Compiler Verification Meets Cross-Language Linking via Data Abstraction

Peng Wang
MIT CSAIL
wangpeng@csail.mit.edu

Santiago Cuellar
Princeton University
scuellar@princeton.edu

Adam Chlipala
MIT CSAIL
adamc@csail.mit.edu

Abstract

Many real programs are written in multiple different programming languages, and supporting this pattern creates challenges for formal compiler verification. We describe our Coq verification of a compiler for a high-level language, such that the compiler correctness theorem allows us to derive partial-correctness Hoare-logic theorems for programs built by linking the assembly code output by our compiler and assembly code produced by other means. Our compiler supports such tricky features as storable cross-language function pointers, without giving up the usual benefits of being able to verify different compiler phases (including, in our case, two classic optimizations) independently. The key technical innovation is a mixed operational and axiomatic semantics for the source language, with a built-in notion of *abstract data types*, such that compiled code interfaces with other languages only through axiomatically specified methods that mutate encapsulated private data, represented in whatever formats are most natural for those languages.

1. Introduction

Proof assistants like Coq and Isabelle have become standard tools for machine-checked proof of interesting theorems about programs. For instance, Hoare-style program logics [3] have been formalized and used to prove deep correctness theorems about nontrivial programs. Realistic compilers [17] have been verified to preserve the behavior of programs, spanning the gap from source code to assembly. One of the satisfying things about writing these proofs in a common proof assistant is that we can connect them to each other, giving us a way to, say, reason rigorously about C programs and end up with rigorous guarantees about assembly programs "for free." However, one hole in this story to date is: **how do we connect verified compilers and verified source files to produce whole verified assembly programs, when one program includes source files written in *different languages*?** In this paper, we present our method of plugging that hole, via a compiler verified in Coq, with a novel operational semantics for orchestrating cross-language function calls.

`ListSet.ll`:

```
1  typedef /* ...representation... */ ListSet;
2  ListSet ListSet_new() { /* ... */ }
3  void ListSet_delete(ListSet this) { /* ... */ }
4  void ListSet_add(ListSet this, int key) { /* ... */ }
5  int ListSet_size(ListSet this) { /* ... */ }
```

`Count.hl`:

```
6  // Spec: Return number of unique values in arr.
7  int countUnique(int[] arr) {
8    FiniteSet set = new ListSet();
9    for (int i = 0; i < arr.length; ++i)
10     set.add(arr[i]);
11   int ret = set.size();
12   delete set;
13   return ret;
14 }
```

`Main.ll`:

```
15 void main() {
16   int a[3] = {10, 20, 10};
17   printf("Result = %d\n", countUnique(a));
18 }
```

Figure 1. A multilanguage program

As a motivating example, consider the code examples in Figure 1. We show three source files, with filename extension `.ll` for a C-flavor low-level language and extension `.hl` for a higher-level language similar to C++ or Java. The first source file defines an efficient data structure for representing finite sets of integers. Perhaps it takes advantage of manual memory management or bit twiddling in ways unsupported by most high-level languages. However, this low-level code is packaged to be usable in higher-level code with a simpler memory model. For example, the second source file uses the finite-set data structure to compute how many *unique* values appear in an integer array. Naturally, the code for this simple algorithm is independent of representation details in the low-level code we rely on. The final snippet is low-level wrapper code that invokes the algorithm on a

specific input. Notice that, running the full program, our call stack ends up with low-level code on the bottom, high-level code in the middle, and then more low-level code on top.

Now say we want to reason about Figure 1 as a whole program compiled to assembly, proving that it meets some behavioral specification. We want to apply the main theorem of a verified compiler like CompCert [17], saying something like "any observable behavior of the compiled program was also a legal observable behavior of the source program." Unfortunately, the CompCert correctness theorem applies only to *whole programs implemented entirely in C*. It is not even possible to derive facts about what happens when we link C object files produced by different versions of CompCert. Such a limitation is no great problem in the world of high-assurance aerospace software, CompCert's first application domain, but it is incompatible with the way that most software is built today. Verified compilation needs to adapt to multilinguistic programs, with many modules handled by many different compilers. Benton and others [5, 16] have studied this problem before for statically typed high-level languages, but their solutions are not known to be compatible with multiphase compilers, where each phase should be provable separately in a natural way; and the connection with source-program verification has not been explored in detail. We bring all of these elements together for the first time.

We approached the problem in the context of the Bedrock framework [12, 13], a library for the Coq proof assistant that turns Coq into an IDE for verified programming. We implement, specify, prove, and compile (to assembly) programs entirely within Coq. At the bottom of the system is a *partial-correctness* Hoare logic for assembly language. When using Bedrock, our goal is to prove a correctness theorem for a complete program, by proving simpler theorems about its constituent library modules. Previous work with Bedrock implemented all modules in a C-like language [13] with a rather complex low-level semantics. In this paper, we show how to support coding some modules in a higher-level language, with a much simpler semantics enabling easier verification, and with a verified compiler bridging the gap to the rest of the Bedrock system.

Our main contribution in this paper is **the first integration of verified compilation within a framework for proving functional correctness of multilanguage programs**. We will step through the technical devices that enable us to verify, in Coq, the moral equivalent of the program from Figure 1, based on a broad compiler correctness theorem. Our verified compiler is for a high-level language like the one in the figure. We are able to establish the compilation correctness of each piece of high- or low-level code *modularly* and then compose them into a full-program correctness theorem. Along the way, we designed a *novel approach to phrasing compiler correctness on top of a target-language program logic*.

The program logic (Bedrock XCAP) is one of partial correctness, without modeling input-output interaction with the outside world, so our results are only oriented toward proving termination-insensitive properties that only constrain the internal state of an assembly-language abstract machine; but we are optimistic that the big-picture proof-structuring ideas would adapt to richer low-level program logics. The classic partial-correctness setting is also sufficient both to reveal some serious technical challenges and to illustrate some pleasant consequences of our approach, which integrates a modular target-language program logic with a compiler correctness proof from the start.

While carrying out this research, we were repeatedly surprised at how many technical challenges go away with a *unified* treatment of compiler verification and Hoare-logic proof of individual programs.

In order to take advantage of **data abstraction**, reasoning about program modules in terms of the narrow interfaces they export to manipulate private state, we decided to *integrate data abstraction into the operational semantics of our source language*. That is, we avoid needing to reason about the operational semantics of many different languages by getting all the languages to agree on one axiomatic semantics for method calls. A language's *operational* semantics only comes up in reasoning about its own module code, while *axiomatic* reasoning provides the glue between languages.

Our new formulation of compiler correctness is not too hard to explain at a high level. As a bit of a review, program logics are used conventionally for, e.g., proving Hoare triples like:

$$\{n \geqslant 0\}\ r = \texttt{fact}(n)\ \{r = n!\}$$

That is, one identifies a particular mathematical specification for a program and proves that the program meets the specification.

Applying the same idea in compiler verification, for each source program with its corresponding compiled target program, we apply a program logic **to the target program**, and the mathematical specification we pick to prove is **preservation of the source program's semantics**. Such a specification is relatively straightforward to write in higher-order program logics, even if it is out of scope in the more common first-order Hoare logics. In particular, consider a function compile from source-language commands s to target-language commands t. To do compositional verification of such a compiler, we informally prove the following proof rule for any s.

$$\frac{\mathsf{compile}(s) = t}{\{\text{safe to run } s\}\ t\ \{\text{state could result from running } s\}}$$

Using the separation-logic [24] interpretation of Hoare triples (where the precondition must imply that the program is crash-free), this rule means that **(1)** t **is safe to run whenever** s **is safe to run; (2) if** t **terminates, the final state is**

one that could have resulted from running s. We say that s and t are in a *backwards preservation* relation. We formalize these conditions in the remainder of the paper, showing how to relate the states of our source and target languages and how to model cross-module function calls.

This rule can be seen as an informal version of the compiler's **main correctness theorem**. Note that the rule supports the usual modularization of end-to-end program verification into *source-program verification* and *compilation verification*. The compiler's correctness theorem guarantees that a safe program's behavior will be preserved, regardless of how safety was established. Such program-specific proofs are part of source-program verification, an independent task. Therefore, the compiler's correctness theorem holds without any source-program verification or annotation effort by the programmer.

Our source language, named Cito, is a C-like language with expressions, standard control-flow constructs, and function calls via function pointers. Most interestingly, a notion of *abstract data type* is fundamental to the semantics. All data types beside integers are accessed through methods, which are specified in terms of their effects on mathematical models of private state. The Cito operational semantics is parametric in a set of abstract data types, and these data types may be implemented in any languages connected to Bedrock. Different types may even be implemented in different languages, without any impact on the process of verifying a specific Cito program.

This paper is structured as follows. Section 2 introduces Cito, our source language, with its operational semantics. Section 3 recaps the Bedrock framework. Section 4 gives an overview of our compiler, including its architecture and proof strategy. Sections 5, 6, and 7 describe in detail the compilation and proof techniques for statements, function calls, and program modules, respectively. Section 8 applies our main compiler theorem in two types of case studies: proof of program-specific specifications and verification of compiler optimization phases and their composition. The last two sections discuss related work and conclude.

Complete Coq source code of our compiler, proofs, and examples is available at: `http://people.csail.mit.edu/wangpeng/oopsla2014.tgz`

2. Cito: The Source Language

The source language of our verified compiler is called Cito. Much of the novelty of our verification approach is motivated by the desire to reason about the results of linking compiled Cito programs with programs produced by other compilers for other languages. One way to support such reasoning would be via some kind of explicit modeling of operational semantics for all the languages that appear in some full program. That way, we would be able to verify a compiler without committing to any particular program logic or other proof strategy for verifying individual programs.

Optional	$\lfloor\cdot\rfloor$	List Of	$(\cdot)^*$
Product Type	$\times$	Sum Type	$+$
Machine Word	$\mathbb{W}$	String	$\mathbb{S}$

Figure 2. Type notations

Constant	w	$\in$	$\mathbb{W}$
Label	l	$\in$	$\mathbb{S}_{\text{module}} \times \mathbb{S}_{\text{fun}}$
Variable	x	$\in$	$\mathbb{S}$
Binary Op	o	$::=$	$+ \mid - \mid \times \mid = \mid \neq \mid < \mid \leqslant$
Expression	e	$::=$	$x \mid w \mid e\ o\ e$
Statement	s	$::=$	$\mathsf{skip} \mid s; s \mid \mathsf{if}\ e\ \{s\}\ \mathsf{else}\ \{s\}$
			$\mid \mathsf{while}\ e\ \{s\} \mid x := \mathsf{call}\ e\ (e^*)$
			$\mid x := e \mid x := \mathsf{label}\ l$
Function	f	$\in$	$\mathbb{S}^*_{\text{arg}} \times \mathbb{S}_{\text{ret}} \times s$
Module	m	$\in$	$\mathbb{S}_{\text{name}} \times (\mathbb{S}_{\text{fname}} \times f)^*$

Figure 3. Cito syntax

Of course, it is far from clear how to phrase compiler correctness *generically* in the operational semantics of *all* other languages that our compiled code might interact with. To sidestep this problem, we **build axiomatic-semantics features into Cito's operational semantics**. The intuition is that *we use axiomatic features to reason about calls to other languages*, while we have our choice of operational or axiomatic reasoning within spans of code implemented entirely in Cito. In particular, our compiler correctness theorem applies usefully to Cito programs that have not been specified axiomatically, let alone verified.

Cito is a simple, untyped idealization of C, with crucial additional support for *abstract data types*, which are central to the style of axiomatic specification that we support. Cito includes the usual statement syntax constructs for assignment, sequencing, conditional tests, loops, and function calls. Function calls are made via function pointers, which can be stored and passed around freely across function boundaries.

Figure 3 gives the syntax of Cito. To simplify the semantics, we allow *expressions* e to include only safe and total operations like arithmetic, while function calls can be made only via dedicated forms of *statements* s. Function calls are made via function pointers, calculated from expressions. The $\mathsf{label}\ l$ statement is for retrieving the pointer corresponding to a given named function. A function label consists of a module name and a function name. A function consists of a list of formal parameter names, a return variable name, and a body. A module has a name and a list of named functions.

We give the (big-step) operational semantics of Cito in Figure 5, drawing on notations summarized in Figures 2 and 4. The last 3 rules, related to function calls, deserve special

$$
\begin{array}{rcl}
\text{Machine State } (\Sigma) & = & E \times H \\
\text{Variable Assignment } (\sigma) \quad E & = & \mathbb{S} \to \mathbb{W} \\
\text{Heap } (\mu) \quad H & = & \mathbb{W} \to \lfloor A \rfloor \\
\text{ADT Domain} \quad A & = & \text{[parameter of theory]} \\
\text{Context } (\Psi) & = & (\text{Label} \to \lfloor \mathbb{W} \rfloor) \times \\
 & & (\mathbb{W} \to \lfloor F \rfloor) \\
\text{Function Spec} \quad F & = & \mathsf{OP}(F_o) + \mathsf{AX}(F_a) \\
\text{Operational Func. Spec} \quad F_o & = & f \\
\text{Axiomatic Func. Spec} \quad F_a & = & P \times Q \\
\text{Precondition} \quad P & = & I^* \to \mathsf{Prop} \\
\text{Postcondition} \quad Q & = & (I \times O)^* \times I \to \mathsf{Prop} \\
\text{Input/Return Value} \quad I & = & \mathsf{ADT}(A) + \mathsf{SCA}(\mathbb{W}) \\
\text{Output Value} \quad O & = & \lfloor A \rfloor
\end{array}
$$

Figure 4. Notations in operational semantics

$$\frac{}{((\sigma,\mu), x := e) \Downarrow (\sigma[x \to [\![e]\!]_\sigma], \mu)}\ \textsc{Assign}$$

$$\frac{}{(\Sigma, \mathsf{skip}) \Downarrow \Sigma}\ \textsc{Skip} \qquad \frac{(\Sigma, s_1) \Downarrow \Sigma' \qquad (\Sigma', s_2) \Downarrow \Sigma''}{(\Sigma, s_1; s_2) \Downarrow \Sigma''}\ \textsc{Seq}$$

$$\frac{([\![e]\!]_{\Sigma.1} \neq 0 \wedge (\Sigma, s_T) \Downarrow \Sigma') \ \vee \ ([\![e]\!]_{\Sigma.1} = 0 \wedge (\Sigma, s_F) \Downarrow \Sigma')}{(\Sigma, \mathsf{if}\ e\ \{s_T\}\ \mathsf{else}\ \{s_F\}) \Downarrow \Sigma'}\ \textsc{If}$$

$$\frac{(\Sigma, \mathsf{if}\ e\ \{s; \mathsf{while}\ e\ \{s\}\}\ \mathsf{else}\ \{\mathsf{skip}\}) \Downarrow \Sigma'}{(\Sigma, \mathsf{while}\ e\ \{s\}) \Downarrow \Sigma'}\ \textsc{While}$$

$$\frac{\Psi.1(l) = w}{\Psi \vdash ((\sigma,\mu), x := \mathsf{label}\ l) \Downarrow (\sigma[x \to w], \mu)}\ \textsc{Label}$$

$$\frac{\begin{array}{c}\Psi.2([\![e_f]\!]_\sigma) = \mathsf{AX}(P, Q) \\ |w^*| = |I^*| = |O^*| \qquad P(I^*) \qquad Q(I^*, O^*, r_a) \\ \mathbf{match}(\mu, [\![e]\!]^*_\sigma, I^*) \qquad \mu' = \mathbf{upd}(\mu, [\![e]\!]^*_\sigma, I^*, O^*) \\ \mathbf{matchr}(\mu', r_w, r_a) \qquad \mu'' = \mathbf{updr}(\mu', r_w, r_a)\end{array}}{\Psi \vdash ((\sigma,\mu), x := \mathsf{call}\ e_f\ (e^*)) \Downarrow (\sigma[x \to r_w], \mu'')}\ \textsc{CallAx}$$

$$\frac{\begin{array}{c}\Psi.2([\![e_f]\!]_\sigma) = \mathsf{OP}(x^*_a, x_r, s) \\ \sigma'(x^*_a) = [\![e]\!]^*_\sigma \qquad \Psi \vdash ((\sigma', \mu), s) \Downarrow (\sigma'', \mu')\end{array}}{\Psi \vdash ((\sigma,\mu), x := \mathsf{call}\ e_f\ (e^*)) \Downarrow (\sigma[x \to \sigma''(x_r)], \mu')}\ \textsc{CallOp}$$

Figure 5. Operational semantics of Cito

explanation, which we work up to providing. The first 5 rules, however, are completely standard.

A Cito machine state, ranged over by Σ, is a pair of a variable assignment and a heap. A variable assignment σ is a total map from local variable names (strings) to fixed-width machine integer values. A heap μ is a partial map from addresses (integers) to objects of **abstract data types (ADTs)**. A domain A of ADT values is a parameter to our final theorem. Standard examples include sequences, finite sets, finite maps, and so on. In practice, A will often be a sum type, combining several standalone ADTs into one. Critically, values in the domain A are *mathematical models* of ADT values, not pointer-based *implementations*. For instance, we represent a finite set with a mathematical set, rather than with a balanced tree or hash table. Later we will return to the question of how ADTs are implemented.

The judgment $\Psi \vdash (\Sigma, s) \Downarrow \Sigma'$ (pronounced as "runs to") indicates that there exists an execution of statement s from machine state Σ to machine state Σ', under context Ψ. Notation $[\![e]\!]_\sigma$ refers to the obvious denotational semantics for expressions in terms of assignments to their free variables. Notation $m[a \to b]$ stands for updating map m at key a (possibly a new key) with value b.

The context Ψ of the operational semantics is only meaningful for the rules about label resolution and function calls, so "$\Psi \vdash$" is implied before every other use of the judgment. A context consists of a partial map from labels to function addresses (integers), plus a *function specification map*, which is a partial map from a function address to either an *axiomatic function specification* or an *operational function specification*.

Axiomatic Specification and ADTs. Figure 4 defines the domain of *function specifications* F. Such a specification is either an *operational* spec F_o or an *axiomatic* spec F_a. An operational spec constrains the behavior of a compiled function to match the behavior of a literal source function. In contrast, axiomatic specs use **data abstraction** with *preconditions* and *postconditions* to give a formal contract that a function must satisfy. With the axiomatic style, we avoid needing to commit to details of the programming language used to implement a function, focusing only on input-output behavior.

An axiomatic function specification consists of a *precondition* predicate P on the actual parameters (called input values) and a *postcondition* predicate Q over argument input-output pairs and the return value. That is, postconditions are primarily predicates over the ways that arguments *evolve* over the course of function calls, thanks to imperative side effects. A precondition-postcondition pair represents a Hoare-style function specification. The reason to have P in addition to Q is that P is used in the "safe" predicate (see Section 5) to define safe states from which all execution paths (possibly nonterminating) are crash-free.

An input value, passed as an argument to a precondition, can be either an ADT object (ADT constructor) or an integer (SCA constructor, for "scalar"). When given an ADT object as input, a function can choose to either (1) in-place modify it or (2) deallocate it. Leaving the state of an argument unchanged is a special case of "in-place modification" where the new mathematical model is the same as the old. This space of possibilities is modeled with the type $\lfloor A \rfloor$ of outputs, whose values are $\perp$, to indicate that a value is deallocated; or any element of A, the domain of ADT models, giving the *new* state of an argument upon function return.

Figure 6 illustrates the regime with an example. We specify finite sets of machine words, one of the ADTs used in the code example of Figure 1. Here a natural functional model of state (i.e., A parameter value) is arbitrary subsets of $\mathbb{W}$, the set of machine integers. We give the preconditions and postconditions of the four finite-set methods used in Figure 1. A precondition is a function over a list of input arguments, while a postcondition is a function over a list of output arguments (each element capturing "before" and "after" states) and a return value.

The specification of the `new` method says: the argument list should be empty, and the return value points to a freshly allocated finite set that encodes the mathematical empty set $\varnothing$. The specification for `delete` indicates that its argument finite set is deallocated, by constraining the output argument list to include some set s in the "before" state and the null value $\perp$ in the "after" state. Similarly, the spec of `add` indicates that the scalar argument w is added to the input set s, with an output list that shows the transition of the first argument from representing s to representing $s \cup \{w\}$. It is also easy to support methods that must provably leave the abstract state of their arguments alone, as in the case of `size`, whose spec constrains output lists to "transform" a finite set from s to s.

The notation requires some digesting, but basically it just formalizes standard intuitions about data abstraction in imperative languages. To integrate this style of specification with the Cito operational semantics, we need a few auxiliary definitions. First we have $\mathbf{upd}'$ and $\mathbf{upd}$, in Figure 7, which show how to use an output list to modify a heap. Here I represents the input value, and if it is an ADT object, w represents its address. O is the output value. If I is not an ADT object, O is ignored, and we do not change the heap.

Notice that an output list only lets us describe how an object passed as a function argument is modified or deallocated. To express allocation, we use return values, as described by auxiliary functions $\mathbf{matchr}$ and $\mathbf{updr}$ in Figure 7. A return value is also either an ADT object or an integer. An integer return value gives the scalar result directly. An ADT return value stands for a new object, and the semantics will nondeterministically pick an unused heap address to assign to the new object. This address, or the scalar return value, will be assigned to the variable on the left-hand side of $:=$. This slightly convoluted treatment provides substantial flexibility in the sorts of memory ownership transfer that can be modeled, including operations on multiple objects like list concatenation (taking two lists, "deallocating" the second by appending it to the first) and list split (taking a list, "allocating" a new list by stealing a portion from the input).

$$\mathbf{match}'(\mu, w, I) \equiv \begin{cases} \mu(w) = a & \text{if } I = \mathsf{ADT}(a) \\ w = w' & \text{if } I = \mathsf{SCA}(w') \end{cases}$$

$$\mathbf{match}(\mu, w^*, I^*) \equiv \bigwedge_{0 \leqslant i < |w^*|} \mathbf{match}'(\mu, w^*(i), I^*(i)) \wedge (\forall_{0 \leqslant i < j < |w^*|}, a_i, a_j.\ I^*(i) = \mathsf{ADT}(a_i) \wedge I^*(j) = \mathsf{ADT}(a_j) \Rightarrow w^*(i) \neq w^*(j))$$

$$\mathbf{upd}'(\mu, w, I, O) \equiv \begin{cases} \mu[w \to O] & \text{if } I = \mathsf{ADT}(a) \wedge O \neq \perp \\ \mu - w & \text{if } I = \mathsf{ADT}(a) \wedge O = \perp \\ \mu & \text{if } I = \mathsf{SCA}(w') \end{cases}$$

$$\mathbf{upd}(\mu, w^*, I^*, O^*) \equiv \mathbf{fold}(\mathbf{upd}', \mu, w^*, I^*, O^*)$$

$$\mathbf{matchr}(\mu, w, I) \equiv \begin{cases} w \notin \mathsf{dom}(\mu) & \text{if } I = \mathsf{ADT}(a) \\ w = w' & \text{if } I = \mathsf{SCA}(w') \end{cases}$$

$$\mathbf{updr}(\mu, w, I) \equiv \begin{cases} \mu[w \to a] & \text{if } I = \mathsf{ADT}(a) \\ \mu & \text{if } I = \mathsf{SCA}(w') \end{cases}$$

Figure 7. Auxiliary functions in operational semantics

Auxiliary functions $\mathbf{match}'$ and $\mathbf{match}$ take care of projecting ADT objects out of the heap given their addresses. A function at runtime only receives integers as actual parameters. Its specification describes how it will treat them. If the specification says this input value is an ADT object, the function will treat the runtime actual parameter as the address of that object. Otherwise it will just treat it as a scalar integer value. $\mathbf{match}'$ and $\mathbf{match}$ describe this behavior by stating the relation between the runtime values w^* and the semantic values I^*.

One important element of this specification approach may not be apparent from the formalism: we follow the *small footprint* style of separation logic [24], where function preconditions and postconditions need only describe the parts of the heap that the function manipulates. The auxiliary function definitions are chosen to support small-footprint reasoning soundly, and our compiler correctness proof uses separation logic to prove that soundness formally.

Operational Specification An operational specification is just the callee function itself, i.e. its formal parameter names, its result variable name, and its function body. The semantics says that the caller will give the whole heap to the callee, nondeterministically pick a variable environment for the callee, initialize the formal parameters with the actual parameters, run the function body, and take the heap and the value of the return variable as the result.

$$
\begin{array}{rcl}
A = \mathsf{FSET}(\mathbb{P}) + \cdots & & \mathbb{P} = \mathcal{P}(\mathbb{W}) \text{ (* sets of machine integers *)} \\
\{\lambda I.\ I = []\} & \texttt{new} & \{\lambda(O, R).\ R = \mathsf{ADT}(\mathsf{FSET}(\varnothing))\} \\
\{\lambda I.\ \exists s.\ I = [\mathsf{ADT}(\mathsf{FSET}(s))]\} & \texttt{delete} & \{\lambda(O, R).\ \exists s.\ O = [(\mathsf{ADT}(\mathsf{FSET}(s)), \bot)] \wedge R = \mathsf{SCA}(\cdot)\} \\
\{\lambda I.\ \exists s.\ I = [\mathsf{ADT}(\mathsf{FSET}(s))]\} & \texttt{size} & \{\lambda(O, R).\ \exists s.\ O = [(\mathsf{ADT}(\mathsf{FSET}(s)), \mathsf{FSET}(s))] \wedge R = \mathsf{SCA}(|s|)\} \\
\{\lambda I.\ \exists s, w.\ I = [\mathsf{ADT}(\mathsf{FSET}(s)), \mathsf{SCA}(w)]\} & \texttt{add} & \{\lambda(O, R).\ \exists s, w.\ O = [(\mathsf{ADT}(\mathsf{FSET}(s)), \mathsf{FSET}(s \cup \{w\})), \\
 & & (\mathsf{SCA}(w), \bot)] \wedge R = \mathsf{SCA}(\cdot)\}
\end{array}
$$

Figure 6. An example ADT specification (finite sets)

The merit of axiomatic specifications is that they hide implementation details of algorithms and data structures (with the help of ADTs) behind pure mathematical relations. They even hide which language an implementation uses. In principle all we need are axiomatic specifications. However, just like loop invariants, axiomatic specifications must be supplied by the programmer for specific programs. The benefit of having operational specifications is that each function then automatically gets a specification for free, and given a program, the semantics can define its behavior without the programmer giving any annotations. It also frees up the programmer to later use any verification tools (program logics, abstract interpretation, etc.) she likes to work on top of this semantics. Some higher-order properties of programs, especially those involving embedded code pointers, are not expressible by our axiomatic specifications, but all program properties are expressible by operational specifications (since what they say is just "it behaves as it behaves"). The programmer can later choose a more powerful program logic to deal with those higher-order properties. CompCert [17] goes the other way by having only operational semantics, but then it loses the nice interface axiomatic specifications provide to enable cross-module and cross-language linking.

3. The Bedrock Framework

Bedrock is a library for the Coq proof assistant, supporting implementation, compilation, specification, and verification of programs within the normal Coq environment. The library provides support for effective mostly automated proofs [12, 19] of correctness of low-level programs, using specifications inspired by higher-order separation logic [24]. The Bedrock structured programming system [13] supports efficient coding and verification of assembly programs, using higher-level *macros* that expand into assembly code, while supporting verification more at a C-like abstraction level.

In the rest of this section, we outline three crucial elements of Bedrock: the cross-platform assembly language at the base, the program logic that is applied to phrase correctness results for modules of assembly code, and the structured programming system that is layered on top of the assembly language for more convenient implementation of verified libraries. These contributions are all from past work, and they will form the foundation for our compiler verification approach.

$$
\begin{array}{rcll}
\text{Constants} & c & ::= & \text{[width-32 bitvectors]} \\
\text{Code labels} & \ell & ::= & \ldots \\
\text{Registers} & r & ::= & \mathsf{Sp} \mid \mathsf{Rp} \mid \mathsf{Rv} \\
\text{Addresses} & a & ::= & r \mid c \mid r + c \\
\text{Lvalues} & L & ::= & r \mid [a]_8 \mid [a]_{32} \\
\text{Rvalues} & R & ::= & L \mid c \mid \ell \\
\text{Binops} & o & ::= & + \mid - \mid \times \\
\text{Tests} & t & ::= & = \mid \neq \mid < \mid \leqslant \\
\text{Instructions} & i & ::= & L \leftarrow R \mid L \leftarrow R\ o\ R \\
\text{Jumps} & j & ::= & \mathsf{goto}\ R \mid \mathsf{if}\ (R\ t\ R)\ \mathsf{then}\ \ell\ \mathsf{else}\ \ell \\
\text{Blocks} & B & ::= & \ell : \{\lambda\gamma.\ \phi\}\ i^*; j \\
\text{Modules} & M & ::= & B^*
\end{array}
$$

Figure 8. Syntax of the Bedrock IL

3.1 The Bedrock IL Language

In the end, any Bedrock program is written in a cross-platform assembly language that is easy to compile to any of the common assembly languages, whose syntax is shown in Figure 8. We refer readers to past work [13] for details of this unremarkable language, the Bedrock IL. The language design follows standard assembly-language conventions, with, e.g., a small set of registers and a finite memory indexed by fixed-width machine words. One central concept is that of *code module*, a set of basic blocks, where each block has a label, a specification (precondition), and a sequence of straightline instructions followed by one jump instruction. Only the details of specifications are nonstandard, as they are based on the program logic sketched in the next subsection.

Ignoring specifications, Bedrock IL is given just the sort of operational semantics one expects. We will use metavariable γ to stand for machine states, which include register and memory values. The operational semantics, given a Bedrock module and a mapping from labels to words representing the real placement of code blocks in memory, expresses when one machine state is reachable from another, without violat-

ing safety by, e.g., jumping to a code address with no associated block.

Our final result in this paper is *foundational* in that we prove theorems about Bedrock IL programs, where the theorem statements only mention the operational semantics of the Bedrock IL, not any of the other technical devices we build up later. As a result, we need not worry about bugs in those technical devices corrupting our final conclusions.

3.2 The XCAP Program Logic

Bedrock's verification support is inspired by XCAP [22], a program logic for assembly programs that manipulate first-class code pointers. In particular, Bedrock directly adopts XCAP's *assertion logic* (in which code specifications are written) and uses a proof approach inspired by XCAP's *program logic* (in which programs are proved to satisfy their specifications).

It is challenging to design a program reasoning approach to support *modular verification* in the presence of first-class code pointers. One would like to prove a correctness theorem for a code module whose functions may be passed pointers to code in other modules, where the author of the first module should not need to specialize the proof to details of those other modules. Code pointers may even appear within arbitrarily complex data structures in memory.

Several semantic techniques have been developed to support this kind of higher-order specification. One is step-indexed logical relations [4]. XCAP takes a different tack: deep embedding of an assertion language PropX that is expressive enough to be useful but restrictive enough to support kinds of recursive definitions that would be ill-founded in normal logic. Most details of the assertion language will not be important in this paper. The usual connectives of higher-order logic are available with mostly standard meanings. Less usual features are an operator $\lceil \cdot \rceil$, letting us *lift* any normal Coq proposition into PropX; and a *Hoare double* operator $\{\lambda\gamma.\ \phi\}w$, which allows us to assert that the code at address w has a specification implied by $\lambda\gamma.\ \phi$ [1].

In the rest of the present paper, we will not need to appeal to details of the PropX semantics, relying on common-sense understanding of the connectives and considering the lifting brackets $\lceil \ldots \rceil$ implicit where necessary, since the messier details in our proofs are checked by Coq. For our purposes, the essential PropX feature is Hoare doubles $\{\lambda\gamma.\ \phi\}w$, which we will use in specifications for compiled programs, to impose requirements on pointers to code that may not have come out of our compiler. In other words, rather than a syntactic contract for compiled code via refinement of executions as in CompCert [17], we use a semantic contract via Hoare doubles, giving programmers and compiler writers more implementation freedom. The main ideas of our approach should generalize to other styles of higher-order program logic that define expressive specification languages by other means, like with step indexing.

XCAP also provides a *program logic*, specifying inference rules that can be used to conclude that a program component satisfies a specification. We write $\Psi \vdash M$ and $\Psi \vdash B$ to indicate correctness of a program module M or a single basic block B, respectively, under the assumptions about other basic blocks encoded in Ψ, a partial map from block addresses to assumed preconditions.

Bedrock simplifies the definition of this component into a single "inference rule" that refers directly to the operational semantics of basic blocks. We write $(\gamma, c) \to (\gamma', \ell)$ to indicate that running basic block code c transforms the machine state from γ to γ' and jumps to label ℓ. A basic block is written as $\{p\}\ c$, giving precondition p and instructions c.

$$\frac{\forall \gamma, \Psi' \supseteq \Psi.\ p_{\Psi'}(\gamma) \Rightarrow \exists \gamma', \ell.\ (\gamma, c) \to (\gamma', \ell) \wedge \Psi'(\ell)_{\Psi'}(\gamma')}{\Psi \vdash \{p\}\ c}$$

In other words, a block is correct if, whenever we enter it in a state satisfying its precondition, we step safely to the end of the block, jumping to a label whose own precondition (pulled from our assumptions) is then satisfied. We write p_Ψ for interpretation of a predicate according to a set of assumptions; see the XCAP paper [22] for formal details. Such an assumption-specific interpretation is necessary to assign a meaning to the Hoare double connective. Here we quantify over all assumption contexts that include at least the entries assumed by the code.

We say that a module is correct when each of its blocks is correct.

$$\frac{\forall B \in M.\ \Psi \vdash B}{\Psi \vdash M}$$

When Ψ exactly matches the specifications included in M, we have a whole-program correctness proof, and a straightforward induction establishes that, when control begins in a block whose precondition is satisfied, execution continues safely forever, and every block's precondition is satisfied every time control enters that block.

To support *modular* verification, we can derive a *linking* rule, where we overload the notation dom for calculating the code labels defined by a module or included in a finite map.

$$\frac{\Psi_1 \vdash M_1 \quad \Psi_2 \vdash M_2 \quad \mathsf{dom}(M_1) \cap \mathsf{dom}(M_2) = \varnothing \quad \forall \ell \in \mathsf{dom}(\Psi_1) \cap \mathsf{dom}(\Psi_2).\ \Psi_1(\ell) = \Psi_2(\ell)}{\Psi_1 \cup \Psi_2 \vdash M_1 \cup M_2}$$

Informally, when we have proved two modules correct under some assumptions, we have proved the concatenation of the modules correct, under the union of their assumptions. There are further requirements that both modules do not try to define the same label, and that there is no disagreement between the assumptions, but these side conditions are purely syntactic, so that all the real verification work happens in

[1] Actually, a more primitive connective is included in place of the Hoare double and used to derive it.

verifying individual modules according to Ψ values formalizing their assumptions about other modules.

At this point we have seen the key ingredients behind our new approach to compositional compiler verification. While past work [5, 16] has tackled the problem via *approximation* using step indices, we will achieve similar results using *syntax* via the XCAP approach. By requiring proofs of correctness for individual compilation runs to be phrased in the Bedrock program logic rather than with arbitrary Coq arguments, we gain the ability to use the standard *linking* theorem to reason about the combination of code output by our compiler with arbitrary other verified Bedrock modules.

3.3 The Bedrock Structured Programming System

In theory, XCAP provides all the tools needed to do modular verification. In practice, direct reasoning about unstructured assembly programs is excessively time consuming. Bedrock provides a macro system [13] that effectively creates an extensible C-like language for programming and verification within Coq. The key construct is *certified low-level macros*, which introduce new programming notations along with their associated *compilation rules* and *proof rules*, plus proofs that the rules taken together are sound with respect to the Bedrock IL operational semantics.

In this paper, we will not need to refer to the details of Bedrock macros. Instead, we will proceed as though our compiler were targeting a C-like programming language with invariant annotations in the usual places (e.g., before loops). All of the programming constructs we use are defined independently as macros, and, beyond the original set of macros [13], we implemented a few new ones that were convenient for our compiler. Overall, our compiler rules will look suspiciously like translating one C-like language into another, but the interesting part will be the *specifications* that we associate with compiled code, supporting later linking with code compiled from other languages, without any knowledge of our compilation strategy.

The central theorem of the Bedrock structured programming system is that any verified macro may be used to produce a verified Bedrock IL module, with function specifications based on the Hoare triples proved for structured programs. By *verified* here, we mean that each module has an appropriate syntactic proof using the inference rules from the last subsection. Thus, these compiled modules may be linked together in the usual way to produce larger verified programs.

One of the abstractions that can be built on top is the normal Hoare triple of separation logic [24], where we write $\{P\}c\{Q\}$ for command c and *stateful* predicates P and Q, declaring that c is safe to run in any memory containing a submemory satisfying P, such that c will modify only that submemory and leave it in a state satisfying Q. Now we can state the rule for our verified Cito compiler, which is itself a Bedrock macro, as:

$$\frac{\mathsf{compile}(s) = c}{\forall \Sigma.\{(\Sigma, s) \downarrow \wedge\ \mathsf{state}(\Sigma)\}\ c\ \{\exists \Sigma'.\ (\Sigma, s) \Downarrow \Sigma' \wedge \mathsf{state}(\Sigma')\}}$$

This is **the main correctness theorem we prove for our compiler** (though here we omit a few details to be filled in by Section 7).

Here Σ's are Cito states, and stateful predicate $\mathsf{state}(\Sigma)$ establishes when the Bedrock IL state suitably implements Σ. Relation $(\Sigma, s) \downarrow$ asserts that executing s in Σ will not crash, and $(\Sigma, s) \Downarrow \Sigma'$ asserts that one possible result of running s in Σ is Σ'. In other words, the macro's proof rule says that, if the current Bedrock IL state implements a Cito state that leads to safe execution of s, then a postcondition for running c is that the Bedrock IL state implements some Cito state that could result from running s. This is a *backwards preservation* property in the sense we adopt throughout this paper, which is a big-step simplification of the backwards simulation relations of CompCert.

In a hypothetical Bedrock-like system based on a total-correctness program logic, our basic technique should adapt naturally to proving preservation of program termination properties. We would just interpret the $\downarrow$ relation as enforcing safe termination, rather than just crash freedom, along all possible execution paths.

4. Compilation Overview

The core component of the compiler is the statement compiler, which is a total function from a Cito statement s to a Bedrock structured code chunk. Following Bedrock terminology, we call this function a macro and associate a proof rule with it. A specific form of Hoare-style precondition and postcondition will be useful as an intermediate invariant during compilation. We formalize an invariant schema $\mathsf{inv}_{\mathcal{V}}(s)$, where s is a Cito statement, and $\mathcal{V}$ is the list (position-sensitive) of declared local variables of the enclosing function (generally omitted later when not relevant).

$$\begin{aligned}\mathsf{inv}_{\mathcal{V}}(s) \equiv \lambda c.\ \forall \Psi, \Sigma.\ \{&\mathsf{funcsOk}(\Psi) \wedge \Psi \vdash (\Sigma, s) \downarrow \\ &\wedge\ \mathsf{state}_{\mathcal{V}}(\Sigma)\}\ c\ \{\exists \Sigma'.\ \Psi \vdash (\Sigma, s) \Downarrow \Sigma' \wedge \mathsf{state}_{\mathcal{V}}(\Sigma')\}\end{aligned}$$

Invariant $\mathsf{inv}(s)$ is defined as a specification for a Bedrock code chunk c. We read it as: the code chunk c is safe to call for any function specs Ψ and Cito state Σ that lead to safe execution of s, when the Bedrock IL state contains implementations of the functions from Ψ (formalized with funcsOk, explained in Section 6) and accurately represents Σ (formalized via state). If c then terminates, it does so in some Bedrock IL state corresponding to a Cito state Σ' that could really result from running s in Σ. We will use $\{P\}\_\{Q\}$ as a short-hand for $\lambda c.\{P\}c\{Q\}$. $\mathsf{inv}(s)$ will be the specification for the code chunk generated for statement s.

The predicate state is defined in separation logic. We write $\bigotimes$ for iteration of the separating conjunction operator

$$\frac{(s_1, \Sigma)\downarrow \quad \forall \Sigma'.(\Sigma, s_1) \Downarrow \Sigma' \Rightarrow (s_2, \Sigma')\downarrow}{(s_1; s_2, \Sigma)\downarrow}$$

Figure 9. A selected rule of the safety judgment

for carving memory into disjoint pieces described by different assertions. Where μ is a heap of abstract data type values, we have:

$$\mathsf{heap}(\mu) \equiv \bigotimes_{(p,a)\in\mu} \mathsf{repInv}(p, a)$$

The predicate repInv is the *representation invariant* for ADT objects, which describes, in the separation-logic style, how the low-level machine state should faithfully represent an ADT object. For example, if the object is a mathematical finite set, a possible repInv may require the low-level memory heap portion to be organized as a balanced search tree. Like the ADT domain A, repInv is also a parameter of our formal development, which can be customized for different representation invariants (e.g. changing from a splay tree to a red-black tree).

Now, with $\mathcal{V} = \{x_1, \ldots, x_n\}$ as a list of local variables, we define:

$$\mathsf{state}_{\mathcal{V}}(\sigma, \mu) \equiv \mathsf{heap}(\mu) \wedge \textstyle\bigwedge_i x_i = \sigma(\text{“}x_i\text{”})$$

Here we play a bit fast and loose with the separation-logic convention of allowing program variables to appear free in assertions. The details of variable manipulation are made fully formal in our Coq code, where local variables are just one part of the Bedrock macros for function definition. The intuitive content of the state definition is that the heap is laid out properly in memory and all local variables have the values assigned to them by the Cito state.

Figure 5 defined the $\Downarrow$ relation used in the postcondition part of inv. A similarly defined relation $\Psi \vdash (\Sigma, s)\downarrow$ indicates that statement s is *safe* to run in Cito state Σ, a selected representative rule of which is shown in Figure 9. While the base operational semantics $\Downarrow$ only captures a single terminating execution, $\downarrow$ forces consideration of all possible executions, even those that do not terminate. None of them are allowed to violate the safety rules implied by $\Downarrow$. The relation is defined *coinductively* (denoted by the doubled horizontal line) to allow a state to be considered safe even if it may lead to nonterminating executions. This formulation has much in common with coinductive big-step operational semantics [18].

In the course of verifying substatements of a function body, we need to choose loop invariants and other characterizations of intermediate states. To do so, it is helpful to interpret the inv definition in a slightly nonstandard way. When given as a loop invariant, its “precondition” part describes the *current* machine state, while the “postcondition” part describes the machine state *when the current function returns*. Such a schema is straightforward to encode in XCAP, using the higher-order features of the PropX assertion language.

```
compile(“a; b”, k) =
  compile(a, “b; k”); compile(b, k)

compile(“if e {s1} else {s2}”, k) =
  compileExpr(e, 0);
  if Rv ≠ 0 {compile(s1, k)} else {compile(s2, k)}

compile(“while e {s}”, k) =
  compileExpr(e, 0);
  [loopInv(e, s, k)]
  while Rv ≠ 0 {
    compile(s, “while e {s}; k”);
    compileExpr(e, 0) }
```

Figure 10. Compilation of control-flow structures

With this interpretation of inv, our proof obligation for the correctness of compiling statement s is: for any *continuation* statement k, if we know $\mathsf{inv}(k)$ is a valid specification for the compilation of k, can we establish that $\mathsf{inv}(s; k)$ is a valid specification for the compilation of $s; k$? Abusing the Hoare-triple notation, the proof obligation is:

$$\frac{\mathsf{compile}(s, k) = c}{\{\mathsf{inv}(s; k)\}\ c\ \{\mathsf{inv}(k)\}}$$

In other words, the job of a statement s is to take us *from* a state where we must run $s; k$ before returning, *to* a state where we only need to run k before returning (the reason to have a k in $\mathsf{compile}(s, k)$ is explained in Section 5). This continuation-based reasoning is important to enable us to come up with invariants translatable to the level of assembly, which is naturally considered to be in continuation-passing style.

5. Compilation of Basic Statements

To finish the verification, we must not just define the compiler and its Hoare rule, but we must prove that the latter is sound with respect to the former. We implement the compiler using more primitive macros from the Bedrock library, for constructs like if and while. Interpreting these macros directly allows us to derive a very specific Hoare rule for the compiler. To translate to the more abstract interface $\mathsf{inv}(s)$, we apply a Bedrock macro counterpart to the Hoare-logic rule of consequence. The implication between the two specifications (proved by induction on statement syntax) is responsible for the bulk of our verification effort.

As an example, we show in Figure 10 compilation of the classic control-flow forms for sequencing, conditional test, and looping. To distinguish between the syntax of Cito statements and Bedrock macros, we enclose the former in double quotes when otherwise ambiguous. Our compile function takes an explicit continuation argument, which has no algorithmic significance but is used to state invariants.

These rules have something of the feeling of many denotational semantics, where one translates between languages

by simply "removing the quotation marks." We take advantage of the preexisting Bedrock macros for the very control-flow constructs we are compiling. Where the Bedrock structured programming system provides an extensible programming language with support for proving specific functional correctness theorems, here we derive a compiler for a *fixed* programming language where we prove fidelity of compilation without requiring the programmer to state any program-specific correctness theorems.

The case for if in Figure 10 is perhaps the simplest, most directly following the strategy of "remove the quotation marks." Since the Bedrock IL does not support compound test expressions in its conditional jump instruction, the primitive Bedrock if macro does not either, so we use a standard expression compiler compileExpr to generate code that leaves the result of the conditional test e in IL register Rv, where the second argument i to compileExpr in general requires leaving untouched the first i numbered temporary variables. Afterward, we test if Rv stores zero, running either the "then" or "else" case as appropriate. The recursive calls to compile pass along the continuation k unchanged.

The shortest case is for sequencing, but it also introduces a wrinkle not found for if. We again implement sequencing using the normal Bedrock sequencing macro, but we must be a bit more careful in our recursive calls. The second statement b is associated with the original continuation k, but the first statement a is compiled according to an expanded continuation "$b; k$", expressing that not only k but also b must be run afterward.

It may seem strange that compilation needs to be parameterized on continuations at all. Why can we not just refer to continuations in the theorems we prove about the compiler? A conventional compiler would not generate different code based on knowledge of which other code is to follow, but our compiler is outputting not conventional code but *Bedrock modules*, which combine code and specifications. The macros used in the sequencing and if cases hide generation of proper specifications, but the remaining while case is instructive for its inclusion of an explicit invariant that must mention k.

The algorithmic part of the generated while code begins by storing the value of test expression e in Rv. The loop itself tests nonzeroness of Rv. The body runs the statement s and then recomputes the value of the test expression.

The additional specification-oriented parts of this case are more interesting. First, body s is compiled according to the expanded continuation "while e $\{s\}; k$", since the loop must finish running before we get to k. Second, we need to give a *loop invariant*, in terms of loopInv, a slight variant of inv. $\mathsf{loopInv}(e, s, k)$ is *almost* identical to inv("while e $\{s\}; k$"). The one exception is that we require that the current value of register Rv equals the current value of the loop test e, so that the test of the generated while captures the right property.

$$
\begin{array}{l}
\mathsf{compile}(\text{“}x := \mathsf{call}\ e_f\ (e^*)\text{”}, k) = \\
\quad \mathsf{compileExpr}(e(0), 0); \quad \#0 \leftarrow \mathsf{Rv}; \\
\quad \mathsf{compileExpr}(e(1), 1); \quad \#1 \leftarrow \mathsf{Rv}; \\
\quad \cdots \\
\quad \mathsf{compileExpr}(e_f, n); \\
\quad \mathsf{icall}\ \mathsf{Rv}(\#0, \#1, \cdots)\ [\mathsf{afterCall}(k)]; \\
\quad x \leftarrow \mathsf{Rv}
\end{array}
$$

Figure 11. Compilation of function calls

$$
\begin{array}{rl}
\mathsf{funcsOk}(\Psi) \equiv & (\forall w, s_a.\ \Psi.2(w) = \mathsf{AX}(s_a) \Rightarrow \{\mathsf{axSpec}(s_a)\}\ w) \\
& \wedge\ (\forall w, s_o.\ \Psi.2(w) = \mathsf{OP}(s_o) \Rightarrow \{\mathsf{opSpec}(\Psi, s_o)\}\ w) \\
\mathsf{axSpec}(P, Q) \equiv & \lambda c.\ \forall \mu, w^*, I^*.\ \{\mathsf{state}(w^*, \mu) \wedge P(I^*) \wedge \\
& \quad |w^*| = |I^*| \wedge \mathbf{match}(\mu, w^*, I^*)\}\ c \\
& \quad \{\exists O^*, \mu', \mu'', r_w, r_a.\ \mathsf{state}(\cdot, \mu'') \wedge \\
& \quad |w^*| = |O^*| \wedge Q(I^*, O^*, r_a) \wedge \\
& \quad \mu' = \mathbf{upd}(\mu, w^*, I^*) \wedge \mathbf{matchr}(\mu', r_w, r_a) \wedge \\
& \quad \mu'' = \mathbf{updr}(\mu', r_w, r_a) \wedge \mathsf{Rv} = r_w\} \\
\mathsf{opSpec} \equiv & \lambda c.\ \forall \Sigma.\ \{\mathsf{state}_{x_a^*}(\Sigma) \wedge \Psi \vdash (\Sigma, s) \downarrow\}\ c \\
(\Psi, x_a^*, x_r, s) & \quad \{\exists \Sigma'.\ \mathsf{state}(\cdot, \Sigma'.2) \wedge \Psi \vdash (\Sigma, s) \Downarrow \Sigma' \\
& \quad \wedge\ \mathsf{Rv} = \Sigma'.1(x_r)\}
\end{array}
$$

Figure 12. Definition of funcsOk

Loop invariants and similar inputs to Bedrock macros are used internally to generate the preconditions annotated on Bedrock IL basic blocks. This uniform scheme of assigning preconditions to blocks is what supports the simple syntactic proof method for modular verification, so the extra parameter k and the generation of loop invariants are at the heart of our approach to building compositional compiler correctness on top of a program logic.

6. Compilation of Function Calls

Compilation of calls to functions with either axiomatic or operational specifications share the same syntax and algorithmic code. The compilation code for function calls is shown in Figure 11. As in the prior examples, we appeal mostly to a preexisting Bedrock macro icall, for implementing an indirect function call via a code pointer. Before that, we use compileExpr to compile the function pointer expression e_f and the argument expressions e^*, stashing the argument values in numbered temporary variables $\#0$ to $\#(|e^*| - 1)$. Similarly to the while macro, icall requires an *after-call invariant*, which is used by Bedrock to pick a precondition for the basic block where control returns after the call. The invariant afterCall that we use here is again almost the same as inv, except that it requires that a bit of post-call code be run to adjust the stack pointer.

The definition of the funcsOk predicate, which we have postponed until now, is the crucial ingredient to support verification of the call implementation. Recall that inv is defined in terms of funcsOk, using it to enforce that a context Ψ accurately describes the actual functions available at the Bedrock level. Figure 12 gives the definition of funcsOk as a conjunction of two parts, corresponding to axiomatic specifications and operational specifications. The Hoare-double notation "$\{P\}\ w$" is used to express that the code at address w satisfies a specification P. The definition of funcsOk reads as: if the context indicates that the function at address w has an axiomatic specification s_a, the real code at w will satisfy a specification (expressed in PropX) $\mathsf{axSpec}(s_a)$; if the context indicates that the function at w has an operational specification s_o, the real code at w will satisfy a specification $\mathsf{opSpec}(\Psi, s_o)$.

Predicate axSpec is basically a verbatim translation of the premises required by the operational-semantics rule CallAx. Here we write $\mathsf{state}(w^*, \_)$ to express that at the bottom of the current call-frame are the actual parameters w^*. The end state is written as $\mathsf{state}(\cdot, \mu)$ to express that the local variables are ignored, since we only make a function call for its effect on the heap and the return value, not its own locals that are about to be deallocated.

Predicate opSpec is much like the Hoare triple within inv. There is some minor complication in the postcondition to project out and ignore the local-variable part of the final Cito state Σ'. The subscript in $\mathsf{state}_{x_a^*}(\sigma, \mu)$ is to emphasize that the low-level variable environment (call-frame) will only guarantee that variables x_a^* agree with σ, with no guarantee for variables beyond that domain.

It is worth pausing here to reflect on the central role of funcsOk in supporting transfer of control back and forth between code output by our compiler and code from other languages. By connecting to XCAP's mechanism for asserting specifications for first-class code pointers, we impose a discipline that is compatible with general XCAP module correctness derivations. Since inv quantifies universally over the Ψ that is passed to funcsOk, we verify our compiled Cito modules once and for all with respect to any environment of properly specified functions in which those modules may eventually be run.

7. Bundling

The end goal of all this formal development is certification of whole programs composed of some modules produced by our compiler and some modules constructed by other means. We package the statement compiler into a function compiler and finally into a module compiler, which, given a Cito module and an *import table* (a finite map from labels to axiomatic specifications), compiles each function and bundles the compiled functions into a Bedrock module. Each compiled function receives a Bedrock precondition:

$$\phi_f \quad \equiv \exists \Psi.\ \mathsf{funcsOk}(\Psi) \wedge \mathsf{opSpec}(\Psi, f) \tag{1}$$

The fact that our compiled function satisfies this specification can be considered as **the final correctness theorem of our compiler**.

One last puzzle remains when we actually want to call a function with this specification. The precondition begins with quantification over some context Ψ, with a requirement that $\mathsf{funcsOk}(\Psi)$. For a module, it is easy to choose the Ψ: just use the one that includes one entry for each of its functions and one entry for each of the imports, that is:

$$\begin{aligned}\Psi_m^i \equiv\ & \{l \mapsto \mathsf{OP}(f) \mid (l, f) \in m\} \\ \cup\ & \{l \mapsto \mathsf{AX}(s) \mid (l, s) \in i\}\end{aligned}$$

What is more challenging is how to prove $\mathsf{funcsOk}(\Psi_m^i)$. Consulting Figure 12 and using XCAP's rule of consequence, we arrive at subgoals of this form:

$$\mathsf{opSpec}(\Psi_m^i, f) \Rightarrow \phi_f \tag{2}$$

Unfortunately, consulting (1), there is a funcsOk in ϕ_f, so we wind up with a circularity, needing to prove funcsOk in the course of proving funcsOk!

Our solution takes advantage of the syntactic nature of XCAP reasoning in another way. We give each compiled function f a program-specific *stub* f' that just jumps to its namesake directly. Stub functions are assigned simpler XCAP specifications with a Ψ value inlined:

$$\psi_{f'} \quad \equiv \mathsf{opSpec}(\Psi_m^i, f)$$

With the specificational circularity broken, subgoal (2) becomes

$$\mathsf{opSpec}(\Psi_m^i, f) \Rightarrow \psi_{f'}$$

which is trivially true. We then have $\mathsf{funcsOk}(\Psi_m^i)$ as a lemma, which is the one piece of information missing to conclude ϕ_f from $\psi_{f'}$, to justify the jump from the stub to its namesake.

8. Composing with Compiler Correctness

Our compiler correctness theorem was chosen to satisfy one crucial dictum: **make it possible to reason just about the operational semantics of source programs, such that conclusions port soundly to compiled programs, while allowing us to link against programs written in other languages**. That is, consider any Cito program property of the form $\forall \Sigma, \Sigma'.\ (\Sigma, s) \Downarrow \Sigma' \Rightarrow P(\Sigma, \Sigma')$, for some P relating initial and final states. We have proved that compilation preserves any such property. P might be a Hoare-style program specification, proved semi-manually using a program logic. We then get for free that the compiled program satisfies the same property. Furthermore, we can implement *optimization passes* and prove that they preserve all such specifications. These proofs need only refer to the Cito operational semantics, not any of the details of our core compiler or of Bedrock. In this section, we demonstrate proofs of both kinds, as case studies for how effectively our compiler theorem encapsulates implementation details.

ExampleADT.v:

```
Inductive ADTModel :=
  | Arr : list W → ADTModel
  | FSet : MSet.t W → ADTModel
  ...
Definition ListSet_addSpec :=
  PRE[I] ∃ s n, I = [ADT (FSet s), SCA n]
  POST[O, R] ∃ s n any, O = [( ADT (FSet s), Some (FSet (
     add n s))), (SCA n, None)] ∧ R = SCA any.
```

CountUnique.v:

```
Definition count_spec :=
  PRE[I] ∃ arr len, I = [ADT (Arr arr), SCA len] ∧ len =
     length arr
  POST[O, R] ∃ arr, O[0] = (ADT (Arr arr), Some (Arr arr))
     ∧ R = SCA (count_unique arr).
Definition imports := [
   ("ArraySeq"!"read", ArraySeq_readSpec),
   ("ListSet"!"add", ListSet_addSpec), ... ]
Definition count :=
 cmodule "count" {{ [count_spec]
   cfunction "count"("arr", "len") return "ret" (*7*)
     "set" ← Call "ListSet"!"new"();; "i" ← 0;; (*8*)
     [INIT (V, H) NOW (V', H') ∃ arr fset,
        find (V "arr") H = Some (Arr arr) ∧
        H' == H * (V' "set" → FSet fset) ∧
        fset == to_set (firstn (V' "i") arr)]
     While ("i" < "len") { (*9*)
       "e" ← Call "ArraySeq"!"read" ("arr", "i");;
       Call "ListSet"!"add"("set", "e");; "i" ← "i" + 1
     };;
     "ret" ← Call "ListSet"!"size"("set");; (*11*)
     Call "ListSet"!"delete"("set") (*12*)
   end
 }}.
Definition count_compil := compile count imports.
Theorem count_ok : moduleOk count_compil.
  compile_ok. (* apply main compiler theorem,
    with automated proof of side conditions *)
Qed.
```

Figure 13. Implementation and verification of CountUnique example: the high-level Cito code part

Program Proof Example. We give an example of source-program verification using the "count unique values" example shown in Figure 1, to illustrate how to use our compiler to verify a program involving interaction between high-level and low-level pieces of code.

The actual Coq code for implementing and verifying the CountUnique program is excerpted in Figures 13 and 14. We leveraged Coq's extensible parser to define some notations for Cito's syntax so programmers can write the program and verify it entirely within Coq. The code is split in two parts, which can be verified separately. Figure 13 shows the high-level part written in Cito, while Figure 14

```
Definition main :=
  bimport [[ "count"!"count" @ [op_spec count_body], ... ]]
  bmodule "main" {{
    bfunction "main"("arr", "R") [topS] (*15*)
      ... (* initialize "arr" to [10, 20, 10] *)
      "R" ← Call "count"!"count"("arr", 3);; (*17*)
      Call "sys"!"printInt"("R");; (*17*)
      ...
    end
  }}.
Theorem main_ok : moduleOk main.
  (* manual proof, partly using program logic to
     derive axiomatic spec from operational *)
Qed.
Definition all := link [count_compil, main, list_set, ...].
Theorem all_ok : moduleOk all.
  link_ok [count_ok, main_ok, list_set_ok, ...].
Qed.
```

Figure 14. Implementation and verification of CountUnique example: low-level Bedrock (assembly) code part

shows the low-level part written in the old Bedrock notation. Bedrock has its own notations that make its programs – assembly programs in essence – look like C programs, but keep in mind the language and abstraction gap between the two parts. High-level code is proved according to an operational semantics exposing a heap of ADT values, while low-level code is proved according to an operational semantics exposing memory as an array of bytes. To highlight the correspondence with Figure 1, we put corresponding line numbers in Figure 1 as comments here.

In the high-level part, the function `count` relies on axiomatic specifications of ADT operations to implement its algorithm, and file `ExampleADT.v` defines the formal interface. Our compound ADT for this example is just the sum type of a series of Cito types acting as mathematical models. Here we use Coq's lists as the model for arrays and Coq's MSet finite-set library for the model of finite sets. `ListSet.add`'s specification describes its obvious behavior of adding the second argument to the first, and not allocating any new objects (by forcing the return value to be scalar). Elided here are specifications in the same file for all of the other ADT methods, which must be implemented in some language connected to Bedrock.

File `CountUnique.v` gives the main Cito function `count` and its specification `count_unique`, which is defined in Coq as (`fun ls ⇒ FSet.cardinal (FSet.of_list ls)`). The compiler takes as inputs the program and its import table, generating the target code and providing the correctness theorem. The compilation process works fine without additional axiomatic specification information, like `count_spec` and the loop invariant in the example; we include those annotations only to help us verify this specific program, and they are deleted before calling the compiler proper.

The only thing left to do is for the programmer to prove that the program meets its specification, with respect to the Cito operational semantics. Here she can use any of her favorite tools or methodologies for program proof. We implemented a simple proof-of-concept program logic, where the programmer annotates the program with an axiomatic specification for each function and an invariant for each loop. The program logic will generate a list of verification conditions for the programmer to discharge, all of which are pure logical implications that do not mention program syntax. Often semantic complications arise in defining a program logic that supports mutually recursive functions [27], but we have already tackled all of those challenges in designing the mixed operational and axiomatic semantics of Cito from Section 2. One key operation we use is a weakening of a Cito program context Ψ to replace an operational spec with an axiomatic spec implied by the original. This operation is useful to verify a recursive function, giving an operational proof of the body where we appeal to an axiomatic spec for any recursive call.

Figure 14 shows more standard Bedrock code defining a wrapper to call `count` on a particular input array and then linking together all program modules into a closed program with a partial-correctness theorem. Elided here is conventional Bedrock code [12, 19] to implement the different ADTs of our program. The module `list_set`, for instance, may apply arbitrary assembly code, so long as it is proved to satisfy the separation-logic contract given for axiomatic specs in the definition of funcsOk in Figure 12. For this example, we verified a simple unsorted-list implementation of finite sets, but any other implementation can be dropped in without any changes to other proofs.

The proof of theorem `main_ok`, establishing correctness of the main program function, is where we apply the program logic to deduce that the Cito implementation of `count` from Figure 13 implements the specification `count_spec` from the same figure. All of the proofs from Figure 13 are automated, and we only need to dive into manual proofs in Figure 14 to prove deeper, program-specific properties than preservation of Cito program behavior by the compiler, taking that preservation as given.

We also want to emphasize again that what we have done here is prove a *partial correctness* theorem for a full Bedrock IL program (`all` in the figures). The theorem is based on the specifications annotated on programs, and includes the following fact: if the call to `printInt` in Figure 14 is ever reached, the argument passed to it is 2 (the number of unique elements in our particular array). This theorem statement refers only to the operational semantics of Bedrock IL, so we do not need to worry that bugs in our compiler verification methodology could impugn it.

Optimization Examples. A key property of a compiler correctness proof style is *vertical compositionality*, where we want to verify different passes independently and compose their theorems to produce one about the whole compiler. Our approach in this paper supports vertical compositionality very well, as we demonstrated by verifying two standard source-to-source optimizers, constant folding and dead-code elimination. The example from earlier in this section and our other examples are all compiled via a pipeline that includes these optimizations, and we verify whole programs using black-box composition of the theorems for the different passes.

We specify the correctness of a Cito-to-Cito optimizer with a relation between its input and output statements. We define the *backwards preservation* relation between an input function with body s and return variable r, and an output function body s', as follows:

$$\forall \Psi, \Sigma, \Sigma'.\ \Psi \vdash (\Sigma, s') \Downarrow \Sigma' \\ \Rightarrow \exists \sigma''.\Psi \vdash (\Sigma, s) \Downarrow (\sigma'', \Sigma'.2) \wedge \sigma''(r) = \Sigma'.1(r) \quad (3)$$

It says that if s' terminates, the final state is one that could have resulted from running s, modulo the values of local variables (except r). We have found this property rather easy to prove directly (by rule induction) for our example optimizations, constant folding via a simple single-pass forward dataflow analysis with conservative treatment of loops, and dead-code elimination via a single-pass backward dataflow analysis. We see no obstacles in the way of similarly direct proofs for other classic optimizations, including with iterative dataflow analysis.

Furthermore, it is easy to prove a *transitivity* theorem showing that composing two optimizations that are sound in our sense leads to a new optimizer that is also sound. Our Coq proof of transitivity is about 10 lines long and requires no new induction or ingenuity in general.

9. Related Work

The best-known mechanically verified compiler is CompCert [17], where many of the core techniques of the field were introduced. CompCert's main theorem is not compositional, as it says nothing about the behavior of compiled modules when linked with code not produced by the same version of CompCert. Beringer et al. [8, 25] have extended CompCert with support for verified separate compilation, tackling several technical issues specific to the C language, like handling of pointers to local variables. While their compiler theorem is designed to support verification of multilanguage, compiled programs, they leave concrete proofs of that kind to future work. In a concurrent project, Ramananandro et al. [23] have designed a refinement-based framework for integrating certified compilation and modular program verification.

Benton and collaborators initiated the study of compositional compiler correctness in a series of papers [5–7] about compilers from functional languages to assembly languages and abstract machines. These proofs rely on step-indexed logical relations and biorthogonality, where we instead use a

program logic. Hur and Dreyer [16] presented a Kripke step-indexed logical relation intended for use in compositional verification of a compiler from ML to assembly. This line of work uses the static type of each source term to produce a specialized statement of compilation correctness, which allows, for instance, verified preservation of parametricity properties of polymorphic source code. This relation applies to a more intricate source language (ML) than ours, but it has not been used in mechanized proofs and is not known to support a transitivity theorem that would enable modular verification of compiler phases. We also want to emphasize that Benton's definition of *compositional compiler correctness* does not mean *quite* the same thing as ours, since it depends critically on static types, while we treat an untyped language and focus on simulations between source and target program behaviors, in a sense hybridizing the approaches of CompCert and Benton et al.

Ahmed and Blume have studied the problem of *fully abstract compilation* for functional languages [1, 2], which implies that equivalent source programs are mapped to equivalent target programs, no matter which contexts malicious users might place those target programs into. The proof techniques based on different varieties of logical relations have much in common with those mentioned above for compositional compiler correctness.

Chlipala has verified compilers for non-Turing complete functional languages [9, 10], using normal logical relations and thus gaining some compositionality, though of course the source languages are too simple to expose the main challenges of compositional correctness. Past non-compositional mechanized proofs of verified compilers for functional languages include those by Flatau [15], Minamide and Okuma [20], Tian [26], Dargaye and Leroy [14], and Chlipala [11]. Myreen and Gordon presented a verified Lisp interpreter [21].

10. Conclusion

We have presented a new approach to integrating multiple languages and compilers within a framework for Hoare-style program proof. A compiler correctness proof should be independent of details of other compilers it may be used in concert with, but providing this intuitive property is not trivial. To allow us to verify particular source programs whose correctness depends on the behavior of functions implemented in other languages, we give our source language a *mixed operational and axiomatic semantics*, modeling cross-language function calls only through the axiomatic features. To allow different languages to use different encodings of mutable objects in memory, we build *data abstraction* into our semantics and compiler proof. The source-language semantics sees foreign functions mutating mathematical models of *abstract data types*, while our compiler theorem formalizes the requirements on code in other languages to implement those types by giving *representation invariants* in separation logic.

We plan to build verified compilers for higher-level languages on top of Cito, taking advantage of this paper's compiler correctness theorem to reason about Cito only in terms of its operational semantics.

Acknowledgments

We thank Christian J. Bell, Lennart Beringer, and Benjamin Delaware for their feedback on earlier versions of this paper. This work has been supported in part by NSF grant CCF-1253229, DARPA under agreement number FA8750-12-2-0293, and the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program, under Award Number DE-SC0008923. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] A. Ahmed and M. Blume. Typed closure conversion preserves observational equivalence. In *Proc. ICFP*, pages 157–168. ACM, 2008.

[2] A. Ahmed and M. Blume. An equivalence-preserving CPS translation via multi-language semantics. In *Proc. ICFP*, pages 431–444. ACM, 2011.

[3] A. W. Appel. Verified software toolchain. In *Proc. ESOP*, volume 6602 of *LNCS*, pages 1–17. Springer-Verlag, 2011.

[4] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS*, 23 (5):657–683, Sept. 2001.

[5] N. Benton and C.-K. Hur. Biorthogonality, step-indexing and compiler correctness. In *Proc. ICFP*, pages 97–108. ACM, 2009.

[6] N. Benton and C.-K. Hur. Realizability and compositional compiler correctness for a polymorphic language. Technical Report MSR-TR-2010-62, Microsoft Research, 2010.

[7] N. Benton and N. Tabareau. Compiling functional types to relational specifications for low level imperative code. In *Proc. TLDI*. ACM, 2009.

[8] L. Beringer, G. Stewart, R. Dockins, and A. W. Appel. Verified compilation for shared-memory C. In *Proc. ESOP*, 2014.

[9] A. Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Proc. PLDI*, pages 54–65, 2007.

[10] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proc. ICFP*, pages 143–156, 2008.

[11] A. Chlipala. A verified compiler for an impure functional language. In *Proc. POPL*, pages 93–106, 2010.

[12] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proc. PLDI*, pages 234–245. ACM, 2011.

[13] A. Chlipala. The Bedrock structured programming system: Combining generative metaprogramming and Hoare logic in an extensible program verifier. In *Proc. ICFP*, pages 391–402. ACM, 2013.

[14] Z. Dargaye and X. Leroy. Mechanized verification of CPS transformations. In *Proc. LPAR*, pages 211–225, 2007.

[15] A. D. Flatau. *A Verified Implementation of an Applicative Language with Dynamic Storage Allocation*. PhD thesis, University of Texas at Austin, Nov. 1992.

[16] C.-K. Hur and D. Dreyer. A Kripke logical relation between ML and assembly. In *Proc. POPL*, pages 133–146. ACM, 2011.

[17] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. POPL*, pages 42–54. ACM, 2006.

[18] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, Feb. 2009.

[19] G. Malecha, A. Chlipala, and T. Braibant. Compositional computational reflection. In *Proc. ITP*, pages 374–389, 2014.

[20] Y. Minamide and K. Okuma. Verifying CPS transformations in Isabelle/HOL. In *Proc. MERLIN*, pages 1–8, 2003.

[21] M. O. Myreen and M. J. Gordon. Verified LISP implementations on ARM, x86 and PowerPC. In *Proc. TPHOLs*, pages 359–374. Springer-Verlag, 2009.

[22] Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *Proc. POPL*, pages 320–333. ACM, 2006.

[23] T. Ramananandro, Z. Shao, S. Weng, and J. Koenig. A compositional semantics for verified separate compilation and linking. Technical Report YALEU/DCS/TR-1494, Dept. of Computer Science, Yale University, New Haven, CT, January 2014.

[24] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS*, pages 55–74. IEEE Computer Society, 2002.

[25] G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional CompCert. Technical report, July 2014.

[26] Y. H. Tian. Mechanically verifying correctness of CPS compilation. In *Proc. CATS*, pages 41–51, 2006.

[27] D. Von Oheimb. Hoare logic for mutual recursion and local variables. In *Foundations of Software Technology and Theoretical Computer Science*, pages 168–180. Springer, 1999.

A. Notations from Coq Code Examples

Example code in Figures 13 and 14 uses some notations that we do not have space to introduce in the main paper. We hope they are reasonably straightforward to deduce from context, but we also introduce them more explicitly in this appendix.

In Figure 13, axiomatic specifications are defined by notation "`PRE[I] _ POST[O,R] _`", where `PRE` is the precondition part and `POST` is the postcondition part. `I` is the input values. `O` is the input-output pairs. `R` is the return value. Constraints already expressed in the precondition can be omitted in the postcondition, as `count_spec` does, since the operational semantics requires both to be met.

Cito modules are defined like so:

```
cmodule NAME {{
  [SPEC1]
  cfunction NAME1 (ARG1, ARG2, ...)  return RET
    ...
  end with
  [SPEC2]
  cfunction NAME2 ...
}}
```

`cmodule` and `cfunction` stand for "Cito module" and "Cito function," respectively. Each function is annotated with an axiomatic specification at the head. Note, however, that our compiler is perfectly able to handle Cito code that is *not* annotated with specifications; we only include the annotations here since we plan to reason about the example with a program logic.

In the body of a `cfunction`, we use binary operators ;; for statement sequencing and ← for assignment. Program variable names appear in double quotes, to appease Coq's lexer.

Loops are annotated with loop invariants in the form "`INIT(V,H) _ NOW(V',H') _`", where the `INIT` part describe the state at the beginning of function execution, and the `NOW` part describes the state at the start of each loop iteration. Each part binds a name `V` for the local variable environment and `H` for the Cito heap. A local variable environment is used as a function, which we call on a variable name to retrieve its value.

In the loop invariant, notation `w → a` denotes a singleton map from address `w` to ADT object `a`. Notation `H == H1 * H2 * ...` indicates that map `H` is the union of maps `H1`,`H2`,... and maps `H1`,`H2`,... are pairwise disjoint. == is used to distinguish map/set equivalence (permutation-insensitive) from normal equality (=).

Notation `A!B` is used for choosing function `B` from module `A`.

Some notations here are simplified from those that appear in the source code, to help keep the figures more self-contained.

B. Step-by-Step Usage Guide

The *CountUnique.v* file in the source code tarball (of which Figure 13 is just an excerpt) shows a complete workflow of using the compiler to compile a program. The workflow is summarized below:

- **Step 1:** Write the Cito program. Example: `count_body`, `main_body`, and `m` in *CountUnique.v*.
- **Step 2:** Prove a syntactic well-formedness property of the program. Example: `good` in *CountUnique.v*.

- **Step 3:** Declare imported specifications of external functions that the program calls. Example: `imports` in *CountUnique.v*. Those imports typically include specifications of ADT methods from the standard library. (The methods themselves are implemented and verified in a conventional Bedrock module, in *platform/cito/examples/ExampleImpl.v*.)
- **Step 4:** Invoke the compiler using `link_with_adts`. Example: the line under comment "Invoke the compiler" in *CountUnique.v*.

In Step 1, we can write the program using Cito's syntax or, as *CountUnique.v* does, using another syntax for Cito that supports annotation of programs with assertions, for verification purposes later. (Repeating ourselves a bit from the main paper, we remind the reader that the compiler correctness theorem applies to unannotated programs! We erase annotations before calling the main compiler.) Step 2 corresponds to the static checking phase (syntax checking, type checking, static analysis, etc.) of conventional languages. Since it is a decidable problem, we will provide a decider (acting as an automatic checker) in Gallina (Coq's typed functional programming language) in the future. For now, we call a heuristic proof automation procedure `good_module`, implemented in the dynamically typed tactic language Ltac.

At this point, we already have the fact that the generated assembly program simulates the source program, without doing any verification work. However, since we know nothing about the source program, we cannot get any interesting property of the target program yet. The next portion of the workflow is for verifying the source program:

- **Step 5:** Use the compiled module in some Bedrock code. Example: `top` in *CountUnique.v*. A Bedrock module definition needs declarations of a specification for every external function called in the module. Here we can use the default specifications automatically generated by the compiler for every source Cito function, such as `main_spec_Bedrock` in *CountUnique.v*. In the Bedrock code one typically expresses the expected behavior of the source program via some Bedrock assertions, such as the line under the call to `main`.
- **Step 6:** Verify the Bedrock code. Example: `top_ok` in *CountUnique.v*. In the process of verifying the Bedrock code, the biggest job is to show that it is OK to call the compiled functions of the source program. This amounts to proving safe/runsto lemma pairs, such as `main_safe` and `main_runsto` in *CountUnique.v*. The former shows that the Cito code avoids the equivalents of "undefined behavior" in C, while the latter shows that it actually computes the correct answer.

Our compiler story ends here, from which point it is possible to use any tools to prove the safe/runsto lemma pairs, which are standard source-program verification. For *CountUnique.v*, we designed a simple program logic (in *platform/cito/ProgramLogic2.v* and *platform/cito/SemanticFacts4.v*) that supports annotation of source program with preconditions, postconditions, loop invariants, and assertions, generates verification conditions (VC) from those annotations, and guarantees the program correct when one proves all the VCs. This facility is still primitive and hard to use. We are working on making this last (and most substantial) step easier for the user.

If one wants to run the program and see the actual result on a 64-bit x86 machine, there are some extra steps:

- **Step 7:** Provide driver files. Example: *CountUniqueDriver.v* and *CountUniqueAMD64.v* in directory *platform/cito/examples*. The former defines a small entry-point function for the program, which needs to set up the stack and heap before calling the code from *CountUnique.v*.
- **Step 8:** Build the executable, following instructions in file *INSTALL*.

Note that *CountUniqueDriver.v* shows an example of a final, assembly-level theorem about a whole program. The final theorem `safe` shows that the program runs safely without out-of-bounds accesses to program or data memory, when started in the entry-point function.

C. Building Executable Programs

The example programs in the source code (`http://people.csail.mit.edu/wangpeng/oopsla2014.tgz`), including the CountUnique example, can be compiled to assembly and run, on AMD64 Linux platforms (and possibly others that we have not tested yet). Detailed build instructions and a summary of source-code structure can be found in `<SOURCE>/platform/cito/README`.