translation validation [106, 114]: the Œuf compiler reflects Gallina programs into a deeply embedded expression language, then generates a proof that the computational denotation of the reflected expression is equivalent to the original function. This validation theorem, like that of CakeML [94], guarantees that the program Œuf compiles really is the program provided as input. Second, Œuf-compiled executables rely only on a simple (but unverified) memory allocator, reducing the potential for runtime system bugs to affect program behavior. Third, Œuf provides an extensible mechanism for customizing data layout and code generation. This allows developers of denotational compilers to incrementally optimize performance for the Gallina constructs used in the denotation of the source language.

In summary, this chapter contributes:

1. A method for developing verified compilers with less verification effort, while still maintaining acceptable performance of the generated code (Section 4.2);

2. A design for verified extraction of Gallina programs, with features that are particularly well-suited for use in a denotational compiler (Section 4.4 and Section 4.8); and

3. The prototype Œuf verified compiler (Section 4.5), and examples of its use in constructing a denotational compiler for a non-trivial expression language (Section 4.9).

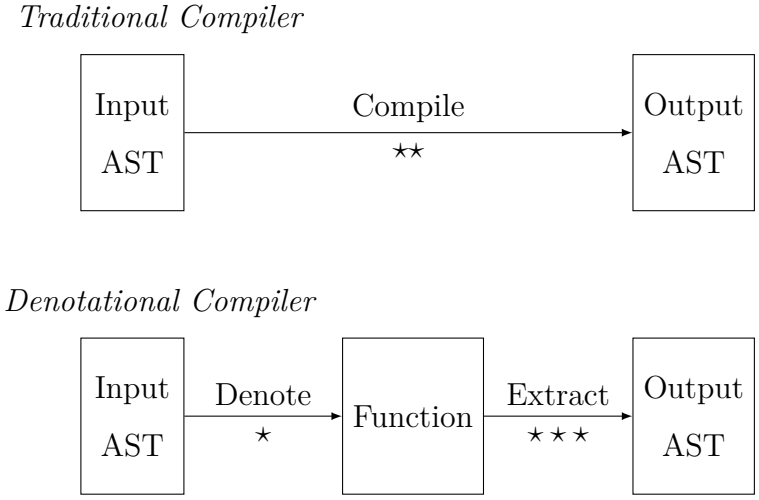

Figure 4.1: Architecture of traditional and denotational compilers. Stars indicate the relative verification effort for each translation step. While the extraction procedure is difficult to verify, it is general-purpose and can be reused to build denotational compilers for many languages.

4.2 Denotational Compiler Architecture

Developing a verified language implementation with adequate run-time performance remains a challenging engineering problem. Interpreters are relatively easy to write and verify, but performance suffers due to explicit run-time manipulation of program representations. Compilers, on the other hand, can produce high-performance generated code, but compiler verification is tremendously labor-intensive.

In this section, we propose a new design for verified compilers, which we call the *denotational compiler* architecture. Denotational compilers are nearly as easy to verify as an interpreter (given the availability of certain common infrastructure, described below), but can be engineered to produce optimized code, achieving run-time performance similar to that of a traditional verified compiler.

Concretely, a denotational compiler is the composition of a *denotation function* and

an *extraction procedure*, as shown in Figure 4.1. The denotation function, which itself is written in the prover's language, maps a deeply-embedded representation (*i.e.*, an AST) of a source-language program p to a shallowly-embedded function in the prover's language that implements the behavior of p. The type of the denotation function depends on the design of the source language semantics, but a type such as `program → (input → output)` would be typical. The extraction procedure compiles functions of the prover's language down to the final target language. Extraction may involve intermediate steps (for example, running Coq's built-in extraction of Gallina to Ocaml followed by Ocaml compilation), but here we consider only the overall input (prover's) and output (target) language of the extraction procedure.

4.2.1 Verification

The primary advantage of the denotational compiler architecture is that is it especially amenable to formal verification. Proving correctness of a denotational compiler amounts to proving correctness of its two components. Verifying the extraction procedure for a given prover's language is a significant engineering effort, but once completed, the same extraction procedure can be reused in multiple denotational compilers with different source languages. (And we have already constructed and verified one such extraction procedure: the Œuf compiler for Coq's Gallina language, described in detail in later sections of this chapter.) Each source language still requires a separate denotation function, but the verification process for these functions is much easier than development of a traditional verified compiler.

Where a traditional compiler transforms one deeply-embedded program representation to another, the denotation function's output is shallowly embedded—essentially, it is an ordinary function in the prover's language. And provers typically have extensive built-in support for reasoning about their own internal languages, which the compiler developer can readily exploit when establishing correctness of the denotation function. The result is that verifying the denotation function is more similar to verifying an interpreter than to the much more complex process of verifying a traditional compiler.

```
1 Fixpoint denote (e : expr) : Z -> Z :=
2     match e with
3     (* ... *)
4     | Add a b =>    (* a, b : expr *)
5         let af : Z -> Z := denote a in
6         let bf : Z -> Z := denote b in
7         (fun x : Z => af x + bf x)
8     (* ... *)
9     end.
```

Figure 4.2: Example showing part of a denotation function for a simple arithmetic expression language. The `add` case carefully avoids interleaving computations over the expression AST `e` and over the run-time input value `x` to ensure that partial evaluation of `denote` applied to an expression can eliminate all explicit AST manipulation.

4.2.2 Denotation

The design of the denotation function is essential for attaining the performance benefits of the denotational compiler design. Using a naïvely written denotation function, or using an interpreter as the denotation (as the two functions typically have similar types), will produce a functioning compiler, but one whose performance is more similar to a basic extracted interpreter. A properly desgined denotation function should "evaluate away": when the denotation is applied to a source program representation (but not to that program's input values), partial evaluation should produce a term implementing only the program behavior, with no traces of the original deeply-embedded program representation or code for manipulating it.

The key to designing a high-quality denotation function is to maintain a clear distinction

between code that makes up the implementation of the denotation function and code that is part of the denotation's output. See, for example, Figure 4.2: in this simple example, `af`, `bf`, and the parenthesized lambda term will appear in the generated output, while the pattern match on `e` and the recursive calls to `denote` are parts of the denotation that will evaluate away when the denotation is applied to an expression. When implementing more complex languages, maintaining this separation can be challenging: the shallowly-embedded nature of the target program representation means that the output code is written using the same syntax as the denotation code and can freely refer to its definitions and local variables, but doing so will cause portions of the denotation to be embedded in the output program, sabotaging performance of the generated code. However, while the similarity of denotation and output code can make construction of the denotation more complex, it is essential for easing verification, as it permits reasoning about the combination of both denotation and output code with minimal friction between the two.

4.2.3 Extraction

The use of a general-purpose extraction procedure means that the performance of code produced by a denotational compiler will typically be lower than that of code compiled in a traditional manner. A major reason for this performance deficit is the lack of fine-grained control over low-level data representation, which is absent from most prover languages. However, many extraction procedures support customization of code generation. For example, Coq's (unverified) built-in extraction mechanism allows replacing Gallina types and functions with more efficient representations in the target language, which can significantly improve performance of extracted code. If performance is a concern, the compiler developer can configure extraction to replace performance-critical types and operations used in the output of the denotation function with more efficient equivalents.

The Œuf verified Gallina compiler, presented in Sections 4.3 through 4.8, is uniquely well-suited for use as the extraction procedure for a verified denotational compiler. Œuf's parser (rather, its reflection procedure) is verified by translation validation, providing assurance that

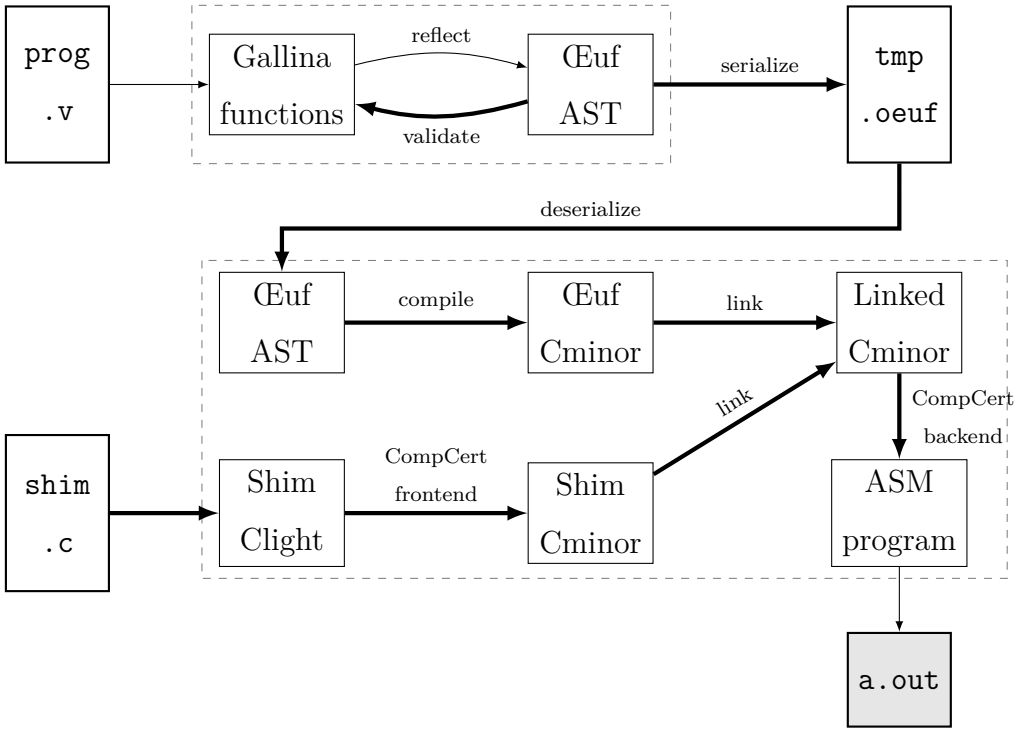

Figure 4.3: The Œuf compilation workflow. Arrows in bold indicate formally-verified operations. The output is `a.out`, indicated with gray.

the AST compiled by Œuf is an accurate representation of the Gallina function emitted by the denotation function. Œuf's runtime requirements are minimal: compiled code needs only the CompCert support library and an implementation of `malloc`. Finally, Œuf provides a verified mechanism for customizing code generation, allowing the developer of the denotational compiler to provide optimized low-level implementations of operations generated by the denotation function.

4.3 Œuf Compiler Overview

Figure 4.3 shows an overview of the Œuf compilation process. The user provides a Gallina function, such as the `max` function in Figure 4.4, which computes the maximum of two natural numbers. The user also writes C code as a "shim" to connect their pure Gallina functions

```
1     Fixpoint max_orig (n m : nat) : nat :=
2         match n with
3         | O    => m
4         | S n' => match m with
5                   | O    => n
6                   | S m' => S (max n' m')

8     Definition max (n m : nat) : nat :=
9         nat_rect (fun _ => nat -> nat)
10            (fun m => m)
11            (fun n' rec m =>
12                nat_rect (fun _ => nat)
13                    (S n')
14                    (fun m' _ => S (rec m'))
15                m)
16        n m
```

Figure 4.4: A Gallina function for computing the maximum of two natural numbers and its equivalent written in terms of `nat_rect`.

with the effectful real world by performing I/O and other operations not possible in Gallina. Figure 4.5 shows C code for a program that reads two unsigned integers and computes their maximum by calling `max`. Given these two inputs, Œuf compiles both to a common representation, links them, and invokes the CompCert backend to produce an executable.

To begin the compilation process, the user must ensure that their function is written in a form that Œuf can reflect. The design of Œuf requires that the input function be written in an "ML-like" subset of Gallina, free of dependent types. This is required for compatibility with the computational denotation described in Section 4.4.3. Our prototype implementation further requires that all recursion and pattern matching be expressed using recursion schemes (also called *eliminators*). For a complete description of Œuf's input language and the rationale for its restrictions, see Section 4.4.

The function `max` in Figure 4.4 is already in the required format for reflection. A more idiomatic version of `max`, which an actual user would likely write first, is also given as `max_orig` in that figure. Since our prototype implementation does not automatically convert programs to use eliminators, the user must manually perform the translation and prove that the converted function is equivalent to the original; as both are ordinary Gallina functions, this proof can be carried out in the standard fashion.

Given a Gallina function in the proper form, the first step in the Œuf compilation process is to reflect the function into an abstract syntax tree (AST) suitable for processing by the Œuf compiler. Œuf's reflection procedure is implemented as a Coq plugin. Invoking the reflection procedure, as shown in Figure 4.6, generates a value of type `compilation_unit` that contains a set of functions ready for processing by the rest of the Œuf compiler.

The Œuf reflection procedure need not be trusted: it is verified using translation validation [106, 114]. After constructing the Œuf compilation unit, the user invokes a script to generate a proof of equivalence between the denotation of the reflected function and the original function the user intended to compile. For simple functions, such as `max`, the proof is trivial, as shown in Figure 4.6; more complex functions require a more complex proof structure to work around performance problems in the Coq proof checker. The denotation

```
1 oeuf_function max;

3 int main() {
4     unsigned int n, m, result;
5     coq_nat *n_, *m_, *result_;
6     scanf("%u %u", &n, &m);
7     n_ = nat_of_uint(n);
8     m_ = nat_of_uint(m);
9     result_ = OEUF_CALL(max, n_, m_);
10    result = uint_of_nat(result_);
11    printf("%u\n", result);
12    return 0;
13 }
```

Figure 4.5: A C program that calls the `max` function defined in Figure 4.4.

```
1 Oeuf Reflect max As max_ast.
2 Check max_ast : compilation_unit.

4 Lemma max_ast_validate : denote max_ast = max.
5 Proof. reflexivity. Qed.

7 Oeuf Eval lazy Then Write To File "max.oeuf"
8     (compilation_unit.print max_ast).
```

Figure 4.6: The Coq Vernacular commands to reflect, denote, verify the roundtrip, and write the reflection to a file of the `max` function.

function also appears in Œuf's correctness theorem, so the validation theorem helps connect the input Gallina function to the compiled code.

Up to this point, all steps of the Œuf workflow have taken place within an interactive Coq session. However, because the Œuf compiler integrates with CompCert, which relies on unverified OCaml code, the Œuf compiler must be executed as a separate binary, produced using Coq's existing extraction process. Using extraction also provides better performance than running inside the Coq interpreter for the Œuf compiler's translation passes.

To transfer the Œuf compilation unit from the interactive session to the Œuf compiler, the user invokes a serialization procedure to convert the compilation unit to a string, then writes the string to a temporary file (labeled `tmp.oeuf` in Figure 4.3). The code for this step is shown in Figure 4.6.

Once the compilation unit has been written to disk, the user can invoke the Œuf compiler binary to complete the compilation process. The compiler takes the compilation unit and shim source as input. It then deserializes the compilation unit. The deserializer is verified to be the inverse of the serializer: $\forall x$, deserialize(serialize(x)) $= x$. Next, the compiler translates the compilation unit to Cminor, a simplified C-like intermediate language used in CompCert, using the process described in Section 4.5. It similarly compiles the shim from its C source code to Cminor and links the two build products. The resulting complete Cminor program is translated to object code with the existing CompCert backend.

4.3.1 Guarantee

Œuf's correctness theorem guarantees that valid calls to Œuf-compiled functions will behave in a manner equivalent to the user's original Gallina implementation. A valid call is one that follows the Œuf application binary interface, which defines a relation $\sim$ between Gallina values and assembly-level memory representations of those values. Roughly stated, Œuf guarantees that "applying related functions to related arguments produces related results." That is, for any Gallina function f and argument x, and any assembly-level function f' and argument x', if $f \sim f'$ and $x \sim x'$, then $f(x) \sim f'(x')$.

This theorem is established in several steps.

1. First, when Œuf reflects the original Gallina functions to produce the input AST, it also generates a proof equating the denotation of the AST to the original function, as discussed above.

2. Second, Œuf comes with a proof that the denotational and operational semantics of its source language are compatible. If the operational semantics permit a step from state s to state s' (written $s \to s'$), then the denotations of the two states are equivalent ($[\![s]\!] = [\![s']\!]$).

3. Third, Œuf uses a verified serializer and deserializer when writing the input AST to the temporary file and reading it back. This ensures that the same AST that was reflected interactively is processed by the Œuf backend.

4. Fourth, the Œuf compiler comes with a proof of simulation between an operational semantics for the Œuf input language and the operational semantics of Cminor.

5. Finally, Œuf relies on CompCert's own simulation proof to connect the behavior of the Cminor program to that of the final assembly program.

The complete details of Œuf's correctness proof are covered in Section 4.6.

4.4 Front End

We begin with a discussion of the design constraints on Œuf's source language, emphasizing the requirement that the language have a computational denotational semantics in Coq. Then, we present the syntax of the source language, which is a lambda-lifted simply typed lambda calculus over a set of predetermined base types. Finally, we describe how to denote and reflect the source language, and emphasize how our design simplified early compiler passes by enabling us to implement them in the untrusted reflection and automatically ensure their correctness *a posteriori* using the computational denotation.

4.4.1 Design Constraints

Œuf's source language must satisfy several constraints, some of which are in tension.

Reflection. It must be possible to *reflect* Gallina terms into corresponding ASTs. Reflection is the first step of the compiler pipeline. However, we need not trust reflection directly, as we can check its result later using denotation.

Denotation. The language must support a *computational denotation*, *i.e.*, a function defined in Coq that converts the syntax of an expression into the corresponding Coq term. This is essential in order to formally connect verified Gallina programs with their representation as an AST. The denotation need not be trusted, since it is verified against the operational semantics of the input language.

Expressiveness. We would like to maximize the subset of Gallina programs we can compile. This is in direct tension with the requirement for a computational denotational semantics. While there exist many different possible tradeoffs, we believe the input language for Œuf is reasonably expressive.

Tractable Compilation. Lastly, it should be possible to write and verify (in Coq) a compiler from the source language to a low-level language.

Since our Œuf prototype compiler is a mainly intended as a proof-of-concept to explore validated reflection and ABI reasoning, its source language satisfies these constraints somewhat conservatively, sacrificing expressiveness for simplicity. For example, the choice to restrict base types to a predefined set means that the language definition and denotation need not handle user-defined types, simplifying the design and implementation at the cost of restricting user programs.

The process of translating a Gallina program into the supported language subset is a largely mechanical process, but it has not been automated in the current implementation of

$$
\begin{array}{lll}
e & \in & \mathit{expr} \quad \text{(Expressions)} \\
e & ::= & x \quad \text{(variable)} \\
 & | & e\ e \quad \text{(function application)} \\
 & | & C\ e^* \quad \text{(data constructor)} \\
 & | & E\ e^*\ e \quad \text{(data eliminator)} \\
 & | & f\ e^* \quad \text{(closure creator)} \\
C & \in & \mathit{constr} \quad \text{(Constructors)} \\
C & ::= & \text{true} \mid \text{cons} \\
 & | & \ldots \\
\tau & \in & \mathit{type} \quad \text{(Types)} \\
\tau & ::= & b \mid \tau \to \tau
\end{array}
\qquad
\begin{array}{lll}
f & \in & \mathit{func} \quad \text{(Function name)} \\
b & \in & \mathit{base} \quad \text{(Base types)} \\
b & ::= & \text{bool} \\
 & | & \text{list}\ b \\
 & | & \ldots \\
E & \in & \mathit{elim} \quad \text{(Eliminators)} \\
E & ::= & \text{bool\_elim} \\
 & | & \text{list\_elim} \\
 & | & \ldots \\
g & \in & f \to e \quad \text{(Global environment)}
\end{array}
$$

Figure 4.7: Syntax of the Œuf prototype compiler's source language.

Œuf. In practice, these restrictions do not limit expressiveness for systems implemented in Gallina since users can easily prove the idiomatic version of their Gallina code equivalent to the corresponding eliminator-based version.

4.4.2 Syntax

Figure 4.7 presents the syntax of the Œuf prototype compiler's source language, which is a lambda-lifted simply typed lambda calculus over a particular set of base types. Variables and function application are standard.

The next two expression forms are for manipulating data of base type. Base types consist of a pre-defined set of types, including common types in the Coq standard library, such as booleans, lists, natural numbers, and so on. The Œuf source language supports parameterized types, such as lists, by making the definition of base types recursive; however, since the

recursion is not mutual with the definition of types, thus function types as arguments to type constructors are not supported. We believe this limitation could be lifted with a little engineering effort, but it simplifies the implementation.

In the expression grammar, a constructor C is a data constructor for one of the base types, *e.g.*, true for bool. A constructor has some number of argument subexpressions, written e^* in the figure.

An eliminator E represents a structural recursion principle for each base type. An eliminator for a particular base type takes a list of expressions representing "cases", one for each constructor of the base type, and then takes one last expression, which is the "target" of the elimination. The Œuf prototype's source language encodes all recursive functions explicitly in terms of eliminators instead of a fixpoint construct. Eliminators were chosen instead of recursive functions due to their ease of denotation.

The last expression form is closure creation. A closure creation expression consists of a function name f and list of subexpressions, whose values will serve as the closure's environment. At run time, a closure creation expression evaluates each of its subexpressions and produces a closure value that contains the function name and the resulting values.

Since the Œuf prototype's source language has explicit closures, a top-level program is not simply an expression; instead, it is a global environment containing a mapping from function names to function definitions. Each function definition consists of a single expression that makes up the function's body. The body expression is not closed: it has one free variable corresponding to the function's sole argument, plus zero or more free variables provided by the closure environment.

4.4.3 Denotational Semantics

The key property of the source language is the ability to write a computational denotation function. The denotation function is a normal Coq function which takes the syntax of an expression and returns the corresponding Coq term. We use the standard technique of typed dependent de Bruijn indices to represent the syntax of binding (see, for example, Chlipala's

CPDT [27]). In this style, the syntax of expressions carries the relevant typing information, which ensures that only well-typed expressions can be created. The syntax of types follows Figure 4.7 directly.

Expressions are represented as an inductive type family indexed on the types of free variables and the return type. Our representation of variables and application is standard and follows CPDT. We make modest extensions to standard techniques to support constructors, eliminators, and closures. We represent constructor names and eliminator names as elements of a type family indexed by their arguments and return types. Constructor and eliminator expressions use dependent types to enforce correct argument types.

A closure is represented with a reference to the body, and a list of expressions whose values will make up the closure's environment. An invariant of closure expressions is that the types of values in the environment match those of the free variables in the body of the closure.

These extensions to standard typed de Bruijn techniques pose no fundamental difficulties in the computational denotation function. First, each type is denoted to a corresponding Coq type; for example, if the syntax for the boolean type was `ty_bool`, the type denotation function would map this to Coq's `bool` type. Similarly, ASTs which represent function types denote to native Coq function types. Expressions are denoted by a function mapping expression syntax into the corresponding Coq term. The expression denotation function has a return type which uses the type denotation function, which captures the idea that an expression in the object language should correspond to a Coq term with a corresponding type. Expression denotation takes additional arguments to represent the environment in which to denote, including the available top-level functions and the values for free variables. The denotations of variables and application are standard. The cases for constructors and eliminators branch on the particular constructor or eliminator used and return the corresponding Coq constructor or eliminator.

The denotation of closures requires some care, since Gallina has no explicit notion of closures. We handle this by giving each denoted function an additional initial argument

which packages up its free variables. Then, when constructing a closure, we partially apply the denoted function to a structure containing the denotation of the closure's environment expressions. To access a free variable, the body of a closure indexes into this structure with the corresponding variable name.

4.4.4 Operational Semantics

In addition to the denotational semantics, we also give an *operational* semantics for the source language. These and additional operational semantics for each intermediate language are used to verify semantics preservation later for each pass of the compiler. In order to connect the compiler guarantees to the original Coq term, we prove a theorem relating the denotational and operational semantics, namely that each step of the operational semantics ($\rightarrow$) preserves the denotation ($[\![\cdot]\!]$): $\forall\ s\ s',\ \ s \rightarrow s' \Rightarrow [\![s]\!] = [\![s']\!]$. Together with the CompCert compiler's correctness theorem, this guarantees that values computed at the machine level correspond to the original Coq terms.

Our semantics handles local variables using an explicit local environment, which we chose because it simplifies later reasoning. This decision also makes the handling of closures fairly straightforward: calling a closure amounts to replacing the expression being evaluated with the body of the function and replacing the current local environment with a combination of the argument value and the closure's environment values. The only wrinkle is that retrieving a function body produces an expression that is valid in a restricted global environment, containing only the functions defined prior to the function of interest, whereas evaluation occurs only in the full environment. We handle this using a computational version of the standard weakening lemma, a function of type `expr G L ty -> expr (G' ++ G) L ty` that converts an expression into an equivalent one that is valid in an extended global static environment.

4.4.5 Reflection

Another important aspect of the source language is that it is possible to *reflect* a Coq term into the corresponding syntax. Of course, reflection will only succeed for Coq terms in the image of the denotation function; Coq terms that use features not supported by the source language (*e.g.*, dependent types) will simply fail to reflect. We implemented the reflection procedure for the source language as a custom OCaml plugin to Coq. The reflection procedure is relatively straightforward and uses standard techniques.

The most important aspect of the reflection procedure is that we *need not trust it*. After reflection, the generated syntax can be checked by denoting it and ensuring that it is definitionally equal to the original Coq term. This is essentially a form of translation validation. This design decision significantly eases the proof burden for the early passes of the compiler. Performing lambda lifting during reflection allows us to avoid implementing a complex verified lambda-lifting pass in Coq. We are also able to extend the reflection with support for monomorphizing invocations of polymorphic functions, allowing our Œuf prototype compiler to support most instances of ML-style polymorphism with no modification to the source language and minimal verification effort.

The fact that reflection is not trusted is a key difference between our approach and related work, but it comes at the cost of supporting a restricted subset of Gallina. Even aside from the decisions we imposed to simplify the development of our prototype, the design of Œuf requires that the source language support computational denotation into Gallina imposes significant restrictions—in particular, it remains an open research question whether it is possible to denote a general Gallina AST into a Gallina term, all within Gallina.

However, we believe that with some engineering effort, it will be possible to extend the source language to include user defined datatypes, pattern matching, and recursion, which will capture typical verified systems implementations.

4.5 Compiler Internals

The bulk of our effort developing the Œuf prototype compiler was implementing and verifying 45 translation passes from our front end down to CompCert Cminor. To ease verification effort, we followed the standard practice of building many passes between closely related languages to simplify the refinement proof between each stage [83, 118].

4.5.1 From Gallina to Register Machine

Type Erasure Œuf first erases type information. The input language uses a dependently-typed program representation that admits only well-typed terms, which is necessary to write the denotation function used for validating the reflection procedure Section 4.4. Because the Œuf compiler includes simulation proofs for every translation pass, the untyped intermediate programs are guaranteed to behave identically to the well-typed input program.

Eliminators Next the Œuf compiler contains 6 passes to translate inductive datatype eliminators to recursive functions. These passes collectively replace each eliminator expression with a call to a new top-level function that performs a `switch` on the argument's constructor tag and executes the appropriate arm. If a constructor contains further values of the same argument type, the eliminator function will generate recursive calls to itself, executing the same pattern-matching code on the structurally smaller values.

Stack Machine The Œuf compiler then converts nested expressions to flat sequences of stack machine operations. We use this stack machine language as an intermediate step, allowing us to separate flattening of the program from assigning of names to temporary values. There is generally one stack machine operation for each expression type, whose behavior is to pop values corresponding to any subexpressions, perform the same operation as the original expression, and push the result back onto the stack. Execution of this stack machine representation directly mirrors the order of evaluation of the original expression, but makes explicit the creation and use of intermediate values.

Register Machine Once manipulation of intermediate values has been made explicit, the Œuf compiler converts the program to a high-level register machine program that gives an explicit name to each value. The set of operations for the register machine is nearly identical to that of the stack machine, except that each operation now explicitly reads and writes named registers instead of manipulating an implicit stack. The pattern of register accesses corresponds closely to C-level local variable accesses, except that programs still manipulate abstract "closure" and "constructed" values rather than machine-level integers and pointers.

4.5.2 Lowering towards Cminor

At this point the program is in a language which computes using closures and constructors. However, to meet with any language from the CompCert pipeline, the program must use integers and pointers, i.e. C level values.

Switch Statements To transform a program from a high level functional language to a lower level procedural language, the Œuf compiler generates C-style `switch` statements, with explicit breaks and implicit fall through. In earlier phases, a single step takes a switch statement into the corresponding correct case; in this phase the transition may take multiple steps in order to "step over" all non-chosen cases. While this is conceptually straightforward, the proof was quite subtle, due to the arbitrary number of steps the target program may take, as well as the relatively strong invariants which must be true about the current program continuation during all of these steps.

Heap Introduction The Œuf compiler next transforms programs to use heap-allocated memory, which requires introducing memory into the program state. In this phase, explicit allocations and stores are inserted to construct well formed closure and constructor values. The simulation relation for this phase maps higher level values to their lower level memory representation: where the higher level program would produce a value, the lower level program produces a pointer whose contents correspond to the high level value.

Stack Introduction The next major step is stack introduction, where allocations and frees of runtime stack blocks are introduced. While the compilation strategy is straightforward (since stack blocks are never read nor written), the correctness proof is subtle and long due to the relatively complex relation of the unified, high-level heap memory to the split, lower-level heap and stack memories.

Backend Once the heap and stack are introduced, only minor changes are necessary to compile the program to a subset of Cminor. Afterward, the Œuf compiler relies on existing CompCert passes to compile the program from Cminor down to any of CompCert's assembly backends.

4.6 Shim

Œuf provides verified compilation of pure Gallina functions to assembly code. But real-world applications are not completely pure: they read inputs from disk, communicate over the network, or print results to the terminal. For any extraction or compilation mechanism, including Œuf's, to be useful, it must allow integrating the resulting code with a *shim* that connects the pure functions with impure side effects. For instance, one might verify and compile a pure function for computing the maximum of two numbers, then link that with a shim that reads two numbers from the terminal, invokes the compiled function, and outputs the result. When using Coq's built-in extraction mechanism, the shim is an OCaml or Haskell program; with Œuf, it is written in C.

Most verification projects do not bother verifying the shim code. Indeed, there is limited value in verifying a program whose main purpose is to call into code generated by unverified extraction and which itself will be compiled with an unverified compiler. With Œuf, however, stronger guarantees are possible. Œuf's compilation process is verified, and thus it is possible to establish strong correctness properties about the compiled code. Furthermore, Œuf is designed for use with shims written in C, which can be verified against the CompCert C semantics (e.g., using VST [8]) and then compiled with CompCert.

The remainder of this section describes aspects of Œuf's design that enable verification of complete programs, consisting of compiled code together with a C shim. Section 4.6.1 describes Œuf's primary correctness theorem, which is designed for reasoning about calls spanning the boundary between the shim and the compiled code Section 4.6.2 gives the complete definition of the Œuf application binary interface (ABI), including the definitions of several relations that are mentioned in the correctness theorem. Finally, Section 4.6.3 describes the assumptions currently relied on by Œuf's proofs.

4.6.1 Correctness Theorem

Œuf's correctness theorem is the primary interface for users who are verifying a shim program. Like most compiler correctness theorems, it relates the behavior of the high-level input source program (in this case Gallina) to the low-level output target program. However, we have been careful to state the theorem in terms that will be meaningful and relevant to the shim verifier. In particular, we state the theorem using Cminor as the low-level language, rather than any of CompCert's assembly languages, as we expect users will prefer verifying their shims at this higher level. If needed, one could directly compose Œuf's correctness theorem with the CompCert backend correctness theorem to obtain the equivalent theorem for CompCert assembly.

The correctness theorem relies on three Œuf-specific relations, called *value-matching*, *callstate*, and *returnstate*. The details of these relations define the ABI by which shims may interact with Œuf-compiled code. We describe each relation briefly, leaving their full definitions to the next subsection.

The value-matching relation $v \sim v'$ relates a high-level Gallina value v to its low-level memory representation v'. This is unlike the design of CompCert, which uses the same value representation at all levels and thus can use simple equality or "more-defined-than" to relate values in the input and output programs. Here, the high-level values are built from inductive data type constructors, such as the natural number `S (S (S O))` (representing 3), while low-level values are concrete integers and pointers in a particular machine-level memory

layout.

The *callstate* and *returnstate* relations describe a particular subset of program states that stand in relation to values. We say that a program state is a *callstate* for closure f and argument a if it represents the point in execution "during the function call" of f applied to a, when control has transferred to the callee's code, but none of that code has executed yet. A program state is a *returnstate* carrying value r if it represents the point in execution "during the function return", with r as the value being returned. The precise details of these definitions depend on the language in question, and in fact, each intermediate language in the Œuf compiler has its own definition of callstates and returnstates. This is necessary because the structure of functions, calls, and returns changes as the program is transformed from an expression tree down to low-level sequential code. The correctness theorem refers to callstates and returnstates at the Cminor level, where these relations describe the calling convention of Œuf-compiled code. The statement of Œuf's correctness theorem is as follows:

Theorem 4.6.1. *Let s' be a Cminor callstate, representing an application of the Cminor closure value f' to the Cminor value a'. Suppose there exist Gallina values f and a such that $f \sim f'$ and $a \sim a'$ and the application $f(a)$ is well-typed in Gallina. Then there exists a unique Cminor returnstate t' carrying a value r' such that $f(a) \sim r'$ and the Cminor state s' will always reach t'.*

This theorem establishes total correctness of Œuf-compiled functions. Once the program begins performing the call to a compiled function, it is guaranteed that the callee will eventually return, barring exceptional conditions such as memory exhaustion. This is possible because all Gallina functions are terminating, and compiled functions behave identically to their original Gallina counterparts.

The proof of Œuf's correctness theorem is structured in five parts, illustrated in Figure 4.8. First, callstate matching: since s' is a Cminor callstate for closure f' and argument a', $f \sim f'$, and $a \sim a'$, there must exist a Gallina callstate s for f and a, which matches s' using a state-matching (or simulation) relation of the type commonly used in compiler correctness

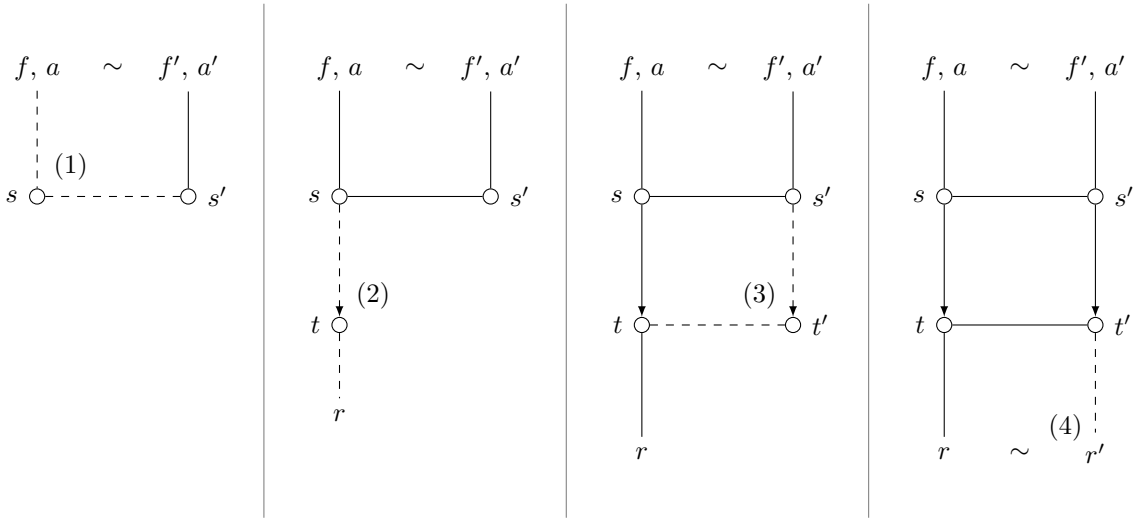

Figure 4.8: The structure of the Œuf compiler correctness proof. Given Gallina values f, a matching Cminor values f', a' and Cminor callstate s' for f' and a', we prove that (1) there exists a Gallina callstate s for f and a that matches s'; (2) s steps to some Gallina state t that is a returnstate for $r = f(a)$; (3) s' steps to a unique Cminor state t' such that t' matches t; and (4) t' is a returnstate for some r' that matches r.

proofs. Second, termination: since all Gallina functions are terminating, there must exist a result value $r = f(a)$ and a Gallina returnstate t carrying r, where s takes zero or more steps to reach t. Third, forward simulation: since s is related to s' and s steps to t, there must exist a Cminor state t' matching t, where s' takes zero or more steps to reach t'. Fourth, returnstate matching: since t is a Gallina returnstate carrying r and t' is a Cminor state matching t, t' must be a Cminor returnstate carrying a value r', where $r \sim r'$. Fifth, backwards simulation via determinacy: since Cminor is deterministic, backwards simulation follows from forwards simulation. We have proved the first, third, and fourth parts in Coq; see Section 4.6.3 for details on our assumptions in the second and fifth parts.

Aside from termination, which is relevant only at the Gallina level, we prove the necessary lemmas by composition. Each transformation pass in the compiler includes proofs of callstate matching, forward simulation, and returnstate matching between its input and output languages. These separate proofs are composed to establish the end-to-end properties relating Gallina to Cminor.

The theorems above ensure that an Œuf-generated Cminor function f' can always simulate the behavior of the corresponding input Gallina function f, and furthermore, all such behaviors will satisfy any theorems proved about f. To ensure that f' never exhibits any "extra" behaviors not possible for f, we adopt the approach used in CompCert and prove our target language, in this case Cminor, is deterministic. Note that (like CompCert), we assume that no program will exhaust memory. For a more in depth exploration of this assumption, we suggest prior work [93].

4.6.2 The Œuf ABI

The Œuf application binary interface (ABI) describes the means by which shim code written in C can call into Œuf-compiled functions and preserve any guarantees established at the source Gallina level. The ABI consists of a Cminor-level representation of Gallina values and a calling convention for invoking Œuf closures. These aspects of the ABI are captured in the definitions of the value-matching, callstate, and returnstate relations used in the compiler's

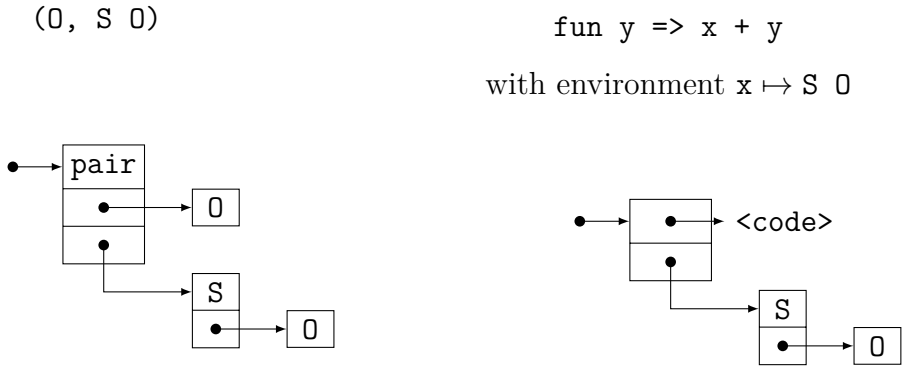

Figure 4.9: Example Cminor-level memory representations of Gallina values. The left shows the representation of the pair `(O, 1)` (represented in unary). The right shows the representation for a closure whose environment contains a single value (for the free variable `x`).

primary correctness theorem. The theorem itself provides guarantees about the results of valid calls made according to the Œuf ABI.

Value Representation There are two kinds of values in the subset of Gallina supported by the prototype compiler: values of inductive data types, which we call "data", and closures. Our prototype uses a pointer to a sequence of machine words as the low-level value representation in both cases, so all Œuf values are one machine word in size when stored in registers, on the stack, or within another data structure. For data values, the first word of the allocation is an integer tag, indicating which constructor was used to build the value, and each remaining word is another Œuf value, comprising the arguments to the constructor. For closure values, the first word of the sequence is a function pointer, and the remaining words make up the environment of the closure. Figure 4.9 shows examples of the memory representations for some simple values.

Given these definitions, shims for use with our prototype are permitted two ways of constructing Œuf values. First, the shim can create a data value from a constructor tag and a sequence of argument values. It does so by allocating $1 + n$ words, where n is the number

of arguments to the constructor, and filling the allocated storage with the constructor tag followed by the values. (Recall that all Œuf values are one word in size.)

Second, the shim can create a closure value for a function f. This is permitted only when f has no free variables—in other words, it must be a top-level input to the compiler, not a lifted lambda. Since the environment of the closure is always empty, the shim allocates exactly one word and writes the function pointer into the allocated memory. There is nothing strictly preventing the shim from building closures for lifted lambdas with nonempty environments, but the Œuf prototype compiler preserves the names only of top-level functions, so it is difficult to refer directly to a specific lifted lambda.

The shim, being an arbitrary C program, could of course perform other operations on Œuf values. Most notably, the shim might construct a closure whose code pointer refers to a C function defined by the shim itself. However, such a closure would not be related to any Gallina value under the value-matching relation, so the compiler's correctness theorem would provide no guarantees for operations that use such a value. This property captures the fact that some C functions, particularly those that use side effects, do not admit any Gallina implementation. Similarly, if the shim were to construct a cyclic data value at the Cminor level, it would not relate to any Gallina value (as Gallina inductive data must be finite), and the result of passing the illegal value to any Œuf function would be unspecified. The burden of reasoning about such unsupported manipulations of Œuf values is left to the user.

Calling Convention The shim can invoke Gallina functions compiled with the Œuf prototype compiler using a straightforward calling convention: given a closure `f` and an argument `a`, the shim should dereference `f` to obtain a function pointer `p`, then call `p(f, a)`. Since C supports only bare function pointers, not closures, the caller must provide `f` explicitly so the closure code can access the environment. The result of applying the closure to the argument is returned directly from the call to `p`.

There are two additional points to note regarding the practical use of this calling convention. First, this convention supports only a single argument for each function. This is in line with

the definition of the Œuf prototype compiler's input language, which currently supports only single-argument lambdas. Multiple-argument functions can be implemented using currying, and they can be invoked by performing multiple individual call steps following this convention. Second, for uniformity, the prototype compiler only supports calls made through a closure object. If the shim needs to call a top-level function directly (as opposed to applying a closure obtained from some previous call), it must construct a closure object for the function as described above, then apply the closure as normal. We believe it would be straightforward to lift these restrictions when developing a more feature-complete compiler.

4.6.3 Verification Assumptions

To verify a complete program, including the interaction between Œuf-compiled code and the surrounding Cminor shim requires the following two additional facts.

1. Œuf-compiled functions will never modify any values in previously allocated memory.

2. The source Gallina program passed to Œuf terminates.

3. Forward simulation plus determinacy implies backwards simulation.

Fact (1) is necessary for carrying any properties *across* calls into Œuf-compiled code; otherwise, verifiers would be forced to assume that any properties established before the call could become invalid after the call. We are confident that this is true because Œuf-generated code only writes to small constant offsets into freshly allocated memory blocks. However, we have not yet formally proved this fact.

Fact (2) is true metatheoretically, but we need to manifest it inside of Coq. This amounts to proving normalization of our operational semantics for the compiler's reflected source language. More formally, given any initial source program state s, we assume there exists a source program state t such that s steps to t in zero or more steps and t is a final state, meaning no further evaluation is possible. We use this fact in our proof of Œuf's primary

correctness theorem (Theorem 4.6.1). We have already proved a progress lemma (every non-final state can take a step), so the only remaining doubt is that we may have defined the semantics in a way that admits infinite sequences of steps through non-final states. We believe this is unlikely because our definition of the semantics largely follows standard practice, but we have not proved that no such infinite sequence exists. The proof could likely be conducted using the standard technique of logical relations [116].

Fact (3) is also true metatheoretically, and in fact CompCert proves it as a lemma for their semantic framework. However, Œuf's semantics cannot directly use this lemma because they do not fit into CompCert's semantic framework.

In the future, Œuf implementations may provide libraries proving these facts; however, in our current prototype we assume the shim implementer dispatches these obligations.

4.7 Trust

This section takes a more thorough look at the TCB (trusted computing base) of programs compiled with Œuf. In particular, we carefully consider how our reliance on the (unverified) OCaml compiler and (unverified) runtime differs from the conventional approach to executing Gallina programs.

At first blush, the situation is fairly straightforward: Because Œuf is itself verified in Coq (as is the CompCert compiler whose back-end we reuse), we have a machine-checked guarantee that the x86 assembly Œuf generates is equivalent to the Gallina source code that Œuf consumes, up to the usual assumption that CompCert's model of x86 assembly is adequate. Given this equivalence, any Coq proofs about the source program apply also to the binary, provided shim code follows our specified ABI.

Remaining in the TCB are components that are typical in the development of formally verified systems: (1) the consistency of Coq's logic and the correctness of Coq proof checking; (2) the shim code and any libraries it uses (e.g., the C standard library); (3) the underlying execution environment (e.g., the operating system and hardware).

The first of these is the most interesting because Coq itself is implemented in OCaml and

so the proof-checker's creation relied on the OCaml compiler and the proof-checker's execution relies on the OCaml run-time system. Therefore, in this sense, Œuf-compiled code still relies on OCaml's implementation, but the situation compared to extraction-of-Coq-to-OCaml is qualitatively different: A compiler or run-time bug cannot be exploited on Œuf-compiled code unless it causes the Œuf compiler itself to be incorrect in such a way that compilation appears to succeed but in fact is incorrect. That is, a security hole would require OCaml miscompiling the Œuf-compiler source-code in such a way that Œuf then miscompiled another Coq program in an exploitable way.

Would it help to compile the Œuf compiler with a verified compiler too, perhaps Œuf itself (i.e., bootstrapping our compiler)? In a practical sense, yes, since each level of a verified compiler compiling another verified compiler would make a false claim about the eventual program compiled by Œuf that much less likely. But on a theoretical level, such a sequence of verified compilers is exactly the focus of Thompson's famous lecture on, "Trusting Trust" [119]: In principle every compiler in the sequence of compiler bootstrapping steps back to the dawn of computing is in the TCB. In theory, Œuf is not immune to this classic argument.

In contrast, most prior work on verified systems relied *directly* on Coq's built-in extraction, despite the extraction procedure being complex, unverified, and known to contain bugs, including some that can cause incorrect code generation. Our Œuf compiler does not directly suffer from these problems unless a Coq-extraction or OCaml implementation bug causes the Œuf compiler itself to be miscompiled. Any Coq/OCaml bugs that do not affect the compilation of Œuf are irrelevant to Œuf's correctness. As for programmers' proofs about code that Œuf compiles, Coq's extraction mechanism is not relevant, but the implementation of Coq proof-checking in OCaml remains trusted.

4.8 Native types

Œuf supports customization of low-level data representation and code generation, a feature we call "native types," which enables improved performance for generated code without compromising the compiler's correctness guarantees. Users can extend the compiler with new

native type definitions, which map ordinary Gallina types to custom data representations, and with *native operations*, which map Gallina functions to arbitrary Cminor code. For example, we have extended Œuf to represent CompCert's 32-bit `Int.int` type as a real 32-bit machine word, rather than using heap-allocated tagged unions as in the ordinary translation of inductive types.

4.8.1 Design constraints

We had three primary goals when designing Œuf's native types feature.

Correctness First, the addition of native types must not compromise Œuf's compiler correctness guarantees. Every native operation must include a proof of correspondence between the original Gallina function definition and its low-level implementation as custom Cminor code. Furthermore, the new source-language constructs introduced by the native types feature must be compatible with the translation validation step that Œuf uses to achieve verified reflection.

Extensibility Second, the native types feature must be extensible to new types and operations. The Œuf compiler uses a large number of compiler passes, so adding a new feature normally requires updating dozens of compiler passes and correctness proofs, each with a slightly different state-matching relation and proof structure. But requiring extensive compiler modifications for every new native type or operation definition would severely limit the usefulness of the native types feature. We thus sought a design where native type and operation definitions are self-contained, not spread across dozens of IRs and compiler passes.

Mixed operations Third, Œuf must support native operations that take values of both native and non-native types. We intend for the native types feature to be used to optimize only performance-critical elements in the input program, leaving other types and functions to use the standard translation. This requires a mechanism for interoperation and conversion between native and non-native values, which mixed operations can provide.

$$
\begin{array}{rcll}
e & \in & \mathit{expr} & \text{(Expressions)} \\
e & ::= & \ldots & \\
 & | & O\ e^* & \text{(native operations)} \\
O & \in & \mathit{native\_oper\_names} & \text{(Native operation name)} \\
\tau & \in & \mathit{type} & \text{(Types)} \\
\tau & ::= & \ldots \mid N & \\
N & \in & \mathit{native\_type\_names} & \text{(Native type name)}
\end{array}
$$

Figure 4.10: Extension of the Œuf prototype compiler's source language with support for native types

4.8.2 Generic implementation

Concretely, we made the native types feature extensible by adding generic "Native" variants to Œuf's type, value, and expression representations, as shown in Figure 4.10. Each of these constructors takes a `native_type_name` or `native_oper_name` (each a simple enumeration, to permit de/serialization and decidable equality), and a function external to the semantics maps each "name" value to a dependent record containing all necessary implementation code for the named type or operation. The implementation record also contains proofs necessary to establish compiler correctness—for example, proofs of correspondence between high- and low-level implementations of each native operation.

Native type definitions

Defining a new native type—that is, a new low-level data representation for some Gallina type—consists of adding a `native_type_name` variant for the new type, providing an implementation record containing the high-level and low-level representations of the type and relevant proofs,

and updating the reflection procedure to represent the original Gallina type using the new native type definition. The verification effort is concentrated in the definition of the implementation record, which includes several lemmas necessary to establish compiler correctness for programs using the native type.

Figure 4.11 shows a partial definition of the native type implementation record (`native_type_impl`) and an example instantiation for CompCert's `Int.int` (`int_impl`). A native type definition begins with the Gallina type `nt_gallina` that is being given a custom data representation. For `int_impl`, `nt_gallina` is simply `Int.int`. Next, the user defines the data representation itself, in the form of a relation `nt_repr` between an instance of the high-level representation and a low-level machine value and memory state in the CompCert memory model. For `Int.int`, this is simple: a high-level value `hv` is related to the low-level value `Vint hv`, and the memory is unconstrained (as the low-level value involves no pointers). No other representations are needed, as Œuf uses the Gallina representations of native-type values in all but the last few IRs, and uses the provided low-level data representation for the rest. Finally, the user proves a number of properties about the data representation relation, such as that any low-level value in the relation is defined (*i.e.*, not `Vundef`). Œuf's main compiler correctness proofs rely on the proof fields of the definition record to establish correctness of programs that use native types.

Native operation definitions

Defining a new native operation—a custom low-level implementation for some Gallina function—is similar to the process of defining a new native type. The user must add a `native_oper_name` variant, provide an implementation record for the operation, and update the reflection procedure to use the new native operation in place of calls to the original function. The verification effort is again primarily in the implementation record, which is more complex for native operations than for native types.

We omit even a simplified definition of the native operation implementation record (`native_oper_impl`), as it depends on numerous auxiliary definitions not given in this

```
1  Record native_type_impl := {
2      nt_gallina : Type;
3      nt_repr : nt_gallina ->
4          Cminor.val -> Cminor.mem -> Prop;
5      nt_defined : forall (hv : nt_gallina)
6              (v : Cminor.val) (m : Cminor.mem),
7          nt_repr hv v m ->
8          v <> Vundef;
9      (* 4 fields omitted *)
10 }.

12 Definition int_impl : native_type_impl := {|
13     nt_gallina := Int.int;
14     nt_repr := fun (hv : Int.int)
15             (v : Cminor.val) (m : Cminor.mem) =>
16         v = Vint hv;
17     nt_defined := (* proof term *);
18     (* 4 fields omitted *)
19 |}.
```

Figure 4.11: Partial type definition for the native type implementation record, and an instance for the `Int.int` type.

document. At a high level, the definition of a native operation consists of

1. a Gallina implementation of the operation,

2. additional implementations suitable for use in lower-level Œuf IRs,

3. simulation proofs relating implementations, for use in Œuf's compiler correctness proofs, and

4. miscellaneous lemmas stating properties of specific implemenations.

These are similar to the elements required in a native type definition, but native operations require many more implementation variants than native types do. This stems from the requirement that native operations be able to manipulate non-native values: the representation of non-native values changes slightly at various points in the compiler, and the native operation definition must include a separate implementation giving the behavior of the native operation under each of those value representations.

The native operation record's simulation proofs are responsible for the bulk of the verification effort for each new native operation. Specifically, each of these proofs states that a higher-level variant of the operation behaves equivalently to a lower-level variant under a particular relation on high-level and low-level values, which is required to show overall compiler correctness for programs that use this native operation. Notably, at the lowest levels, the simulation proofs rely on the relevant data representation relations in the case where the argument and/or return values are of native types. The remaining proofs establish that the various implementations behave "reasonably"—for instance, that an invocation of the operation that succeeds in one environment will also succeed in the environment where some global definitions have been (consistently) renamed. Certain passes inside the Œuf compiler rely on these properties for correctness.

Reflection

The final step in defining a native type or operation is to add support for that type or operation to Œuf's reflection plugin. The plugin pattern-matches on Coq's internal representation of Gallina terms to generate corresponding ASTs in Œuf's input language, and extending this code with additional cases allows generating native types or native operation expressions where appropriate. For example, the call `Int.add x y` should be reflected as an invocation of the native operation `int_add`, not as an ordinary call to the reflected function `Int.add`. These pattern-matching cases are not verified *a priori*, but the translation validation step described in Section 4.4.5 will fail if the denotation provided in the native type or operation definition record does not match the Gallina type or expression being reflected.

4.8.3 Design tradeoffs

While using generic definitions simplifies the task of verifying this extension of the Œuf compiler, making the syntax and semantics fully generic over native type definitions means that the constructor and eliminator expressions built in to Œuf's source language cannot be extended to operate on native types as they do on ordinary inductive types. Instead, construction and elimination of native-type values must be implemented using native operations that produce native values from inductive ones or vice versa.

One significant choice we made in the design of Œuf's native types feature was the decision to equip native operations with one implementation variant per value representation, rather than one variant for each IR used in the compiler. The Œuf compiler internally uses several dozen IRs, each with slightly different syntax and semantics, but each IR uses one of only six distinct representations of Gallina values. Requiring native operation definitions to provide only one implementation per value representation simplified the compiler verification work: in nearly all compiler passes, the source and target languages use the same value representation, so the native operation cases in the source and target semantics are identical, and the corresponding cases of the simulation proof are trivial. The alternative, however,

would permit defining useful classes of native operations that are impossible by construction with our current design. In particular, having only one implementation variant for each value representation means that native operations cannot invoke closures that are passed to them as arguments, as this would require manipulating expressions and program states, which have different definitions even between IRs using the same value representation. This prevents defining recursive eliminators for native types as native operations, requiring use of less efficient workarounds.

A further restriction under our current design is that values of native type cannot contain values of non-native (inductive or function) type. This requirement simplifies parts of the compiler correctness proof that need to reason about all values reachable through a particular returned value. However, with this restriction in place, it is impossible to define efficient collections (such as arrays) that can store values of non-native type. It is possible that the restriction could be lifted by extending native type definitions with additional functions and lemmas concerning contained non-native values, but we have not yet investigated this approach.

4.9 Case Study

We tested the effectiveness of the denotation compiler architecture by developing a proof-of-concept compiler for a simple language. This also serves to demonstrate the effectiveness of Œuf for compiling simple Gallina expressions. We further evaluated the usability and performance of Œuf by compiling larger, hand-written Gallina programs.

4.9.1 Proof-of-Concept Compiler

We developed a simple proof-of-concept denotational compiler for the `calc` expression language used in the EPICS dataflow system. The primary purpose of the `calc` language is to implement simple arithmetic expressions, but it also supports conditional expressions, mutable updates of temporaries, sequencing, and variadic `MIN` and `MAX` functions. The `calc` language comes in three variants: the first allows computation over double-precision floating-point values,

Component	**Size (LOC)**
Denotation	150
Correctness proof	279
Specification	450
Shim	23
Native definitions	2960

Figure 4.12: Line counts for components of our `calc` expression compiler.

the second allows computations involving both doubles and strings, and the third computes over doubles and arrays of doubles. We previously specified the syntax and semantics of the double and string variants during our evaluation of the marquer methodology for rapid semantics development, described in Section 3.6. Our goal in the present evaluation, then, was to build a denotational compiler for the double variant of the `calc` language and prove it correct according to the existing semantics.

The design of the denotation was largely straightforward, with one exception: sequencing. In the `calc` language, most expressions produce values (*e.g.*, `B + 2`), but assignment expressions (*e.g.*, `A := 1`) are executed purely for their side effects and produce nothing. A sequencing expression produces a value if one of its two subexpressions produces a value and produces nothing if both subexpressions produce nothing. (If both subexpression produce values, the sequencing expresion produces an error.) Note that either subexpression, not only the last, can produce a value: for example, `A := 1; B + 2` and `B + 2; A := 1` both produce the same value, that of `B + 2`. Our denotation function uses a simple static type system to distinguish value-producing and non-value-producing expressions, allowing it to avoid run-time checks and statically detect errors in sequencing expressions.

Figure 4.12 shows the code size of our `calc` language compiler, specification, and shim. The proof-to-code ratio of the denotation function itself is notably quite low: the correctness

proof is less than twice the size of the denotation. The specification, on the other hand, is comparatively large. This is due to the design of the semantics: a large portion of the semantics is generic over all three variants of the `calc` expression language, while the proof-of-concept denotation handles only a single variant. Our development also includes a simple C shim that runs the compiled code and prints its results.

A substantial portion of our development effort was spent on native type and operation definitions for double-precision floating-point values. As the `calc` language uses doubles pervasively, we built our semantics and our denotation function around an existing Coq formalization of IEEE-754 floating point. However, this formalization, provided by the open-source Flocq library [23], defines doubles using refinement types, which Œuf cannot compile directly. Rather than redesign our denotation to work around this restriction, which would complicate the verification of the denotation, we chose to add support for doubles to Œuf using the native types feature described in Section 4.8.

Specifically, we extended Œuf with native type definitions for boxed (heap-allocated) double values and for fixed-length, heap-allocated arrays of doubles. We also implemented operations for arithmetic, comparisons, and double literals, as well as array allocation, access, and functional update. The native type definitions, including all required proofs, amounted to 126 lines of code, and the native operations totalled 2834 lines. However, like the rest of Œuf, these native type definitions are reusable: future denotational compilers for languages that use doubles can simply copy these definitions, gaining the same benefits we observed here with no additional verification effort.

4.9.2 Œuf Extraction

In addition to compiling the output of our `calc` expression denotation, we tested Œuf's ability to compile two larger, hand-written programs.

`list_max`. Our first test case is a simple program for finding the largest number in a list. The Gallina component is a function `list_max : list nat -> nat`, which we implemented in idiomatic Coq style. We then adapted the function and all of its dependencies to use

Program	**Component**	**Size (LOC)**
`list_max`	Original	5
	Adapted	16
	Proofs	10
	Total	31
`SHA256_N`	Original	200
	Adapt (`N`)	230
	Proofs (`N`)	1400
	Adapt (elim)	700
	Proofs (elim)	650
	Total	3180
`SHA256_int`	Original	200
	Adapt (elim)	300
	Proofs (elim)	860
	Total	1360

Figure 4.13: Approximate line counts for the Gallina components of our test programs. "Original" is the size of the original implementation, written in idiomatic Gallina. "Adapt" is the size of the code after adaptation for compatibility with the Œuf prototype compiler, and "proofs" is the size of the proofs of equivalence between the original and adapted versions. We adapted `SHA256` code twice, once to use the `N` natural number type and eliminators in place of explicit recursion, and again to use eliminators but maintaining the use of the original `int` type. Totals include both original and adapted implementations and equivalence proofs because reasoning about calling code typically refers to all components.

eliminators, as required by Œuf's input language, and proved the adapted version equivalent to the original. The size of each is shown in Figure 4.13.

The shim for `list_max` reads integers from standard input and builds an Œuf list containing the equivalent `nat`s, using the data representation described in Section 4.6.2 It then invokes the compiled `list_max` function, converts the resulting `nat` to a C integer, and prints it.

SHA256. Our second test case is a SHA-256 hashing utility, similar to `sha256sum`. The Gallina component is a complete implementation of the SHA-256 hash function, originally developed by [10] for their verification of the OpenSSL SHA-256 code. `SHA256` is a nontrivial function, comprising about 200 lines of code, not including the numerous list and integer library functions that it relies upon.

We produced two adapted versions of `SHA256` for compilation with Œuf. The first variant, meant to test Œuf's core compilation strategy, uses Coq's `N` natural number type in place of the CompCert `int` type used in the original `SHA256` implementation. CompCert implements the `int` type as a dependent pair of a mathematical integer $x \in \mathbb{Z}$ and a proof that $0 \leq x < 2^{32}$, but our prototype compiler cannot directly compile types that include proof terms. Our second variant of `SHA256` uses CompCert's `int` type, relying on a native type definition to handle the mapping of the `int` type to a low-level representation.

We constructed the `SHA256_N` variant in two steps. First, we converted all uses of CompCert's `int` type to equivalent operations using Coq's built-in `N` natural number type, which (unlike `nat`) represents each number as a linked list of bits. Removing uses of `int` was largely a straightforward replacement of `int` and operations with equivalent functions over `N`s. Second, we converted uses of fixpoints and pattern matching to explicit invocations of eliminators, as we did for `list_max`. In most cases this transformation was completely mechanical, but two functions required more significant adjustment due to more complex recursive calls. The sizes of these variants (including adapted library functions) and the proofs relating them is shown in Figure 4.13.

The construction of the `SHA256_int` variant was more straightforward. There was no

Program	Input	Default	Boehm	Slab	OCaml
`list_max`	100 items	0.03 s	0.04 s	0.01 s	0.00 s
	1000 items	(OOM)	34.63 s	11.31 s	0.02 s
`SHA256_N`	55 bytes	2.22 s	3.12 s	1.31 s	0.07 s
	500 bytes	(OOM)	24.44 s	10.75 s	0.58 s
	5000 bytes	(OOM)	246.94 s	107.06 s	5.85 s
`SHA256_int`	55 bytes	0.03 s	0.04 s	0.02 s	0.06 s
	500 bytes	0.24 s	0.41 s	0.21 s	0.54 s
	5000 bytes	2.34 s	2.18 s	2.05 s	5.53 s

Figure 4.14: Performance results for our test programs. The first three timing columns show the time taken when running Œuf-compiled code with the default shim and allocator, with the default shim and the Boehm GC, and with the slab-based shim and allocator. The final column shows the time taken by the same program run via unverified extraction to OCaml. Trials marked "OOM" failed after exhausting the 4GB of address space.

need to replace uses of the `int` type, but as in the construction of `SHA256_N`, we replaced all fixpoints and pattern matching with eliminators and proved equivalence to the original implementation.

The C shims for `SHA256_N` and `SHA256_int` are similar: each reads bytes from standard input until EOF, then builds an Œuf list containing each byte (represented as an `N` or as as an `int`, depending on variant). The shim next passes the list to the compiled `SHA256` function, obtaining another list of bytes (again stored as `N`s or `int`s in the range 0–255) representing the hash. Finally, it prints each byte of the hash in hexadecimal, as is standard for hashing utilities.

4.9.3 Performance

Performance results for our example programs, including both variants of `SHA256`, are shown in Figure 4.14. The first two sections of the table (`list_max` and `SHA256_N`) show that Œuf is capable of compiling nontrivial programs using its default data representation (*i.e.*, no native types), but the resulting code is slow and uses extreme amounts of memory. This is caused in part by the naïve memory allocation strategy used in the current implementation. Due to the difficulty of correctly implementing and verifying automatic memory management, the Œuf prototype compiler generates code that never deallocates. Temporary objects are leaked once they become unused, and the size of the heap quickly exceeds the 4GB address space limit for 32-bit processes.

To get some idea of our test programs' performance on larger inputs, we ran them with two other memory allocation strategies. These options reduce the overall memory footprint, but at the cost of increasing the TCB. The fourth column of the table shows the results of running the test programs with the Boehm garbage collector [21], which automatically frees memory that is no longer reachable. The fifth column shows the results with a modified shim that uses slab allocation. Slab allocation is an allocation strategy where new memory is allocated from various slabs, and it is possible to free an entire slab at a time. This is useful when calling Œuf-compiled code, as the result of each call can be copied out, and the slab used to make the call can be freed. The slab-allocated variants are modified such that iteration over the input sequence is handled in C instead of in Gallina. The body of the loop is the same as in the original implementation: a call to a Gallina function that processes the next chunk of data. Aside from this change, the shim also uses a simple slab allocator (under 100 lines of code) that wraps the system `malloc`. After each iteration, the shim reads out the result, then invokes an allocator routine to discard all other data that was allocated during the iteration. Though verifying either a garbage collector or a slab allocator would require significant additional work, doing so would improve Œuf-compiled code performance along the lines of Figure 4.14.

The third section of Figure 4.14 (`SHA256_int`) shows the benefits of compiling with a more efficient data representation. The slower `SHA256_N` variant use the `N` numeric type from the Coq standard library, which is binary representation requiring $O(\log x)$ constructors. As a result, all operations on numbers take time linear or logarithmic in the values of their inputs. The `SHA256_int` variant, on the other hand, uses CompCert's 32-bit `int` type, along with a native type definition that compiles `int`s down to 32-bit machine words. It also relies on several native operation definitions that map arithmetic (mod 2^{32}) on `int`s to Cminor arithmetic primitives, which CompCert typically compiles down to single assembly instructions. The result is a significant speedup: 50x-100x improvement when evaluating SHA-256 on a variety of inputs.

However, while extending Œuf with appropriate native definitions can significantly speed up compiled programs, producing new native type and operation definitions can be time-consuming. The native operation definitions for the `int` type that were responsible for the speedup on `SHA256_int` consist of over 1800 lines of definitions and proofs. Note, however, that native type definitions are reusable across input programs. Future Coq developments that use CompCert's `int` type can reuse the same native definitions that were used in `SHA256_int` to obtain similar speedups on their own integer code. We expect that most developers of denotational compilers will first implement their compiler using only Œuf's built-in data representation and pre-existing native definitions, then define additional native types and operations only where necessary to achieve the desired level of performance.

Finally, we compared Œuf-compiled code with code extracted using Coq's built-in unverified extraction mechanism and then compiled with the standard OCaml compiler. For this comparison, we extracted the Œuf-adapted version of each test program's Gallina component, so the exact same code and data structures were used for both Œuf- and OCaml-compiled variants. Even using the same sub-optimal numeric representation, the OCaml-compiled version of `SHA256` is about 20 times faster than the version produced by the Œuf prototype compiler. We believe this difference is most likely caused by the prototype compiler's unoptimized handling of closures. It currently allocates a closure object for every call, including

every partial application within a call to a curried function, and makes no attempt to remove unused variables from closure environments. Furthermore, the generated code always calls using the closure's code pointer, even when the target is statically known, which prevents any inlining that might normally be performed by CompCert. Improving the prototype compiler's code generation to address these issues is left to future work.

4.10 Related Work

Verified Compilation. Work on verified compilers stretches almost as far as the invention of compilers themselves [89] and continued throughout the second half of the twentieth century [92]. (See [32] for further references.) But the breakthrough work of [83] fundamentally altered the landscape by showing that it was possible to carry out the full development in a proof assistant, showing that the assembly output is guaranteed to be equivalent to the input C program. Over the last decade, CompCert has been extended in a variety of directions, *e.g.*, to make guarantees about its parser [75]. Even more importantly, a wide array of projects use CompCert as a subcomponent [126, 9, 30, 53, 54]. Œuf similarly builds on top of CompCert.

Œuf is most similar in purpose and design to CertiCoq, a verified Gallina compiler developed by [7], CertiCoq uses the template-coq reflection mechanism to reflect Gallina functions into input ASTs, compiles the program down to CompCert's Clight language using a series of verified transformation passes, and runs the CompCert backend to produce executable code. Œuf can compile only a limited subset of Gallina programs[1], while CertiCoq can compile any Gallina program, but Œuf provides stronger guarantees about the programs in that limited subset. The key difference is that Œuf's restriction of the input language allows for translation validation of the reflection procedure, via a verified denotation. The resulting proof of correspondence between the original and reflected terms, together with the compiler's own correctness theorem, enables users of Œuf to reason soundly about compiled code in terms of its original high-level Gallina implementation. Œuf also exposes its correctness

[1]The CertiCoq implementation is not yet publicly available, so we have not been able to make direct comparisons.

theorem to the shim, which allows reasoning about code that calls Œuf-compiled functions.

Other work has also investigated verifying compilers for functional programming languages more generally, including features such as general recursion, side effects, and nondeterminism [14, 25, 26, 104, 41]. Perhaps most closely related is the CakeML verified ML compiler [80, 118]. CakeML supports a rich input language including mutually recursive functions, pattern matching, ML modules, and mutable arrays. CakeML's design was a major inspiration for the design of Œuf's internal passes: both compilers use many small transformation passes between closely related languages. One technical difference is that CakeML is able to bootstrap by directly evaluating the compiler inside HOL4. This avoids adding that "phase" of trust to the TCB, as discussed in Section 4.7. When given shallowly embedded HOL functions, the CakeML compiler uses a form of translation validation to guarantee their input arrives at the frontend of the compiler unchanged [94]. While CakeML syntactically derives a certificate that the input program evaluates to the correct answer in the given big step semantics, Œuf uses denotational semantics to denote the program back into Gallina. Either approach results in similar levels of trust, however the complexity of denotation has been a limitation preventing Œuf from supporting more features, and thus the approach from CakeML may improve the Œuf compiler.

The Œuf correctness theorem is useful for reasoning about programs in the target language that interact with Œuf-compiled code. One special case of such reasoning is *separate compilation*, which previous work has considered both in the context of CompCert [76, 117] and elsewhere [96, 125]. Œuf also does not assume that the entire program is in Gallina, instead linking with a shim during compilation.

Another approach to reasoning about the interaction between the shim and the source program is to use a *multi-language semantics* [103, 5]. Such a semantics gives a unified account of the behavior of the shim, the source program, and the compiled program, which allows clean reasoning.

Cogent [6] is a language for systems code whose compiler produces not only C code, but also a high-level specification in Isabelle/HOL and proof of refinement. Users can reason

about the code in terms of its high-level specification, similar to how Œuf users can reason in terms of the Gallina code.

Œuf's use of the computational denotation to translation validate reflection and lambda lifting is broadly similar to [51], which uses a denotational semantics to translation validate LLVM optimization passes. Œuf's reflection procedure is similar in spirit to Template Coq [88], which can reflect *all* of Gallina into syntax represented by an inductive datatype in Coq. On the other hand, Template Coq does not support *denoting* all programs in this syntax back into Coq.

Verified Software Toolchain. Languages such as Gallina are designed from the ground up for integration into a proof assistant. In contrast, languages not designed with such goals in mind require additional tooling to enable verification. One such example is VST [8], a framework for reasoning about C programs. It provides the ability for users to prove properties of their programs, then appeal to a mechanically verified theorem that those properties are preserved through compilation. VST provides a deeply embedded separation logic over a C-like language, while Œuf lets users prove properties of their programs using Coq's built-in support for reasoning about Gallina. While C programs tend to have orders of magnitude better performance than their Œuf-compiled counterparts, we believe that the proof effort saved puts Œuf at a useful point in the design space.

Other Verified Systems. The primary way to use Coq to build systems today is to use extraction [85]. While convenient, extraction provides no formal guarantee that the resulting program is semantically equivalent to the original. Another way to build systems in Coq is to use Coq.io, which provides monadic I/O primitives [28]. The user writes a Gallina program using these primitives for side-effectful operations, then reasons about it using theorems that describe the behavior of those monadic primitives. Executing the code still requires extraction and compilation with the unverified OCaml compiler. In principle, we could provide similar I/O facilities by extending Œuf with monadic primitives and implementing an interpreter in the C shim.

The seL4 operating system was one of the first major successes in formal verification of

large systems [79]. They spent 20 person years on verification, which was largely refinement proofs. By building verified compilers we can mitigate some of this burden for future projects of this scale.

GCminor [90] is a language similar to the CompCert Cminor IR, except containing garbage collection primitives. From this language, there is a verified translation to Cminor, along with a Cminor specification and implementation of a garbage collector. In the future, Œuf could benefit from a verified garbage collector.

4.11 Conclusion

We presented the denotational compiler architecture, which enables development of verified compilers with low verification effort. We also presented Œuf, a verified compiler from Gallina to assembly language that is well-suited for use as a component of a denotational compiler. Along with standard compiler verification techniques, Œuf uses computational denotation of its input language to perform translation validation of its reflection procedure and lambda-lifting pass, eliminating both from the TCB. Œuf also provides an extensible mechanism for translating Gallina types and functions to arbitrary low-level data representations and code, allowing optimization of denotational compiler output. Finally, we developed a prototype denotational compiler for a simple input language, and we also compiled and ran Appel's Gallina specification of SHA256 with Œuf to ensure that Œuf can handle more complex programs.

Chapter 5
CONCLUSION

Using the techniques described in this dissertation, we achieved a strong end-to-end correctness guarantee for the Clinical Neutron Therapy System. Specifically,

- We proved that prescription safety holds in a model of CNTS, and established correspondence between the model and the actual system implementation. This machine-checkable saftey case rests on only a small number of assumptions and manual analysis steps—the rest can be checked automatically using a variety of analysis tools.

- We constructed formally verified analysis tools for establishing key properties of the software components of CNTS. We based these tools on mechanized semantics for the EPICS dataflow language and for Python, which we developed using a new technique that produces focused and flexible semantics that faithfully reflect the real behavior of a reference implementation.

- We developed and verified a proof-of-concept compiler for the `calc` expression sublanguage of the EPICS dataflow system, using our new denotational compiler architecture to minimize verification effort, and tested it by compiling `calc` expressions from the CNTS therapy control software. Furthermore, we developed the Œuf verified Gallina compiler, which we expect to be useful for developing a variety of verified, high-performance denotational compilers.

Future Work It remains to be seen how well the challenges we identified and the techniques we developed generalize to critical systems other than CNTS. For instance, while CNTS has complex internal architecture and relies on domain-specific languages, the property we set

out to prove was quite simple, in that the desired output (the beam control interlock state) is an easily-specified function of the inputs (the sensor readings). Applying ours or other formal verification techniques to real-world systems with more complex requirements, such as a robotic arm that must sense and avoid collisions with humans, may reveal new obstacles to verification that require new approaches.

For semantics development, the marquer methodology proposes developing the entire semantics around a specific program and property of interest, which reduces development effort at the cost of making the semantics unsuitable for other verification tasks. One avenue for future research is to examine the tradeoffs involved in development of less-focused semantics. In particular, it seems plausible that developing semantics for a variety of languages, programs, and properties will reveal common elements between translation or intermediate language definitions that can be factored out into library code and exploited to further reduce the development effort for future semantics.

Finally, this dissertation introduces the idea of denotational compilers but does not investigate their effectiveness in much detail. It would be interesting to develop denotational compilers for a variety of programming languages to better characterize the types of languages and language features for which denotational compilers are most effective. In particular, it not obvious how common imperative control flow features such as loops and general recursion can be denoted into pure functions in Gallina or a similar language, and furthermore, it is unclear how the denotation techniques required will affect ease of verification, which is the primary advantage of the denotational compiler architecture.

BIBLIOGRAPHY

[1] Alloy: a language and tool for relational models. `http://alloy.mit.edu/alloy/`, 2014.

[2] *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Edinburgh, UK, June 2014.

[3] Java PathFinder. `http://babelfish.arc.nasa.gov/trac/jpf`, 2015.

[4] Safety case for CNTS prescription safety. `http://neutrons.uwplse.org`, August 2015.

[5] Amal Ahmed. Verified compilers for a multi-language world. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, pages 15–31, 2015.

[6] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, et al. Cogent: Verifying high-assurance file system implementations. In *ACM SIGPLAN Notices*, volume 51, pages 175–188. ACM, 2016.

[7] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. Certicoq: A verified compiler for coq. In *CoqPL Workshop*, CoqPL '17, 2017.

[8] Andrew W. Appel. Verified software toolchain. In *European Symposium on Programming*, pages 1–17, Saarbruecken, Germany, 2011. Springer.

[9] Andrew W. Appel. Verified software toolchain. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ESOP'11/ETAPS'11. Springer-Verlag, 2011.

[10] Andrew W. Appel. Verification of a cryptographic primitive: Sha-256. *ACM Trans. Program. Lang. Syst.*, 37(2), April 2015.

[11] Argonne National Laboratory. EPICS - experimental physics and industrial control system. `https://epics.anl.gov`, 2018. Accessed: 2018-01-17.

[12] Domagoj Babic and Alan J. Hu. Calysto: scalable and precise extended static checking. In *ICSE*, 2008.

[13] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS*, 2004.

[14] Nick Benton and Chung-Kil Hur. Biorthogonality, step-indexing and compiler correctness. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 97–108, 2009.

[15] Lennart Beringer, Adam Petcher, Katherine Q. Ye, and Andrew W. Appel. Verified correctness and security of openssl hmac. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 207–221, Washington, D.C., 2015. USENIX Association.

[16] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[17] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '05, pages 265–276, New York, NY, USA, 2005. ACM.

[18] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 55–66, New York, NY, USA, 2006. ACM.

[19] Sandrine Blazy. Which C semantics to embed in the front-end of a formally verified compiler?, March 2008.

[20] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the clight subset of the c language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[21] H Boehm and M Weiser. Garbage collection in an uncooperative environment. In *Software Practice and Experience*, 1988.

[22] Denis Bogdănaş and Grigore Roşu. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages*, POPL '15, pages 445–456, Mumbai, India, January 2015. ACM.

[23] Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in coq. In *Proceedings of the 2011 IEEE 20th Symposium on Computer Arithmetic*, ARITH '11, pages 243–252, Washington, DC, USA, 2011. IEEE Computer Society.

[24] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, San Diego, CA, December 2008.

[25] Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 54–65, 2007.

[26] Adam Chlipala. A verified compiler for an impure functional language. *SIGPLAN Not.*, 45(1):93–106, January 2010.

[27] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, December 2013.

[28] Guillaume Claret. Coq.io, 2016.

[29] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS*, 2004.

[30] David Costanzo, Zhong Shao, and Ronghui Gu. End-to-end verification of information-flow security for c and assembly programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 648–664, 2016.

[31] Simon Cruanes, Grégoire Hamon, Sam Owre, and Natarajan Shankar. Tool integration with the evidential tool bus. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *VMCAI*, volume 7737 of *Lecture Notes in Computer Science*, pages 275–294. Springer, 2013.

[32] Maulik A. Dave. Compiler verification: A bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2–2, 2003.

[33] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, May 1979.

[34] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[35] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Cham, Switzerland, 2015. Springer International Publishing.

[36] E. Denney and G. Pai. Evidence arguments for using formal methods in software certification. In *Software Reliability Engineering Workshops (ISSREW), 2013 IEEE International Symposium on*, pages 375–380, Nov 2013.

[37] Ewen Denney, Ganesh Pai, and Josef Pohl. AdvoCATE: An assurance case automation toolset. In *Proceedings of the 2012 International Conference on Computer Safety, Reliability, and Security*, SAFECOMP'12, pages 8–21, Berlin, Heidelberg, 2012. Springer-Verlag.

[38] Dependability Research Group. Safety cases repository. `http://dependability.cs.virginia.edu/info/Safety_Cases:Repository`, February 2006.

[39] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2016. Version 8.5.

[40] J. Dolby, M. Vaziri, and F. Tip. Finding bugs efficiently with a SAT solver. In *FSE*, 2007.

[41] Maxime Dénès and Xavier Leroy. Coqonut: A verified JIT compiler for Coq. `http://www.maximedenes.fr/download/coqonut.pdf`, January 2015.

[42] Chucky Ellison. *A Formal Semantics of C with Applications*. PhD thesis, University of Illinois at Urbana-Champaign, July 2012.

[43] Chucky Ellison and Grigore Roşu. An executable formal semantics of C with applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 533–544, New York, NY, USA, 2012. ACM.

[44] EPICS. CalcOut Release Notes. `http://www.aps.anl.gov/bcda/synApps/calc/calcReleaseNotes.html`.

[45] EPICSwiki Contributors. EPICS 3-14 record reference manual. `https://wiki-ext.aps.anl.gov/epics/index.php/RRM_3-14`, 2018. Accessed: 2018-01-26.

[46] Michael D. Ernst, Dan Grossman, Jon Jacky, Calvin Loncaric, Stuart Pernsteiner, Zachary Tatlock, Emina Torlak, and Xi Wang. Toward a dependability case language and workflow for a radiation therapy system. In *Summit on Advances in Programming Languages (SNAPL)*, 2015.

[47] Free Software Foundation, Inc. Build status for GCC 6. `https://gcc.gnu.org/gcc-6/buildstat.html`, 2016. Accessed: 2016-09-10.

[48] Andrew Gacek, John Backes, Darren D. Cofer, Konrad Slind, and Mike Whalen. Resolute: An assurance case language for architecture models. *CoRR*, abs/1409.4629, 2014.

[49] J. P. Galeotti. *Software Verification Using Alloy*. PhD thesis, University of Buenos Aires, 2010.

[50] Patrice Godefroid and Ankur Taly. Automated synthesis of symbolic instruction encodings from I/O samples. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 441–452, New York, NY, USA, June 2012. ACM.

[51] Paul Govereau. *Denotational Translation Validation*. PhD thesis, Harvard University, 2012.

[52] William S. Greenwell, John C. Knight, C. Michael Holloway, and Jacob J. Pease. A taxonomy of fallacies in system safety arguments. In *International System Safety Conference*, 2006.

[53] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 595–608, 2015.

[54] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 653–669, 2016.

[55] Dwight Guth. A formal semantics of Python 3.3. Master's thesis, University of Illinois at Urbana-Champaign, 2013.

[56] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 336–345, Portland, OR, USA, June 2015. ACM.

[57] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 1–17, New York, NY, USA, 2015. ACM.

[58] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Security Symposium*, USENIX Security '12, pages 38–38, Berkeley, CA, USA, 2012. USENIX Association.

[59] Gerard J. Holzmann. The power of 10: Rules for developing safety-critical code. *Computer*, 39(6):95–97, June 2006.

[60] Gerard J. Holzmann. Mars code. *Commun. ACM*, 57(2):64–73, February 2014.

[61] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[62] International Atomic Energy Agency. *Investigation of an Accidental Exposure of Radiotherapy Patients in Panama.* International Atomic Energy Agency, Vienna, Austria, 2001.

[63] Daniel Jackson. A direct path to dependable software. *Commun. ACM*, 52(4):78–88, April 2009.

[64] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, February 2012.

[65] Daniel Jackson and Eunsuk Kang. Property-part diagrams: A dependence notation for software systems. Technical report, Massachusetts Institute of Technology, 2009. `http://hdl.handle.net/1721.1/61343`.

[66] Daniel Jackson and Martyn Thomas. *Software for Dependable Systems: Sufficient Evidence?* National Academy Press, Washington, DC, USA, 2007.

[67] Jonathan Jacky. The Clinical Neutron Therapy System. `http://staff.washington.edu/jon/cnts/index.html`.

[68] Jonathan Jacky. Formal safety analysis of the control program for a radiation therapy machine. In Wolfgang Schlegel and Thomas Bortfeld, editors, *The Use of Computers in Radiation Therapy*, pages 68–70. Springer Berlin Heidelberg, 2000.

[69] Jonathan Jacky. EPICS-based control system for a radiation therapy machine. In *International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS)*, 2013.

[70] Jonathan Jacky and Ruedi Risler. Clinical neutron therapy system reference manual. Technical Report 99-10-01, University of Washington, Department of Radiation Oncology, 2002.

[71] Jonathan Jacky and Ruedi Risler. Clinical neutron therapy system therapist's guide. Technical Report 99-07-01, University of Washington, Department of Radiation Oncology, 2002.

[72] Jonathan Jacky, Ruedi Risler, Ira Kalet, and Peter Wootton. Clinical neutron therapy system control system specification part I. Technical Report 90-12-01, University of Washington, Department of Radiation Oncology, 1990.

[73] Jonathan Jacky, Ruedi Risler, David Reid, Robert Emery, Jonathan Unger, and Michael Patrick. A control system for a radiation therapy machine. Technical Report 2001-05-01, University of Washington, Department of Radiation Oncology, 2001.

[74] Karl Heinz John and Michael Tiegelkamp. *IEC 61131-3: Programming Industrial Automation Systems Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids*. Springer Publishing Company, Incorporated, 2nd edition, 2010.

[75] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. Validating $LR(1)$ parsers. In *Proceedings of the 21st European Symposium on Programming (ESOP 2012)*, volume 7211 of *Lecture Notes in Computer Science*, pages 397–416. Springer, March 2012.

[76] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. Lightweight verification of separate compilation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 178–190, 2016.

[77] Daniel Kästner, Ulrich Wünsche, Jörg Barrho, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler. In *ERTS 2018: Embedded Real Time Software and Systems*, Toulouse, France, January 2018. SEE.

[78] Tim Kelly and Rob Weaver. The goal structuring notation – a safety argument notation. In *Proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases*, July 2004.

[79] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09. ACM, 2009.

[80] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. Cakeml: A verified implementation of ml. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14. ACM, 2014.

[81] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* [2], pages 216–226.

[82] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness, 2010.

[83] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 42–54, Charleston, SC, USA, 2006. ACM.

[84] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[85] Pierre Letouzey. Extraction in Coq: An overview. In *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*, volume 5028 of *Lecture Notes in Computer Science*, pages 359–369, 2008.

[86] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, July 1993.

[87] Michael R. Lyu, editor. *Handbook of Software Reliability Engineering*. McGraw-Hill, Inc., 1996.

[88] Gregory Malecha. Template Coq plugin. `https://github.com/gmalecha/template-coq`, 2015.

[89] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. *Mathematical aspects of computer science*, 1, 1967.

[90] Andrew McCreight, Tim Chevalier, and Andrew Tolmach. A certified framework for compiling and executing garbage-collected languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 273–284, 2010.

[91] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[92] J. Strother Moore. A mechanically verified language implementation. *J. Autom. Reasoning*, 5(4):461–492, 1989.

[93] Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. Verified peephole optimizations for compcert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, 2016.

[94] Magnus O. Myreen and Scott Owens. Proof-producing synthesis of ml from higher-order logic. ICFP '12, 2012.

[95] Joseph P. Near, Aleksandar Milicevic, Eunsuk Kang, and Daniel Jackson. A lightweight code analysis and its role in evaluation of a dependability case. In *Proceedings of the 33rd International Conference of Computer Safety, Reliability and Security*, pages 31–40, Waikiki, Honolulu, HI, May 2011.

[96] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. Pilsner: a compositionally verified compiler for a higher-order imperative language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 166–178, 2015.

[97] Matthew Newville. PyEpics overview. `http://cars9.uchicago.edu/software/python/pyepics3/overview.html`, 2014. Accessed: 2017-11-09.

[98] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[99] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[100] Scott Owens. A sound semantics for ocamllight. In *Proceedings of the Theory and Practice of Software, 17th European Conference on Programming Languages and Systems*, ESOP'08/ETAPS'08, pages 1–15, Berlin, Heidelberg, 2008. Springer-Verlag.

[101] Sam Owre, Natarajan Shankar, and John Rushby. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction (CADE)*, pages 748–752, Sam Owre, Natarajan Shankar, and John Rushby, June 1992.

[102] Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 346–356, Portland, OR, USA, June 2015. ACM.

[103] James T. Perconti and Amal Ahmed. Verifying an open compiler using multi-language semantics. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, pages 128–148, 2014.

[104] Clément Pit-Claudel. Compilation using correct-by-construction program synthesis. Master's thesis, Massachusetts Institute of Technology, August 2016.

[105] Plum Hall, Inc. The Plum Hall validation suite for C. `http://www.plumhall.com/stec.html`, 2016. Accessed: 2016-09-10.

[106] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98. Springer-Verlag, 1998.

[107] Python Software Foundation. dis - disassembler for Python code. `https://docs.python.org/2.7/library/dis.html`, 2017. Accessed: 2017-10-10.

[108] Python Software Foundation. The Python language reference (section 3: Data model). `https://docs.python.org/2.7/reference/datamodel.html`, November 2017. Accessed: 2017-11-09.

[109] Python Software Foundation. ast - abstract syntax trees. `https://docs.python.org/2.7/library/ast.html`, January 2018. Accessed: 2018-01-22.

[110] Tahina Ramananandro, Gabriel Dos Reis, and Xavier Leroy. A mechanized semantics for c++ object construction and destruction, with applications to resource management. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 521–532, New York, NY, USA, 2012. ACM.

[111] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[112] John Rushby. Formalism in safety cases. In *Safety-Critical Systems Symposium*, 2010.

[113] John Rushby. Mechanized support for assurance case argumentation. In Yukiko Nakano, Ken Satoh, and Daisuke Bekki, editors, *New Frontiers in Artificial Intelligence*, volume 8417 of *Lecture Notes in Computer Science*, pages 304–318. Springer International Publishing, 2014.

[114] Hanan Samet. Proving the correctness of heuristically optimized code. *Commun. ACM*, 1978.

[115] Traian Florin Şerbănuţă. *A Rewriting Approach to Concurrent Programming Language Design and Semantics*. PhD thesis, University of Illinois at Urbana-Champaign, December 2010.

[116] R. Statman. Logical relations and the typed lambda-calculus. *Information and Control*, 65(2):85 – 97, 1985.

[117] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional compcert. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 275–287, 2015.

[118] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. A new verified compiler backend for cakeml. ICFP '16, 2016.

[119] Ken Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8), August 1984.

[120] Emina Torlak. *A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2009. AAI0821754.

[121] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 135–152, New York, NY, USA, 2013. ACM.

[122] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* [2], pages 530–541.

[123] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'07, pages 632–647, Berlin, Heidelberg, 2007. Springer-Verlag.

[124] Daniele Varrazzo. psycopg. `http://initd.org/psycopg/`, 2017. Accessed: 2018-01-28.

[125] Peng Wang, Santiago Cuellar, and Adam Chlipala. Compiler verification meets cross-language linking via data abstraction. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 675–690, 2014.

[126] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. Jitk: A trustworthy in-kernel interpreter infrastructure. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14. USENIX Association, 2014.

[127] Charles Weinstock, John Goodenough, and John Hudak. Dependability cases. Technical Report CMU/SEI-2004-TN-016, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2004.

[128] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA, 1993.

[129] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 2007.

[130] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294, San Jose, California, USA, 2011. ACM.