A Minimalistic Verified Bootstrapped Compiler (Proof Pearl)

Magnus O. Myreen
Chalmers University of Technology
Gothenburg, Sweden

Abstract

This paper shows how a small verified bootstrapped compiler can be developed inside an interactive theorem prover (ITP). Throughout, emphasis is put on clarity and minimalism.

CCS Concepts: • **Theory of computation** → *Program verification*; *Higher order logic*; *Automated reasoning*.

Keywords: compiler verification, compiler bootstrapping, interactive theorem proving

ACM Reference Format:
Magnus O. Myreen. 2021. A Minimalistic Verified Bootstrapped Compiler (Proof Pearl). In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '21), January 18–19, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3437992.3439915

1 Introduction

Bootstrapping is a milestone in any compiler development. We say that a compiler bootstraps itself when it can generate its own low-level implementation by applying itself to its own source code [8, 14].

In traditional compiler development, the bootstrapping milestone means that the compiler can, from then on, be expressed in its own source language and no longer needs to rely on another compiler for development. Due to this independence, bootstrapped compilers are called self hosting.

In the context of verified compilation, compiler bootstrapping also means that one can arrive at a low-level executable implementation of the compiler without the use of another code-generation path. Verified compilers live within the logic of interactive theorem provers (ITPs), and, even though compilers can be run in this setting, it is often more convenient to have a way to use them outside of ITPs. By evaluating the compiler on itself within the ITP (i.e. bootstrapping it), one can arrive at an implementation of the compiler inside the ITP and get a proof about the correctness of each step [13]. From there, one can export the low-level implementation of the compiler for use outside the ITP, without involving any complicated unverified code generation path.

The concept of applying a compiler to itself inside an ITP can be baffling at first, but don't worry: the point of this paper is to clearly explain the ideas of compiler bootstrapping on a simple and minimalistic example.

To the best of our knowledge, compiler bootstrapping inside an ITP has previously only been done as part of the CakeML project [29]. The CakeML project has as one of its aims to produce a realistic verified ML compiler and, as a result, some of its contributions, including compiler bootstrapping, are not as clearly explained as they ought to be: important theorems are cluttered with real-world details that obscure some key ideas.

The contribution of this paper is a new verified bootstrapped compiler that is designed to clearly explain the concept of compiler bootstrapping inside an ITP. This paper is not aiming for generality, realism or good performance, but instead strives for clarity and minimalism.

The result of this effort is a mechanised proof development that produces verified x86-64 assembly code implementing our new verified compiler. Crucially, the assembly implementation of the compiler is produced via bootstrapping inside the logic of an ITP. That is, the compiler is run on itself within the logic to produce its own implementation in assembly. This is all done with proof, and, as a result, we arrive at a theorem which guarantees correct behaviour of the assembly implementation of the compiler.

This work has been carried out in the HOL4 theorem prover [25] but the ideas ought to translate to other provers such as Coq [17], Isabelle [30] and ACL2 [18]. Our proof scripts are under examples/bootstrap in the sources of HOL4.

2 Idea of Bootstrapping in the Logic

In this section, we start with a look at how the idea of compiler bootstrapping works for our new verified compiler. This section focuses on how all the different parts fit together, while subsequent sections explain the definitions and theorems that we build on in this section.

The text below describes the top-level compiler definition for our new little compiler; the relevant correctness theorem

CPP '21, January 18–19, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8299-1/21/01…$15.00
https://doi.org/10.1145/3437992.3439915

for the code generator that the compiler contains; how one can apply the compiler to itself; and what theorems come out at the end of bootstrapping inside an ITP.

Compiler definition. Our new compiler is defined as functions in logic. The top-level function is the following:

$$\begin{array}{l}\mathsf{compiler}\ \mathit{input} \stackrel{\text{def}}{=} \\ \quad \mathsf{asm2str}\ (\mathsf{codegen}\ (\mathsf{parser}\ (\mathsf{lexer}\ \mathit{input})))\end{array}$$

This compiler function has type string → string, and the internal functions have the following types:

$$\begin{array}{rcl}\mathsf{lexer} & : & \mathsf{string} \rightarrow \mathsf{token\ list} \\ \mathsf{parser} & : & \mathsf{token\ list} \rightarrow \mathsf{prog} \\ \mathsf{codegen} & : & \mathsf{prog} \rightarrow \mathsf{asm} \\ \mathsf{asm2str} & : & \mathsf{asm} \rightarrow \mathsf{string}\end{array}$$

Here prog is a datatype for the abstract syntax tree (AST) of source programs (which we will see in Section 3), and asm is the AST for x86-64 assembly (to be seen in Section 5).

In addition to the compiler functions, we also define a pretty printing function that converts a source program to a string. The second argument (of type string list) is a list of comments to inject into the generated string.

$$\mathsf{prog2str}\ :\ \mathsf{prog}\ \rightarrow\ \mathsf{string\ list}\ \rightarrow\ \mathsf{string}$$

We have proved that the lexer followed by the parser inverts prog2str, regardless of the comments *coms*:

$$\vdash \mathsf{parser}\ (\mathsf{lexer}\ (\mathsf{prog2str}\ p\ \mathit{coms})) = p$$

Correctness of code generation. The in-logic compiler bootstrapping that we do requires a correctness theorem for the code generator, codegen. The required correctness theorem relates terminating executions of the source language with the terminating executions of the target language.

Before we can show the codegen correctness theorem, we need to introduce how we make statements about program execution in the source and target languages. For source-level programs (of type prog), we write:

$$(\mathit{input}, p) \Downarrow_{\mathsf{prog}} \mathit{output}$$

to say that, with *input* available on stdin, source program p terminates and produces *output* on stdout. Similarly, for target-level assembly programs (of type asm), we write:

$$(\mathit{input}, a) \Downarrow_{\mathsf{asm}} \mathit{output}$$

to say that, with *input* to be read on stdin, target program a successfully terminates and produces *output* on stdout.

We use the following correctness theorem for codegen in our compiler bootstrapping. This theorem states that, if execution of source program p terminates and target-level execution of assembly program codegen p terminates, then the outputs produced by the two executions must be equal.

$$\begin{array}{l}\vdash (\mathit{input}, p) \Downarrow_{\mathsf{prog}} \mathit{output}_1 \wedge \\ \quad (\mathit{input}, \mathsf{codegen}\ p) \Downarrow_{\mathsf{asm}} \mathit{output}_2 \Rightarrow \\ \quad\quad \mathit{output}_1 = \mathit{output}_2\end{array}$$

The operational semantics mentioned above, i.e. $\Downarrow_{\mathsf{prog}}$ and $\Downarrow_{\mathsf{asm}}$, are explained in Section 3 and 5. The proof of the correctness theorem above is the topic of Section 6.

While the correctness theorem shown above is sufficient for compiler bootstrapping, it is not quite satisfactory in general. For example, it does not say anything about non-terminating executions. Section 7 shows how preservation of non-terminating behaviour can be proved for codegen.

The compiler in AST form. In order to apply the compiler to itself, we need to somehow get the compiler into a form that fits with what the compiler function takes as input.

The compiler function has a *function type*: string → string. It would be type incorrect to attempt to apply compiler directly to compiler, since the input type, string, does not match the type of the argument, string → string.

The key to this puzzle is to define a new constant, which we call compiler_prog, that is the compiler represented as source AST, i.e. a value of type prog:

$$\mathsf{compiler\_prog}\ :\ \mathsf{prog}$$

We define compiler_prog in such a way that we can prove that it implements the compiler function:

$$\vdash (\mathit{input}, \mathsf{compiler\_prog}) \Downarrow_{\mathsf{prog}} \mathsf{compiler}\ \mathit{input}$$

One should read the theorem above as saying: for any *input*, execution of the compiler_prog program will always terminate and the output on stdout is the string produced by applying the compiler function to *input*.

Section 4 describes how we produce compiler_prog and prove the correctness theorem above for it.

Applying the compiler to itself. Using compiler_prog, we can define a concrete-syntax version of it, compiler_str; a version expressed in assembly, compiler_asm; and the string representation of the assembly version, compiler_asm_str:

$$\mathsf{compiler\_str} \stackrel{\text{def}}{=} \mathsf{prog2str\ compiler\_prog\ coms}$$

$$\mathsf{compiler\_asm} \stackrel{\text{def}}{=} \mathsf{codegen\ compiler\_prog}$$

$$\mathsf{compiler\_asm\_str} \stackrel{\text{def}}{=} \mathsf{asm2str\ compiler\_asm}$$

Note that compiler_asm is applying part of the compiler, namely codegen, to the compiler itself.

From the definition of compiler, the definitions above, and the correctness of prog2str, we can prove an equation describing the result of applying the entire compiler to itself:

$$\vdash \mathsf{compiler\ compiler\_str} = \mathsf{compiler\_asm\_str}$$

This result is reassuring, but it is not on the critical path to the main bootstrap theorem, which we will explain next.

The bootstrap theorem. The result of bootstrapping a compiler inside an ITP is a theorem stating that the *low-level implementation* of the compiler correctly *implements* the compiler *algorithm*. In our case, the low-level implementation is the compiler expressed in assembly, compiler_asm, and the compilation algorithm is the compiler function.

We can easily arrive at the desired theorem by combining the correctness theorems for codegen and compiler_prog to prove the following statement. This final theorem states that compiler_asm implements the compiler function.

$$\vdash (\mathit{input},\mathsf{compiler\_asm}) \Downarrow_{\mathsf{asm}} \mathit{output} \Rightarrow \\ \mathit{output} = \mathsf{compiler}\ \mathit{input}$$

Here it is worth noting that there is a hidden form of partiality in this theorem. This partiality stems from the assumption expressed in terms of $\Downarrow_{\mathsf{asm}}$. The precise reading of the statement above is: if evaluation of compiler_asm *successfully terminates*, then the output has the desired content. We know that compiler_asm avoids diverging, but we do not know whether it will call the system `exit` function with a zero exit code. Our semantics (to be defined in Section 5) considers an execution successful if it has zero as the exit code. The codegen function emits code that resorts to a non-zero exit code when memory has been exhausted during execution, e.g. when the code for allocating a new heap object is called in a state where there is no heap space left. Due to the dynamic way most compilers tend to use memory, it seems unlikely that we can prove memory bounds that would allow us to stay clear of this partiality.

Evaluation. Finally, we want to get our hands on the concrete low-level implementation of the compiler. From the development described above, we know that compiler_asm_str is the string representation of the verified assembly program. However, to run this assembly outside of the logic, we need to have it as a concrete string that we can print to a file. In order to get this string, we simply evaluate the term "compiler_asm_str" inside the ITP using the ITP's rewrite engine. For our case, this evaluation takes less than two minutes and results in a string consisting of 200 815 characters.

Once we have this string, we can put it in a textfile, pass that textfile as input to the GNU assembler, then link the resulting object, and finally run the compiler from the Linux[1] command-line like any other program, see Section 8.

3 Source Language and Its Semantics

We now move on to the technical details. This section presents the definition of the source language.

Design. The source language is designed to be as small as possible under the constraint that an entire compiler needs to be implemented in the source language. We decided to take inspiration from simple Lisp languages since their implementation can be kept small and they can quite naturally express the AST traversals that a compiler performs.

[1]Our definition of asm2str is specific to the syntax used by the GNU assembler and the Linux execution environment for the x86-64 architecture.

Values. We define the source language to operate over values that are binary trees with natural numbers at the leaves. In our formal semantics, we define the semantic value type v as the following recursive datatype.

$$\mathsf{v} \;=\; \mathsf{Pair\ v\ v} \mid \mathsf{Num\ nat}$$

Abstract syntax. The AST for the source language is the following. A complete program (prog) is a list of function declarations (dec list) followed by a main expression (exp):

$$\mathsf{prog} \;=\; \mathsf{Program\ (dec\ list)\ exp}$$
$$\mathsf{dec} \;=\; \mathsf{Defun\ fname\ (vname\ list)\ exp}$$

Note that fname and vname are abbreviations for nat, i.e. all names are represented as natural numbers in the AST. The concrete syntax allows for alphanumeric names, but those are read by the lexer as natural numbers written in base 256.

The expression type exp, the primitive operations op, and the comparisons test are defined as follows.

exp	=	Const nat	natural number
	|	Var vname	variable
	|	Op op (exp list)	primitive op.
	|	If test (exp list) exp exp	if-expression
	|	Let vname exp exp	let-binding
	|	Call fname (exp list)	function call
op	=	Add | Sub | Div	+, -, div for nat
	|	Cons | Head | Tail	heap operations
	|	Read | Write	char-based I/O
test	=	Less | Equal	comparisons

We did not need a primitive for multiplication.

Semantics. We define $\Downarrow_{\mathsf{prog}}$ to say that a whole program terminates with output, if evaluation of the main expression, using $\Downarrow_{\mathsf{exp}}$, produces that output in its state: s.output.

$$(\mathit{input},\mathsf{Program}\ \mathit{funs}\ \mathit{main}) \Downarrow_{\mathsf{prog}} \mathit{output} \stackrel{\text{def}}{=} \\ \exists\, s\ r. \\ (\mathsf{empty\_env},[\mathit{main}],\mathsf{init\_state}\ \mathit{input}\ \mathit{funs}) \Downarrow_{\mathsf{exp}} (r,s) \wedge \\ \mathit{output} = s.\mathsf{output}$$

Our big-step relational semantics for expression evaluation, $\Downarrow_{\mathsf{exp}}$, relates an environment env, a list of expressions exps and a starting state s to a list of values vals and a final state s' that are the result of fully evaluating exps.

$$(\mathit{env},\mathit{exps},s) \Downarrow_{\mathsf{exp}} (\mathit{vals},s')$$

The type of states is defined as follows. States carry the input as a potentially infinite list of characters; the output is a string (i.e. a finite list of characters); and the state also contains all function declarations. The purpose of the clock field will be explained in Section 7.

$$\mathsf{state} = \langle\!|\ \mathsf{input} : \mathsf{char\ llist};\ \mathsf{output} : \mathsf{string}; \\ \mathsf{funs} : \mathsf{dec\ list};\ \mathsf{clock} : \mathsf{nat}\ |\!\rangle$$

The initial state, init_state, has output set to the empty string.

$$\text{init\_state } \mathit{input}\ \mathit{funs} \stackrel{\text{def}}{=} \langle\!|\ \text{input} := \mathit{input};\ \text{output} := \text{``''};\ \text{funs} := \mathit{funs};\ \text{clock} := 0\ |\!\rangle$$

The $\Downarrow_{\text{exp}}$ relation is defined inductively and is not particularly surprising. The rule for Const is:

$$\frac{}{(\mathit{env},[\text{Const } n],s) \Downarrow_{\text{exp}} ([\text{Num } n],s)}$$

Similarly, the rule for Var is simple:

$$\frac{\mathit{env}\ n = \text{Some } v}{(\mathit{env},[\text{Var } n],s) \Downarrow_{\text{exp}} ([v],s)}$$

The rule defining Call is expressed with the help of an auxiliary relation app (to be used in the next section).

$$\frac{(\mathit{env},\mathit{xs},s_1) \Downarrow_{\text{exp}} (\mathit{vs},s_2) \quad \text{app } \mathit{fname}\ \mathit{vs}\ s_2\ (v,s_3)}{(\mathit{env},[\text{Call } \mathit{fname}\ \mathit{xs}],s_1) \Downarrow_{\text{exp}} ([v],s_3)}$$

The app relation is defined as follows.

$$\frac{\begin{array}{c}\text{env\_and\_body } \mathit{fname}\ \mathit{vs}\ s_1 = \text{Some } (\mathit{env},\mathit{body}) \\ (\mathit{env},[\mathit{body}],s_1) \Downarrow_{\text{exp}} ([v],s_2)\end{array}}{\text{app } \mathit{fname}\ \mathit{vs}\ s_1\ (v,s_2)}$$

The final rule that we include here is that of Op:

$$\frac{(\mathit{env},\mathit{xs},s_1) \Downarrow_{\text{exp}} (\mathit{vs},s_2) \quad \text{eval\_op } f\ \mathit{vs}\ s_2 = (\text{Res } v,s_3)}{(\mathit{env},[\text{Op } f\ \mathit{xs}],s_1) \Downarrow_{\text{exp}} ([v],s_3)}$$

Evaluation of the primitives is defined by eval_op. Below are some of the equations of its definition.

$$\begin{array}{l}
\text{eval_op Cons } [x;\ y]\ s \stackrel{\text{def}}{=} (\text{Res (Pair } x\ y),s) \\
\text{eval_op Head [Pair } x\ y]\ s \stackrel{\text{def}}{=} (\text{Res } x,s) \\
\text{eval_op Tail [Pair } x\ y]\ s \stackrel{\text{def}}{=} (\text{Res } y,s) \\
\text{eval_op Div [Num } n_1;\ \text{Num } n_2]\ s \stackrel{\text{def}}{=} \\
\quad \text{if } n_2 \neq 0 \text{ then (Res (Num } (n_1 \text{ div } n_2)),s) \\
\quad \text{else (Err Crash},s)
\end{array}$$

Applying a primitive incorrectly, e.g. applying Tail to a number Num, results in Err Crash.

4 The Compiler Expressed in AST

Section 2 made use of a constant, compiler_prog, of type prog for which we have the following theorem.

$$\vdash (\mathit{input},\text{compiler\_prog}) \Downarrow_{\text{prog}} \text{compiler } \mathit{input}$$

This section explains how we produce this constant and how we prove that theorem.

Approach. There are two directions one can take to produce such a constant: one can (D1:) generate AST from concrete syntax using the lexer and parser, and then verify it interactively against the source semantics $\Downarrow_{\text{prog}}$ or in a program logic that is built on top of the source semantics; or (D2:) use a tool (like [19]) that synthesises source AST from the definition of compiler function and automatically proves that the generated AST is correct w.r.t. $\Downarrow_{\text{prog}}$ (and $\Downarrow_{\text{exp}}$).

We decided to take a hybrid approach where the AST for all pure functions is produced using method D2 and the AST representation of all impure functions (i.e. I/O functions) is produced using method D1. Our implementation of the compiler function only touches I/O in the implementation of the lexer, which reads characters from stdin, and a simple print routine, which prints a list of characters to stdout.

Automation for code synthesis. We explain the approach we use for code synthesis using an example. Our example is the synthesis of AST for the following definition of even.

$$\text{even } n \stackrel{\text{def}}{=} \text{if } n = 0 \text{ then T else } \neg\text{even } (n - 1)$$

The workhorse of the proof-producing code synthesis automation is a routine that builds theorems of the following form. Here tm is the HOL term that we are synthesising AST for, x is the AST expression that we have generated, and $\mathit{encoding}$ is an appropriate function for encoding the HOL term into the value type v of the source semantics.

$$(\mathit{env},x,s) \Downarrow_{\text{exp}} (\mathit{encoding}\ \mathit{tm},s)$$

We want to apply this routine to the right-hand side of the definition for even, i.e. if $n = 0$ then T else ¬even $(n - 1)$. Before we can apply it, we need to have $\mathit{encoding}$ functions for all the types that appear in this term. The types that appear are nat and bool. The Num constructor function works for the nat type. For the bool type, we define a function, called Bool, that maps true (T) to 1 and false (F) to 0.

$$\begin{array}{l}
\text{Bool T} \stackrel{\text{def}}{=} \text{Num } 1 \\
\text{Bool F} \stackrel{\text{def}}{=} \text{Num } 0
\end{array}$$

Furthermore, we need to have lemmas for all of the functions and constants that appear in the term. Below are some of the lemmas that this application of the automation uses. The lemma for producing code for T is:

$$\vdash (\mathit{env},[\text{Const } 1],s) \Downarrow_{\text{exp}} ([\text{Bool T}],s)$$

and the lemma for producing Boolean negation (¬) is:

$$\begin{array}{l}
\vdash (\mathit{env},[x],s) \Downarrow_{\text{exp}} ([\text{Bool } b],s) \Rightarrow \\
\quad (\mathit{env},[\text{Op Sub [Const } 1;\ x]],s) \Downarrow_{\text{exp}} ([\text{Bool } (\neg b)],s)
\end{array}$$

Given the lemmas above, our automation can, e.g., process input ¬T. For this input, it proves the following $\Downarrow_{\text{exp}}$-theorem which shows how ¬T can be implemented in AST.

$$\vdash (\mathit{env},[\text{Op Sub [Const } 1;\ \text{Const } 1]],s) \Downarrow_{\text{exp}} ([\text{Bool } (\neg\text{T})],s)$$

Equipped with enough such lemmas, the workhorse of the automation derives the following theorem for the right-hand side of even. We have abbreviated the generated AST in a constant called even_code and replaced the right-hand side of even with its left-hand side, i.e. even n. The assumption on the theorem below includes an app because of the recursive call. Here N and EVEN abbreviate the respective natural number representations of strings "n" and "even".

$$\begin{array}{l} \vdash\ env\ \mathsf{N} = \mathsf{Some}\ (\mathsf{Num}\ n) \land \\ \quad (n \neq 0 \Rightarrow \\ \qquad \mathsf{app\ EVEN}\ [\mathsf{Num}\ (n-1)]\ s\ (\mathsf{Bool}\ (\mathsf{even}\ (n-1)), s)) \Rightarrow \\ \quad (env, [\mathsf{even\_code}], s) \Downarrow_{\mathsf{exp}} ([\mathsf{Bool}\ (\mathsf{even}\ n)], s) \end{array}$$

From the theorem above, one can easily derive a new version where the last line is phrased like the app-assumption:

$$\begin{array}{l} \vdash\ \mathsf{lookup\_fun\ EVEN}\ s.\mathsf{funs} = \mathsf{Some}\ ([\mathsf{N}], \mathsf{even\_code}) \land \\ \quad (n \neq 0 \Rightarrow \\ \qquad \mathsf{app\ EVEN}\ [\mathsf{Num}\ (n-1)]\ s\ (\mathsf{Bool}\ (\mathsf{even}\ (n-1)), s)) \Rightarrow \\ \quad \mathsf{app\ EVEN}\ [\mathsf{Num}\ n]\ s\ (\mathsf{Bool}\ (\mathsf{even}\ n), s) \end{array}$$

We can remove the app-assumption by applying the induction that arises from the termination proof of even, i.e.

$$\vdash (\forall\, n.\ (n \neq 0 \Rightarrow P\ (n-1)) \Rightarrow P\ n) \Rightarrow \forall\, v.\ P\ v$$

and arrive at:

$$\begin{array}{l} \vdash\ \mathsf{lookup\_fun\ EVEN}\ s.\mathsf{funs} = \mathsf{Some}\ ([\mathsf{N}], \mathsf{even\_code}) \Rightarrow \\ \qquad \mathsf{app\ EVEN}\ [\mathsf{Num}\ n]\ s\ (\mathsf{Bool}\ (\mathsf{even}\ n), s) \end{array}$$

which is the theorem returned by our proof automation as a certificate that even_code is a correct implementation of the function we gave as input, i.e. even.

Most of the functions in the compiler are more complicated than the even function, but the steps taken by the automation are still the same. For more complicated functions, many of the details are instead more verbose: the encoding functions are more complicated; the lemmas are longer; and the induction applied at the end is messier. Below is an example of one of the equations of the encoding function Exp for the exp type. The point of Exp is to define how the recursive exp type is represented in our Lisp's values. Here we use LET to abbreviate the number representation of the string "Let".

$$\begin{array}{l} \mathsf{Exp}\ (\mathsf{Let}\ n\ x\ y) \stackrel{\mathrm{def}}{=} \mathsf{Pair}\ (\mathsf{Num\ LET}) \\ \qquad\qquad\qquad\qquad (\mathsf{Pair}\ (\mathsf{Num}\ n) \\ \qquad\qquad\qquad\qquad\quad (\mathsf{Pair}\ (\mathsf{Exp}\ x) \\ \qquad\qquad\qquad\qquad\qquad (\mathsf{Pair}\ (\mathsf{Exp}\ y)\ (\mathsf{Num}\ 0)))) \end{array}$$

The lemmas used by the automation are also more complicated, particularly lemmas used for code generation for let-expressions and case-expressions where new variable bindings are introduced in the generated AST.

Impure functions. No proof automation was developed for the impure functions because there are only a few of them. The `print` function (displayed below in Lisp-inspired concrete syntax) is one of the few impure functions. It prints a list of characters to stdout and treats 0 as indication of the end of the list. Here `let` is used for sequencing.

```
(defun print (s)
  (if (= s '0) '0
    (let (v (write (head s)))
      (print (tail s)))))
```

We verifed all of the impure functions by interactive proof directly over the definition of the source semantics (D1).

The main expression. Programs in our source semantics end in a main expression. The main expression for the compiler implementation is the following. Here the innermost call to the lexer function takes no arguments, since it reads its input from stdin directly, and the output is written to stdout using the `print` function shown above.

```
(print (asm2str (codegen (parser (lexer)))))
```

Given a main expression, our automation assembles all of the generated pure functions and all of the parsed impure functions, and then defines the compiler_prog constant.

Once the constant compiler_prog is defined, we interactively prove the following correctness theorem for it using the app-theorems for each function used in the main expression. Note that there is no precondition on *input* here, i.e. compiler_prog will always do whatever compiler does.

$$\vdash (input, \mathsf{compiler\_prog}) \Downarrow_{\mathsf{prog}} \mathsf{compiler}\ input$$

5 Target Language and Its Semantics

This section takes a look at the target language of the code generator. The target language is a very small subset of the assembly language for the x86-64 architecture.

Design. We had two main goals when picking and formalising the target language: (1) we wanted to pick an assembly language that allows us to easily run the resulting programs on readily available hardware, and (2) to keep the formalised language subset as minimal as possible w.r.t. what the code generator needs. For (1), we chose x86-64 assembly because most personal computers run x86-64 programs. For (2), we chose to focus on a very narrow subset of x86-64 assembly.

Abstract syntax. In our formalisation, an assembly program asm is a list of instructions inst, defined below. In these type definitions, 64 word is a 64-bit word immediate value and 4 word is a 4-bit word address offset.

$$\begin{array}{rcl} \mathsf{asm} & = & \mathsf{inst\ list} \\ \mathsf{inst} & = & \mathsf{Const\ reg}\ (64\ \mathsf{word}) \mid \mathsf{Mov\ reg\ reg} \\ & \mid & \mathsf{Add\ reg\ reg} \mid \mathsf{Sub\ reg\ reg} \mid \mathsf{Div\ reg} \\ & \mid & \mathsf{Jump\ cond\ nat} \mid \mathsf{Call\ nat} \mid \mathsf{Ret} \\ & \mid & \mathsf{Pop\ reg} \mid \mathsf{Push\ reg} \\ & \mid & \mathsf{Add\_RSP\ nat} \mid \mathsf{Load\_RSP\ reg\ nat} \\ & \mid & \mathsf{Load\ reg\ reg}\ (4\ \mathsf{word}) \\ & \mid & \mathsf{Store\ reg\ reg}\ (4\ \mathsf{word}) \\ & \mid & \mathsf{GetChar} \mid \mathsf{PutChar} \mid \mathsf{Exit} \\ & \mid & \mathsf{Comment\ string} \\ \mathsf{reg} & = & \mathsf{RAX} \mid \mathsf{RDI} \mid \mathsf{RBX} \mid \mathsf{RBP} \mid \mathsf{RDX} \\ & \mid & \mathsf{R12} \mid \mathsf{R13} \mid \mathsf{R14} \mid \mathsf{R15} \\ \mathsf{cond} & = & \mathsf{Always} \mid \mathsf{Less\ reg\ reg} \mid \mathsf{Equal\ reg\ reg} \end{array}$$

The inst type covers only a tiny subset of the instructions, some of the registers and a few of the jump conditions that are available on x86-64. The GetChar, PutChar and Exit instructions expand to external calls. Here Comment expands to a comment `/* ... */` in the generated assembly; it aids readability and has no semantics, more specifically: execution gets stuck at the Comment instruction.

The stack pointer, i.e. register RSP, is not included in the reg type, because the stack is modelled abstractly in our formalisation of the semantics, which is explained next.

Semantics. Our semantics for x86-64 assembly models the state of the x86-64 machine using the following record type. Registers either map to a Some-value or None. We use None to model the case when a value is unknown due to a call to an external function. The stack is modelled as a mathematical list where each element is either a 64-bit word or a return address. The program counter is a natural number and the code of the assembly program is a list of instructions in the instructions field. Memory will be explained further down and I/O follows the approach used for the source semantics.

$$\begin{array}{l} \mathsf{state} = \langle\!| \\ \quad \mathsf{instructions} : \mathsf{asm}; \\ \quad \mathsf{pc} : \mathsf{nat}; \\ \quad \mathsf{regs} : \mathsf{reg} \to 64\ \mathsf{word\ option}; \\ \quad \mathsf{stack} : \mathsf{word\_or\_ret\ list}; \\ \quad \mathsf{memory} : 64\ \mathsf{word} \to 64\ \mathsf{word\ option\ option}; \\ \quad \mathsf{input} : \mathsf{char\ llist}; \\ \quad \mathsf{output} : \mathsf{string} \\ |\!\rangle \end{array}$$

$$\alpha\ \mathsf{option} = \mathsf{None} \mid \mathsf{Some}\ \alpha$$

$$\mathsf{word\_or\_ret} = \mathsf{Word}\ (64\ \mathsf{word}) \mid \mathsf{RetAddr\ nat}$$

The semantics of instruction fetching is to lookup the index of the program counter in the list of instructions.

$$\mathsf{fetch}\ s \stackrel{\mathrm{def}}{=} \mathsf{lookup}\ s.\mathsf{pc}\ s.\mathsf{instructions}$$

$$\begin{array}{l} \mathsf{lookup}\ n\ [] \stackrel{\mathrm{def}}{=} \mathsf{None} \\ \mathsf{lookup}\ n\ (x{::}xs) \stackrel{\mathrm{def}}{=} \\ \quad \mathsf{if}\ n = 0\ \mathsf{then\ Some}\ x\ \mathsf{else\ lookup}\ (n-1)\ xs \end{array}$$

The semantics of each instruction is given by a single-step relation, step, which has the following type in HOL:[2]

$$\mathsf{step} : \mathsf{s\_or\_h} \to \mathsf{s\_or\_h} \to \mathsf{bool}$$

where values of the s_or_h type are either a state or a Halt value indicating termination. Our semantics only allows termination by calls to the C function `exit`. Here Halt carries a 64-bit exit code and the string holding the output that was produced during program execution.

$$\mathsf{s\_or\_h} = \mathsf{State\ state} \mid \mathsf{Halt}\ (64\ \mathsf{word})\ \mathsf{string}$$

[2]In Coq, the type of step would be: s_or_h → s_or_h → Prop

The semantics of the Const instruction is defined by the following rule. Here write_reg updates the value of a register in the state and inc adds one to the pc in the state.

$$\frac{\mathsf{fetch}\ s = \mathsf{Some}\ (\mathsf{Const}\ r\ w)}{\mathsf{step}\ (\mathsf{State}\ s)\ (\mathsf{State}\ (\mathsf{write\_reg}\ r\ w\ (\mathsf{inc}\ s)))}$$

The rules for the other register operations are similar in style. For example, the rule giving semantics to Add is:

$$\frac{\begin{array}{c}\mathsf{fetch}\ s = \mathsf{Some}\ (\mathsf{Add}\ r_1\ r_2) \\ s.\mathsf{regs}\ r_1 = \mathsf{Some}\ w_1 \\ s.\mathsf{regs}\ r_2 = \mathsf{Some}\ w_2\end{array}}{\mathsf{step}\ (\mathsf{State}\ s)\ (\mathsf{State}\ (\mathsf{write\_reg}\ r_1\ (w_1 + w_2)\ (\mathsf{inc}\ s)))}$$

The semantics of Call and Ret, respectively, push and pop the return value to and from the stack.

$$\frac{\mathsf{fetch}\ s = \mathsf{Some}\ (\mathsf{Call}\ n)}{\begin{array}{l}\mathsf{step}\ (\mathsf{State}\ s) \\ \quad (\mathsf{State}\ (\mathsf{set\_pc}\ n \\ \qquad (\mathsf{set\_stack}\ (\mathsf{RetAddr}\ (s.\mathsf{pc} + 1){::}s.\mathsf{stack})\ s)))\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{fetch}\ s = \mathsf{Some\ Ret} \\ s.\mathsf{stack} = \mathsf{RetAddr}\ n{::}rest\end{array}}{\mathsf{step}\ (\mathsf{State}\ s)\ (\mathsf{State}\ (\mathsf{set\_pc}\ n\ (\mathsf{set\_stack}\ rest\ s)))}$$

Our last example is the Exit instruction. It illustrates how we formalise the 16-byte stack alignment requirement of the x86-64 calling convention: we require that the stack has even length. The Exit instruction terminates the assembly program with an $exit\_code$ that is passed in the RDI register.

$$\frac{\begin{array}{c}\mathsf{fetch}\ s = \mathsf{Some\ Exit} \\ s.\mathsf{regs\ RDI} = \mathsf{Some}\ exit\_code \\ \mathsf{even}\ (\mathsf{length}\ s.\mathsf{stack})\end{array}}{\mathsf{step}\ (\mathsf{State}\ s)\ (\mathsf{Halt}\ exit\_code\ s.\mathsf{output})}$$

The semantics of an entire execution is described by the reflexive-transitive closure, step*, of the step relation. The semantics of terminating assembly programs is the following. Here we require that there is some initial state t which satisfies init_state_ok. The last line below requires Halt 0 to be reachable from the initial state.

$$\begin{array}{l} (input, asm) \Downarrow_{\mathsf{asm}}\ output \stackrel{\mathrm{def}}{=} \\ \quad \exists\, t. \\ \qquad \mathsf{init\_state\_ok}\ t\ input\ asm\ \wedge \\ \qquad \mathsf{step}^*\ (\mathsf{State}\ t)\ (\mathsf{Halt}\ 0\ output) \end{array}$$

We define init_state_ok as follows. We will explain our requirement on memory, memory_writable, further below.

$$\begin{array}{l} \mathsf{init\_state\_ok}\ t\ input\ asm \stackrel{\mathrm{def}}{=} \\ \quad \exists\, r_{14}\ r_{15}. \\ \qquad t.\mathsf{pc} = 0 \wedge t.\mathsf{instructions} = asm \wedge t.\mathsf{input} = input\ \wedge \\ \qquad t.\mathsf{output} = \text{“”} \wedge t.\mathsf{stack} = [] \wedge t.\mathsf{regs\ R14} = \mathsf{Some}\ r_{14}\ \wedge \\ \qquad t.\mathsf{regs\ R15} = \mathsf{Some}\ r_{15}\ \wedge \\ \qquad \mathsf{memory\_writable}\ r_{14}\ r_{15}\ t.\mathsf{memory} \end{array}$$

Note that we use this $\Downarrow_{\mathsf{asm}}$ as an assumption in our compiler correctness theorems. As a result, the existential inside $\Downarrow_{\mathsf{asm}}$ can be read as a universal quantifier in these theorems:

$$\begin{array}{l} \vdash ((\mathit{input},\mathit{asm}) \Downarrow_{\mathsf{asm}} \mathit{output} \;\Rightarrow\; \mathit{prop}) \\ \iff \\ \forall\, t.\ \mathsf{init\_state\_ok}\ t\ \mathit{input}\ \mathit{asm} \land \\ \quad \mathsf{step}^*\ (\mathsf{State}\ t)\ (\mathsf{Halt}\ 0\ \mathit{output}) \;\Rightarrow\; \mathit{prop} \end{array}$$

Memory model. Finally, we will describe our memory model and, in particular, how we made it unusually restrictive in order to slightly simplify[3] the proofs about the code generator, which is the topic of the next section.

The memory field of the state record is the following:

$$\mathsf{memory} \;:\; 64\ \mathsf{word} \;\rightarrow\; 64\ \mathsf{word\ option\ option}$$

The memory is word-addressed and each memory location contains one of: None for not available; Some None for available but not yet initialised; and Some (Some w) for this address contains word w. Our semantics gets stuck (i.e., rejects) assignments to memory locations that are not Some None. In other words, once something has been stored to memory, it will be there forever. This property of our x86-64 semantics saves us some effort in the proof for codegen because we do not need to worry about an intermediate computation changing what has previously been stored to memory.

Our assembly semantics assumes that it starts from a state where every 8-byte-aligned memory location between the address held in R14 and the address held in R15 is writable, i.e. is Some None. We require that R14 and R15 are 16-byte aligned, i.e. their four least significant bits are 0.

$$\begin{array}{l} \mathsf{memory\_writable}\ r_{14}\ r_{15}\ m \stackrel{\text{def}}{=} \\ \ r_{14} \leq_{+} r_{15} \land \mathsf{aligned16}\ r_{14} \land \mathsf{aligned16}\ r_{15} \land r_{14} \neq 0 \land \\ \ \forall\, a.\ r_{14} \leq_{+} a <_{+} r_{15} \land \mathsf{aligned8}\ a \Rightarrow m\ a = \mathsf{Some\ None} \end{array}$$

Here $\leq_{+}$ and $<_{+}$ are unsigned comparisons for words.

In the definition above, we forbid the zero word, i.e. 0, from being in the writable part of memory. This restriction lets us determine that no pointer to a representation of Pair is equal to the empty list, i.e. Num 0 which we represent as word 0. Our source semantics allows Pair values to be compared with Num 0, and this restriction on the zero address is required for the verification of compilation of comparison (Equal).

6 Verification of the Code Generator

This section outlines how the code generator, codegen, was defined and how the following key theorem was proved.

$$\begin{array}{l} \vdash (\mathit{input},p) \Downarrow_{\mathsf{prog}} \mathit{output}_1 \land \\ \ (\mathit{input},\mathsf{codegen}\ p) \Downarrow_{\mathsf{asm}} \mathit{output}_2 \Rightarrow \\ \quad \mathit{output}_1 = \mathit{output}_2 \end{array}$$

This section omits definitions that do not fit here.

[3] If we were to drop this restriction, then we would need to establish a separation (e.g. using separation logic's separating conjunction) between unused memory and memory already used for Pair representations.

Design. We attempted to keep the code generator as simple as possible. In particular, this lead to decisions such as:

- to not implement or use any form of garbage collector,
- to not worry about generating verbose code, and
- to use the x86-64 machine as a stack machine.

However, there were also parts for which we felt that a bit of complexity in the code generator had to be tolerated:

- We ensure the code generator handles tail-calls properly, i.e. it generates a Jump instruction instead of a Call instruction for every function call in tail position.
- The assembly semantics forces us to ensure that the stack is of even length at the points where external functions GetChar, PutChar and Exit are called.
- To avoid terrible performance, we wrote the code generator in terms of a type (app_list shown below) that allows us to avoid suboptimal nesting of list append.

Implementation. The definition of the entire codegen function does not fit here. However, we will illustrate the style of definition by showing how code for Add is generated.

As mentioned above, we use an append-friendly type:

$$\begin{array}{l} \alpha\ \mathsf{app\_list}\ = \\ \quad \mathsf{List}\ (\alpha\ \mathsf{list}) \\ \ \mid \mathsf{Append}\ (\alpha\ \mathsf{app\_list})\ (\alpha\ \mathsf{app\_list}) \end{array}$$

We collapse values of this type into normal lists using the flatten function below. This function ensures that all list appends (++) are evaluated as if they were right-associated.

$$\begin{array}{l} \mathsf{flatten}\ (\mathsf{List}\ xs)\ acc \stackrel{\text{def}}{=} xs \mathbin{+\!\!+} acc \\ \mathsf{flatten}\ (\mathsf{Append}\ l_1\ l_2)\ acc \stackrel{\text{def}}{=} \mathsf{flatten}\ l_1\ (\mathsf{flatten}\ l_2\ acc) \end{array}$$

The c_defun function, shown below, generates code for declarations. The generated code consists of a function preamble, generated by c_pushes, followed by the code for the body of the function, generated by c_exp.

Most of our code generator functions take an assembly location l as input and produces one as output. On input, it is the location where the generated code will be. On output, this location is where the next generated instruction will be. Here fs is a mapping from source-level function names to corresponding assembly level locations (as will be clear from the definition of code_rel in Fig. 2).

$$\begin{array}{l} \mathsf{c\_defun}\ l\ \mathit{fs}\ (\mathsf{Defun}\ n\ vs\ \mathit{body}) \stackrel{\text{def}}{=} \\ \ \mathsf{let}\ (c_0,vs,l_0) = \mathsf{c\_pushes}\ vs\ l\ \mathsf{in} \\ \ \mathsf{let}\ (c_1,l_1) = \mathsf{c\_exp}\ \mathsf{T}\ l_0\ vs\ \mathit{fs}\ \mathit{body}\ \mathsf{in} \\ \quad (\mathsf{Append}\ c_0\ c_1,l_1) \end{array}$$

The function c_defun calls c_exp with T as the first argument. This indicates that the expression is to be compiled in tail-position. For the Op case, c_exp T calls the non-tail version, i.e. c_exp F. Below c_exps is a list version of c_exp that is defined in mutual recursion with c_exp.

$$\begin{array}{l}\mathsf{c\_exp}\ \mathsf{T}\ l\ vs\ fs\ (\mathsf{Op}\ op\ xs) \stackrel{\text{def}}{=}\\ \quad \mathsf{make\_ret}\ vs\ (\mathsf{c\_exp}\ \mathsf{F}\ l\ vs\ fs\ (\mathsf{Op}\ op\ xs))\end{array}$$

$$\begin{array}{l}\mathsf{c\_exp}\ \mathsf{F}\ l\ vs\ fs\ (\mathsf{Op}\ op\ xs) \stackrel{\text{def}}{=}\\ \quad \mathsf{let}\ (c,l') = \mathsf{c\_exps}\ l\ vs\ fs\ xs\ \mathsf{in}\\ \quad \mathsf{let}\ insts = \mathsf{c\_op}\ op\ vs\ l'\ \mathsf{in}\\ \quad\quad (\mathsf{Append}\ c\ (\mathsf{List}\ insts),l' + \mathsf{length}\ insts)\end{array}$$

If the primitive operation op is Add then the definitions lead us to the following code.

$$\mathsf{c\_op}\ \mathsf{Add}\ vs\ l \stackrel{\text{def}}{=} \mathsf{c\_add}\ vs$$

$$\begin{array}{l}\mathsf{c\_add}\ vs \stackrel{\text{def}}{=}\\ \quad [\mathsf{Pop}\ \mathsf{RDI};\\ \quad\ \ \mathsf{Add}\ \mathsf{RAX}\ \mathsf{RDI};\\ \quad\ \ \mathsf{Jump}\ (\mathsf{Less}\ \mathsf{R13}\ \mathsf{RAX})\ (\mathsf{give\_up}\ (\mathsf{even\_len}\ vs))]\end{array}$$

$$\mathsf{give\_up}\ b \stackrel{\text{def}}{=} \mathsf{if}\ b\ \mathsf{then}\ 14\ \mathsf{else}\ 15$$

We can see that Add is implemented by three instructions: a stack pop, an addition and a conditional jump. The conditional jump checks whether the result of the addition is greater than the content of register R13, which is accroding to our invariant (see state_rel in Fig. 2) always contains the largest number (i.e. $2^{63}-1$) that our generated code allows. If the result of the addition exceeds this maximum value, then the code jumps to either code location 14 or 15, depending on the length of the list vs. The code at those locations call Exit with exit code 1. The destination of the jump is adjusted so that the stack has even length when Exit is called.

The implementation of Add must resort to an early exit in some cases, because the assembly language can only hold 64-bit values in its registers, but the source semantics allows for arbitrarily large natural numbers. As a result, the generated code sometimes has to give up. Our implementation allows numbers up to 63 bits in size so that the final bit can be used to check overflow.

Cons is the other primitive that can resort to an early exit. It exits with exit code 1 when heap space is exhausted.

Verification. Proving the correctness of the codegen function requires showing that a simulation relation holds between the evaluation of the source program and execution of the generated assembly program. More specifically, both executions must agree on the externally observable events, namely, output and termination/non-termination.

Our source and target languages are deterministic and, as a result, it suffices to show a *forward simulation* theorem even if other results are the final goal. A theorem stated as a forward simulation has the shape: for any source evaluation, the corresponding target execution is similar enough.

Figure 1 shows a forward simulation result that we have proved for the function that compiles expressions, i.e. c_exp.

```
1   ⊢ (env,[x],s) ⇓exp ([v],s1) ∧
2     c_exp is_tail t.pc vs fs x = (code,l1) ∧
3     state_rel fs s t ∧ env_ok env vs curr t ∧
4     has_stack t (curr ++ rest) ∧ odd (length rest) ∧
5     code_in t.pc (flatten code []) t.instructions ⇒
6      ∃ outcome.
7         step* (State t) outcome ∧
8         case outcome of
9           State t1  ⇒
10            state_rel fs s1 t1 ∧
11            ∃ w.
12              v_inv t1 v w ∧
13              if is_tail then
14                has_stack t1 (Word w::rest) ∧
15                fetch t1 = Some Ret
16              else
17                has_stack t1 (Word w::curr ++ rest) ∧
18                t1.pc = l1
19        | Halt ec output  ⇒
20          output ≼ s1.output ∧ ec = 1
```

Figure 1. Correctness of c_exp

The theorem statement is long and requires some explanation. The rest of this section explains this key lemma and the definitions that it relies on.

We focus on this lemma about c_exp because it touches on all noteworthy aspects of the verification of the code generator. Proving the prog-level theorem is an exercise in instantiating this lemma for the main expression of the source program.

When looking at Figure 1, one should note that this is a forward simulation because, on line 1, we *assume that the source semantics has evaluated* to some result, and in the conclusion we see, on line 7, that the assumptions above *imply the existence of an execution in the assembly language.*

How does the assembly execution relate to the source execution? The answer is on line 2 and line 5. Line 2 states that we assume that the compiler function, in this case c_exp, has produced $code$. Line 5 assumes (using code_in below) that the program counter points to the beginning of where $code$ is installed in the list of t.instructions. Thus the compiler output dictates what will happen in the assembly execution.

$$\mathsf{code\_in}\ n\ []\ insts \stackrel{\text{def}}{=} \mathsf{T}$$

$$\begin{array}{l}\mathsf{code\_in}\ n\ (x{::}xs)\ insts \stackrel{\text{def}}{=}\\ \quad \mathsf{lookup}\ n\ insts = \mathsf{Some}\ x \wedge \mathsf{code\_in}\ (n+1)\ xs\ insts\end{array}$$

Next we will look at how the source-level evaluation is mimicked at the assembly level. We treat the x86-64 machine as if it was a stack machine where the top of the stack is held in register RAX and the rest of the stack is in the stack field of the assembly state. We write has_stack t xs to say that state t represents such a stack xs.

$$\begin{array}{l}
\mathsf{has_stack}\ t\ xs \stackrel{\text{def}}{=} \\
\quad \exists\, w\ ws.\\
\quad\quad xs = \mathsf{Word}\ w{::}ws \land t.\mathsf{regs}\ \mathsf{RAX} = \mathsf{Some}\ w \land \\
\quad\quad t.\mathsf{stack} = ws
\end{array}$$

In Figure 1, line 4 assumes that a stack is present that can be divided into a current stack frame, $curr$, and the rest of the stack, $rest$. On line 14, we see that the is_tail case requires execution to finish in a state where the $curr$ has been dropped and only the return value w is left in front of $rest$. In the non-tail case, on line 17, we see that the return value w has been pushed onto the stack, leaving $curr$ ++ $rest$ untouched underneath.

Note that lines 6-8 and 19-20 of Figure 1 always allow the assembly level execution to resort to Halt with exit code 1. In such cases, we care only that the assembly level output is a prefix $\preccurlyeq$ of the source level output.

In case a Halt is avoided (line 9), then there exists some assembly-level result word w (line 11) such that it is v_inv-related to the source-level result value v (line 12). The definition of v_inv is shown below.

The v_inv $t\ v\ w$ relation defines how we represent source-level value v at the assembly level by a word w w.r.t. an assembly state t. A numeric value Num n is represented by a word w if n is smaller than 2^{63} and the word is equal to the n converted to a machine word, which we write as n2w n.

$$\mathsf{v\_inv}\ t\ (\mathsf{Num}\ n)\ w \stackrel{\text{def}}{=} n < 2^{63} \land w = \mathsf{n2w}\ n$$

A Pair $x_1\ x_2$ is represented by a word w if that word is a pointer to the word w_1 in memory that represents x_1, and similarly $w + 8$ is pointer to a word w_2 in memory that represents x_2. (The offset is 8 since there are 8 bytes in a 64-bit word. Memory is byte addressed on x86-64.)

$$\begin{array}{l}
\mathsf{v_inv}\ t\ (\mathsf{Pair}\ x_1\ x_2)\ w \stackrel{\text{def}}{=} \\
\quad \exists\, w_1\ w_2.\\
\quad\quad \mathsf{read_mem}\ (w + 0)\ t = \mathsf{Some}\ w_1 \land \mathsf{v_inv}\ t\ x_1\ w_1 \land \\
\quad\quad \mathsf{read_mem}\ (w + 8)\ t = \mathsf{Some}\ w_2 \land \mathsf{v_inv}\ t\ x_2\ w_2 \land \\
\quad\quad w \neq 0
\end{array}$$

The v_inv relation is used in definition of env_ok, which relates the source semantics environment env, the current stack frame $curr$, and the compiler's model of the current stack frame vs. We define env_ok to say that the model of the stack frame vs must be exactly the same length as the current stack frame $curr$. Furthermore, for any variable binding that exists in the source-level environment env, it must be possible to look up (using find) a position for this variable in the model of the stack frame vs, and a load from that position in the current stack frame must result in a v_inv-related word.

$$\begin{array}{l}
\mathsf{state_rel}\ fs\ s\ t \stackrel{\text{def}}{=} \\
\quad s.\mathsf{input} = t.\mathsf{input} \land s.\mathsf{output} = t.\mathsf{output} \land \\
\quad \mathsf{code_rel}\ fs\ s.\mathsf{funs}\ t.\mathsf{instructions} \land \\
\quad \exists\, r_{14}\ r_{15}.\\
\quad\quad t.\mathsf{regs}\ \mathsf{R12} = \mathsf{Some}\ 16 \land t.\mathsf{regs}\ \mathsf{R13} = \mathsf{Some}\ (2^{63} - 1) \land \\
\quad\quad t.\mathsf{regs}\ \mathsf{R14} = \mathsf{Some}\ r_{14} \land t.\mathsf{regs}\ \mathsf{R15} = \mathsf{Some}\ r_{15} \land \\
\quad\quad \mathsf{memory_writable}\ r_{14}\ r_{15}\ t.\mathsf{memory}
\end{array}$$

$$\begin{array}{l}
\mathsf{code_rel}\ fs\ funs\ instructions \stackrel{\text{def}}{=} \\
\quad \mathsf{init_code_in}\ instructions \land \\
\quad \forall\, n\ params\ body.\\
\quad\quad \mathsf{lookup_fun}\ n\ funs = \mathsf{Some}\ (params, body) \Rightarrow \\
\quad\quad\quad \exists\, pos.\\
\quad\quad\quad\quad \mathsf{lookup}\ fs\ n = \mathsf{Some}\ pos \land \\
\quad\quad\quad\quad \mathsf{code_in}\ pos \\
\quad\quad\quad\quad\quad (\mathsf{flatten} \\
\quad\quad\quad\quad\quad\quad (\mathsf{fst}\ (\mathsf{c_defun}\ pos\ fs\ (\mathsf{Defun}\ n\ params\ body)))\ [\,]) \\
\quad\quad\quad\quad\quad instructions
\end{array}$$

$$\begin{array}{l}
\mathsf{init_code_in}\ instructions \stackrel{\text{def}}{=} \\
\quad \exists\, start.\ \mathsf{code_in}\ 0\ (\mathsf{init}\ start)\ instructions
\end{array}$$

$$\begin{array}{l}
\mathsf{init}\ start \stackrel{\text{def}}{=} \\
\quad [\mathsf{Const\ RAX}\ 0;\ \mathsf{Const\ R12}\ 16;\ \mathsf{Const\ R13}\ (2^{63} - 1);\\
\quad\ \ \mathsf{Call}\ start;\ \mathsf{Const\ RDI}\ 0;\ \mathsf{Exit};\ \mathsf{Comment}\ \text{“cons”};\\
\quad\ \ \mathsf{Jump}\ (\mathsf{Equal\ R14\ R15})\ 14;\ \mathsf{Store\ RDI\ R14}\ 0;\\
\quad\ \ \mathsf{Store\ RAX\ R14}\ 8;\ \mathsf{Mov\ RAX\ R14};\ \mathsf{Add\ R14\ R12};\ \mathsf{Ret};\\
\quad\ \ \mathsf{Comment}\ \text{“exit 1”};\ \mathsf{Push\ R15};\ \mathsf{Const\ RDI}\ 1;\ \mathsf{Exit}]
\end{array}$$

Figure 2. The definition of the state relation state_rel and its components: the code relation code_rel, and a predicate init_code_in about existence of the initial code, init.

$$\begin{array}{l}
\mathsf{env_ok}\ env\ vs\ curr\ t \stackrel{\text{def}}{=} \\
\quad \mathsf{length}\ vs = \mathsf{length}\ curr \land \\
\quad \forall\, n\ v.\\
\quad\quad env\ n = \mathsf{Some}\ v \Rightarrow \\
\quad\quad\quad \mathsf{find}\ n\ vs\ 0 < \mathsf{length}\ curr \land \\
\quad\quad\quad \exists\, w.\ \mathsf{el}\ (\mathsf{find}\ n\ vs\ 0)\ curr = \mathsf{Word}\ w \land \mathsf{v_inv}\ t\ v\ w
\end{array}$$

$$\begin{array}{l}
\mathsf{find}\ n\ [\,]\ k \stackrel{\text{def}}{=} k \\
\mathsf{find}\ n\ (\mathsf{None}{::}vs)\ k \stackrel{\text{def}}{=} \mathsf{find}\ n\ vs\ (k + 1) \\
\mathsf{find}\ n\ (\mathsf{Some}\ v{::}vs)\ k \stackrel{\text{def}}{=} \\
\quad \mathsf{if}\ v = n\ \mathsf{then}\ k\ \mathsf{else}\ \mathsf{find}\ n\ vs\ (k + 1)
\end{array}$$

Lines 3 and 10 of Figure 1 are still to be explained. Line 3 requires that state_rel holds between the initial source state s and the initial assembly state t, and that these are consistent with fs. Here fs is a mapping from function names to locations in the generated assembly code.

The definition of state_rel is shown in Figure 2. It requires that the source and assembly states agree on the content of the input and output fields; it requires that the compilation of each source function is present, at the right locations according to fs, in the assembly code; it requires that registers R12–R15 have specific values; finally, it requires memory_writable, see end of Section 5.

The definition of code_rel, shown in Figure 2, requires that the initial code produced by init is present in memory. This initial code has three parts: (1) the first few instructions initialise registers RAX, R12 and R13 at the start of execution, (2) it contains a helper routine (following Comment "cons") for memory allocation, and (3) it has a helper routine (following Comment "exit 1") for aborting execution with Exit applied to number 1, i.e. the exit code for failure.

The theorem shown in Figure 1 was proved by induction on the semantics of expression evaluation. Our proof, which was not particularly difficult once the correct theorem statement was found, involves careful expansion of the step* relation for each snippet of generated assembly code.

In our formal development, we actually proved a slightly more general statement than the one shown in Figure 1. The reason is that we also wanted to prove divergence preservation for the code generator. The next section explains how the statement was generalised.

7 Extra: Proof of Divergence Preservation

This section explains how we have proved that the code generator preserves behaviour also for non-terminating, i.e. diverging, programs. Proving divergence preservation is not required for in-logic bootstrapping of a compiler, but we include it here because we consider divergence preservation to be an important part of compiler verification in general.

Theorem statement. Divergence preservation means that the generated code agrees with the source semantics also when the program diverges, i.e. when the program runs forever. As pointed out earlier, we allow the generated assembly to exit early due to running out of memory or some number becoming too large to represent in a register. As a result, we state our divergence preservation as an implication in only one direction: if the generated assembly program diverges ($\Uparrow_{\mathsf{asm}}$) and the source program is well-defined (i.e. does not crash), then the source program also diverges ($\Uparrow_{\mathsf{prog}}$). Furthermore, the non-terminating execution agrees on the potentially infinite stream of $output$ that is produced.

$$\begin{array}{l}\vdash \mathsf{prog\_avoids\_crash}\ input\ prog\ \wedge \\ \quad (input,\mathsf{codegen}\ prog) \Uparrow_{\mathsf{asm}} output \Rightarrow \\ \qquad (input,prog) \Uparrow_{\mathsf{prog}} output\end{array}$$

The new notation, i.e. $\Uparrow_{\mathsf{asm}}$ and $\Uparrow_{\mathsf{prog}}$, will be defined below.

Divergence semantics for assembly. We say that an assembly program runs forever from a state t if: for every k, one can take k transitions of step, written step^k, and still successfully arrive at a new state t', without hitting Halt.

$$\forall\, k.\ \exists\, t'.\ \mathsf{step}^k\ (\mathsf{State}\ t)\ (\mathsf{State}\ t')$$

We express the (potentially never ending) output stream produced as a least upper bound, LUB, of all output traces that all of the finite execution can produce.

$$\mathsf{LUB}\ \{\ t'.\mathsf{output} \mid \mathsf{step}^*\ (\mathsf{State}\ t)\ (\mathsf{State}\ t')\ \}$$

Here LUB has type string set → char llist, where llist is either a finite list or an infinite list. This least upper bound is well-defined here since step is deterministic and produces output in a monotonic way.

With these formulations, we define the semantics of a diverging assembly execution, $(input,asm) \Uparrow_{\mathsf{asm}} output$, to be true if there exists some initial state t with $input$ and asm installed such that execution from t will never stop and will produce output described by $output$.

$$\begin{array}{l}\vdash (input,asm) \Uparrow_{\mathsf{asm}} output \stackrel{\mathrm{def}}{=} \\ \quad \exists\, t. \\ \qquad \mathsf{init\_state\_ok}\ t\ input\ asm\ \wedge \\ \qquad (\forall\, k.\ \exists\, t'.\ \mathsf{step}^k\ (\mathsf{State}\ t)\ (\mathsf{State}\ t'))\ \wedge \\ \qquad output = \mathsf{LUB}\ \{\ t'.\mathsf{output} \mid \mathsf{step}^*\ (\mathsf{State}\ t)\ (\mathsf{State}\ t')\ \}\end{array}$$

Just like in Section 5, we note that the existential quantifier on the initial state t ought to be read as a universal quantification in the compiler correctness theorem since $\Uparrow_{\mathsf{asm}}$ appears on the left-hand side of an implication.

Divergence semantics for source. We use a functional big-step semantics [21] to define the semantics for diverging source programs, since the classical relational semantics of Section 3 cannot express non-termination.

Figure 3 shows the expression evaluating function of our functional big-step semantics. One can read the functional big-step semantics as an interpreter for the language. In this figure, we display it expressed in Haskell-inspired do-notation using a state-and-exception monad, where normal results are Res v, for some value v, and errors are Err e, where e is either Crash or TimeOut.

The TimeOut error has to do with the semantic clock that the functional big-step semantics style requires. In Section 3, we included a natural-number-valued clock field in the state in order to be able to define a function big-step semantics. Our functional big-step semantics decrements this clock at every Call: in our semantics, get_env_and_body decrements the clock by one. If the clock is zero on entry to get_env_and_body, then this function return Err TimeOut. This timeout error gets propagated to the top level.

We use these timeouts to define non-termination. First we define eval_from k $input$ p to be the evaluation of a *whole program* from a given $input$ and initial clock k.

$$\begin{array}{l}\mathsf{eval\_from}\ k\ input\ (\mathsf{Program}\ funs\ main) \stackrel{\mathrm{def}}{=} \\ \quad \mathsf{eval}\ \mathsf{empty\_env}\ main \\ \qquad (\mathsf{init\_state}\ input\ funs\ \mathsf{with}\ \mathsf{clock}\ :=\ k)\end{array}$$

$$
\begin{array}{l}
\mathsf{eval}\ env\ (\mathsf{Const}\ n) \stackrel{\text{def}}{=} \mathsf{return}\ (\mathsf{Num}\ n) \\
\mathsf{eval}\ env\ (\mathsf{Var}\ n) \stackrel{\text{def}}{=} \\
\quad \mathsf{case}\ env\ n\ \mathsf{of}\ \mathsf{None} \Rightarrow \mathsf{fail} \mid \mathsf{Some}\ v \Rightarrow \mathsf{return}\ v \\
\mathsf{eval}\ env\ (\mathsf{Op}\ f\ xs) \stackrel{\text{def}}{=} \\
\quad \mathsf{do}\ vs \leftarrow \mathsf{evals}\ env\ xs;\ \mathsf{eval\_op}\ f\ vs\ \mathsf{od} \\
\mathsf{eval}\ env\ (\mathsf{Let}\ vname\ x\ y) \stackrel{\text{def}}{=} \\
\quad \mathsf{do}\ v \leftarrow \mathsf{eval}\ env\ x; \\
\qquad \mathsf{eval}\ env\langle vname \mapsto \mathsf{Some}\ v\rangle\ y\ \ \mathsf{od} \\
\mathsf{eval}\ env\ (\mathsf{If}\ test\ xs\ y\ z) \stackrel{\text{def}}{=} \\
\quad \mathsf{do}\ vs \leftarrow \mathsf{evals}\ env\ xs; \\
\qquad b \leftarrow \mathsf{take\_branch}\ test\ vs; \\
\qquad \mathsf{eval}\ env\ (\mathsf{if}\ b\ \mathsf{then}\ y\ \mathsf{else}\ z)\ \ \mathsf{od} \\
\mathsf{eval}\ env\ (\mathsf{Call}\ fname\ xs) \stackrel{\text{def}}{=} \\
\quad \mathsf{do}\ vs \leftarrow \mathsf{evals}\ env\ xs; \\
\qquad (fenv,body) \leftarrow \mathsf{get\_env\_and\_body}\ fname\ vs; \\
\qquad \mathsf{eval}\ fenv\ body\ \ \mathsf{od} \\
\mathsf{evals}\ env\ [] \stackrel{\text{def}}{=} \mathsf{return}\ [] \\
\mathsf{evals}\ env\ (x{::}xs) \stackrel{\text{def}}{=} \\
\quad \mathsf{do}\ v \leftarrow \mathsf{eval}\ env\ x; \\
\qquad vs \leftarrow \mathsf{evals}\ env\ xs; \\
\qquad \mathsf{return}\ (v{::}vs)\ \ \mathsf{od}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{return}\ v\ s \stackrel{\text{def}}{=} (\mathsf{Res}\ v,s) \\
\mathsf{fail}\ s \stackrel{\text{def}}{=} (\mathsf{Err}\ \mathsf{Crash},s) \\
\mathsf{monad\_bind}\ f\ g\ s \stackrel{\text{def}}{=} \\
\quad \mathsf{case}\ f\ s\ \mathsf{of}\ (\mathsf{Res}\ v,s_1) \Rightarrow g\ v\ s_1 \mid (\mathsf{Err}\ e,s_1) \Rightarrow (\mathsf{Err}\ e,s_1)
\end{array}
$$

Figure 3. A functional big-step semantics for source expressions. This is used for our proof of divergence preservation.

Using eval_from, we define a function that checks for timeout and one that returns the output for a given initial clock.

$$
\begin{array}{l}
\mathsf{prog\_timesout}\ k\ input\ prog \stackrel{\text{def}}{=} \\
\quad \exists\, s.\ \mathsf{eval\_from}\ k\ input\ prog = (\mathsf{Err}\ \mathsf{TimeOut},s)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{prog\_output}\ k\ input\ prog \stackrel{\text{def}}{=} \\
\quad \mathsf{let}\ (res,s) = \mathsf{eval\_from}\ k\ input\ prog\ \mathsf{in}\ s.\mathsf{output}
\end{array}
$$

A source program diverges if the program times out for every initial clock value k. The output is captured by the least upper bound (LUB) of all partial outputs.

$$
\begin{array}{l}
(input,prog) \Uparrow_{\mathsf{prog}} output \stackrel{\text{def}}{=} \\
\quad (\forall\, k.\ \mathsf{prog\_timesout}\ k\ input\ prog)\ \wedge \\
\quad output = \mathsf{LUB}\ \{\ \mathsf{prog\_output}\ k\ input\ prog \mid k \in \mathbb{N}\ \}
\end{array}
$$

Modifications to the verification of the code generator. Only a few minor adjustments need to be made to code generator proofs to also make them support a final theorem about divergence preservation. In the theorem shown in Figure 1, we swap lines 1, 7 and 11 to the following.

$$
\begin{array}{rl}
1 & \mathsf{eval}\ env\ e\ x = (res,s_1) \wedge res \neq \mathsf{Err}\ \mathsf{Crash}\ \wedge \\
\ldots & \\
7 & \quad \mathsf{steps}\ (\mathsf{State}\ t, s.\mathsf{clock})\ (outcome, s_1.\mathsf{clock})\ \wedge \\
\ldots & \\
11 & \qquad \forall\, v.\ res = \mathsf{Res}\ v \Rightarrow \exists\, w.
\end{array}
$$

Here $\mathsf{steps}\ (t_1,n_1)\ (t_2,n_2) \stackrel{\text{def}}{=} \exists\, n.\ \mathsf{step}^n\ t_1\ t_2 \wedge n_1 \le n{+}n_2$.

We use steps to ensure that sufficiently many execution steps are taken by the assembly program. Note that when eval returns res = Err TimeOut, then s_1.clock is always 0. In that case, line 7 states that the assembly program has taken at least s.clock steps, i.e. as many steps as the number of times Call was evaluated as part of eval on line 1. This loose relationship between the number of execution steps between source and assembly is sufficient for our proof.

As part of the top-level divergence preservation proof, we encounter the following proof goal:

$$
\begin{array}{l}
\mathsf{init\_state\_ok}\ t\ input\ (\mathsf{codegen}\ prog) \wedge \ldots \Rightarrow \\
\quad \mathsf{LUB}\ \{\ t'.\mathsf{output} \mid \mathsf{step}^*\ (\mathsf{State}\ t)\ (\mathsf{State}\ t')\ \} = \\
\quad \mathsf{LUB}\ \{\ \mathsf{prog\_output}\ k\ input\ prog \mid k \in \mathbb{N}\ \}
\end{array}
$$

We prove this goal with the help of the following lemma.

$$
\begin{array}{l}
\vdash \mathsf{total}\ s_1 \wedge \mathsf{total}\ s_2 \wedge s_1\ \mathsf{within}\ s_2 \wedge s_2\ \mathsf{within}\ s_1 \Rightarrow \\
\quad \mathsf{LUB}\ s_1 = \mathsf{LUB}\ s_2
\end{array}
$$

Here total requires that any two elements of the given set must be related by the prefix $\preccurlyeq$ relation; and within requires each element of the first set to be a prefix $\preccurlyeq$ of some element of the second set.

$$
\begin{array}{l}
\mathsf{total}\ s \stackrel{\text{def}}{=} \forall\, l_1\ l_2.\ l_1 \in s \wedge l_2 \in s \Rightarrow l_1 \preccurlyeq l_2 \vee l_2 \preccurlyeq l_1 \\
s_1\ \mathsf{within}\ s_2 \stackrel{\text{def}}{=} \forall\, l_1.\ l_1 \in s_1 \Rightarrow \exists\, l_2.\ l_2 \in s_2 \wedge l_1 \preccurlyeq l_2
\end{array}
$$

8 Bootstrapping Results and Proof Scripts

This section concludes our description of the bootstrapping work by recapping the top-level results, showing some of the generated artifacts and presenting some numbers.

Theorems. The two most important top-level theorems of this paper are the following. The first is a theorem which states that the compiler_asm assembly program correctly implements the abstract compiler function:

$$
\begin{array}{l}
\vdash (input,\mathsf{compiler\_asm}) \Downarrow_{\mathsf{asm}} output \Rightarrow \\
\quad output = \mathsf{compiler}\ input
\end{array}
$$

The second theorem is an evaluation of the application of the asm2str function to the compiler_asm assembly program (where compiler_asm $\stackrel{\text{def}}{=}$ codegen compiler_prog).

$$
\vdash \mathsf{asm2str}\ \mathsf{compiler\_asm} = \text{“}\ldots\text{”}
$$

Here "…" is a concrete string that can be printed into a text file. Figure 4 shows the initial and final part of that string.

```
        .bss
        .p2align 3          /* 8-byte align        */
heapS:
        .space 8*1024*1024  /* bytes of heap space */
        .p2align 3          /* 8-byte align        */
heapE:

        .text
        .globl main
main:
        subq $8, %rsp        /* 16-byte align %rsp */
        movabs $heapS, %r14  /* r14 := heap start  */
        movabs $heapE, %r15  /* r15 := heap end    */

L0:     movq $0, %rax
L1:     movq $16, %r12
L2:     movq $9223372036854775807, %r13
L3:     call L10015
L4:     movq $0, %rdi
...
        /* main */
L10015: pushq %rax
L10016: call L4440
L10017: call L6407
L10018: call L3906
L10019: call L8494
L10020: addq $8, %rsp
L10021: jmp L9993
```

Figure 4. The initial and final part of the 10 188-line string resulting from evaluating "asm2str compiler_asm" in logic.

Compiler in source syntax. We also evaluate prog2str applied to compiler_prog to get hold of a string representation of the concrete source syntax for compiler_prog.

$$\vdash \text{prog2str compiler\_prog coms} = \text{``\ldots''}$$

Parts of this string are shown in Figure 5.

Executable artefacts. We can run the compiler outside of the logic, as a command-line program, as follows. If we store the string representation of compiler_asm in a file called `compiler_asm.s`, and the string for compiler_prog in file `compiler_prog.txt`, then we can get an executable by calling GCC to assemble and link the assembly file. We can then run it as a normal program and use `time` to get some measurement of its runtime:

```
$ gcc -o compiler compiler_asm.s
$ time ./compiler < compiler_prog.txt > output

real    0m0.011s
user    0m0.006s
sys     0m0.005s
$ diff output compiler_asm.s
$
```

The `./compiler` command above applied compiler_asm to compiler_prog and stored the output in a file called `output`.

```
# This file is generated from the HOL4 theorem prover.
...
(defun append (v l)
   (if (= v '0) l (cons (head v) (append (tail v) l))))

(defun flatten (v acc)
   (case v ((List xs) (append xs acc))
      ((Append l1 l2) (flatten l1 (flatten l2 acc)))))

(defun even_len (v) (if (= v '0) '1 (odd_len (tail v))))

(defun odd_len (v) (if (= v '0) '0 (even_len (tail v))))

(defun give_up (b) (if (= b '1) '14 '15))
...
(defun c_add (vs)
   (list (Pop RDI) (Add RAX RDI)
       (Jump (Less R13 RAX) (give_up (even_len vs)))))
...
# The main expression

(print (asm2str (codegen (parser (lexer)))))
```

Figure 5. Parts of the 542-line string resulting from evaluating "prog2str compiler_prog coms" in logic. Lines starting with # are comments that are injected via coms.

We use `diff` to check that the `output` textfile is identical to `compiler_asm.s`. We note that `diff` found no differences.

Proof scripts. A summary of line counts is shown below. The files build in less than 5 minutes on an Intel Core i7.

group of HOL4 files	# lines
source syntax, semantics, lemmas	1154
x64 assembly syntax, semantics, lemmas	609
code generator and its proofs	2225
parsing, printing and their proofs	1415
compiler_prog, automation and proofs	2166
top-level compiler, evaluation and proofs	79
total	**7648**

9 Related Work

Chirica and Martin [4] seem to have been the first to consider the gap between compilation algorithm and executable compiler implementation, in the context of compiler verification.

Curzon [5] was the first to propose in-logic execution of the compiler algorithm as a potential way of producing a verified implementation of the compiler algorithm.

The VLISP project [9] was the first to apply compiler bootstrapping in the context of a complier verification project. However, their verification proofs were not mechanised in an ITP, instead they were "rigorous, but not completely formal, much in the style of ordinary mathematical discourse."

Dold et al. [7] describes the next milestone: a mechanically verified compiler that was bootstrapped outside of an ITP but thoroughly inspected. The verification was performed inside the PVS ITP [22]. Bootstrapping was done outside, but the result of the bootstrap underwent a rigorous a-posteriori syntactic code inspection which took 3 months and produced "approx. 1000 pages of code-inspection protocols."

Next, Leroy published his seminal papers on CompCert [15, 16]. CompCert has not (yet) been bootstrapped, but it has become a landmark in compiler verification. The CompCert project showed that realistic compilers can be mechnically verified, and this set off a flurry of activity that have explored, e.g., compositionality [20, 26], concurrency [11, 23, 24] and security [1–3] with regard to compiler verification.

Inspired by CompCert, the CakeML project [29] produced a realistic verified compiler for an ML-like language. The CakeML compiler is the first verified boostrapped compiler for which boostrapping was done inside an ITP. CakeML's compiler boostrapping works in much the same way as the method described in this paper. However, the CakeML boostrap theorems are much harder to understand [29, Sec. 11] due to the CakeML project's aim of realism. The pursuit of realism causes clutter to creep into the compiler correctness statements. Some of this clutter stems from, for example, CakeML's support for more general forms of I/O than the simple I/O considered here and the fact that CakeML is compiled to several different machine languages.

Bootstrapping is self-application. Self-application has also been considered for ITPs themselves [6, 10, 12, 27, 28].

10 Conclusions

This paper has described a small verified bootstrapped compiler development that we have tried to keep as free from unnecessary clutter as possible. We hope that this development makes the concept of bootstrapping clear and that it inspires more use of compiler bootstrapping in ITPs.

Bootstrapping a less minimal compiler? Producing a verified bootstrapped compiler consists of three major tasks: (T1) verifying a code generator, (T2) producing the verified deep embedding (here: compiler_prog), and (T3) evaluating the code generator on the deep embedding in the logic.

The effort involved in task T1 increases as the source language supported by the compiler becomes harder to compile. However, note that a source language that fits compiler implementation makes task T2 simpler. Any increase in the size of the compiler causes task T3 (and to a lesser extent T2) to become computationally heavier to run. For CakeML, running tasks T2 and T3 takes several hours, while T2 and T3 complete in minutes for the simple compiler of this paper.

Acknowledgments

Oskar Abrahamsson, Andreas Lööw, Michael Norrish and Yong Kiam Tan kindly provided feedback on drafts of this text. The anonymous reviewers also gave detailed and helpful comments. This work was supported by funding from the Swedish Foundation for Strategic Research.

References

[1] C. Abate, A. A. de Amorim, R. Blanco, A. N. Evans, G. Fachini, C. Hritcu, T. Laurent, B. C. Pierce, M. Stronati, and A. Tolmach. When good components go bad: Formally secure compilation despite dynamic compromise. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *Computer and Communications Security (CCS)*, pages 1351–1368. ACM, 2018. doi:10.1145/3243734.3243745.

[2] C. Abate, R. Blanco, D. Garg, C. Hritcu, M. Patrignani, and J. Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. In *32nd IEEE Computer Security Foundations Symposium (CSF)*, pages 256–271. IEEE, June 2019. doi:10.1109/CSF.2019.00025.

[3] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. Trieu. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.*, 4(POPL):7:1–7:30, 2020. doi:10.1145/3371075.

[4] L. M. Chirica and D. F. Martin. Toward compiler implementation correctness proofs. *ACM Trans. Program. Lang. Syst.*, 8(2):185–214, Apr. 1986. ISSN 0164-0925. doi:10.1145/5397.30847.

[5] P. Curzon. Deriving correctness properties of compiled code. In L. J. M. Claesen and M. J. C. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications, Proceedings of the IFIP TC10/WG10.2 Workshop HOL'92, Leuven, Belgium, 21-24 September 1992*, volume A-20 of *IFIP Transactions*, pages 327–346. North-Holland/Elsevier, 1992. doi:10.1016/B978-0-444-89880-7.50027-9.

[6] J. Davis and M. O. Myreen. The reflective Milawa theorem prover is sound (down to the machine code that runs it). *Journal of Automated Reasoning (JAR)*, 2015. doi:10.1007/978-3-319-08970-6_27.

[7] A. Dold, F. W. von Henke, and W. Goerigk. A completely verified realistic bootstrap compiler. *Int. J. Found. Comput. Sci.*, 14(4):659, 2003. doi:10.1142/S0129054103001947.

[8] J. Earley and H. Sturgis. A formalism for translator interactions. *Commun. ACM*, 13(10):607–617, Oct. 1970. ISSN 0001-0782. doi:10.1145/355598.362740. URL https://doi.org/10.1145/355598.362740.

[9] J. D. Guttman, J. D. Ramsdell, and M. Wand. VLISP: A verified implementation of scheme. *LISP Symb. Comput.*, 8(1-2):5–32, 1995.

[10] J. Harrison. Towards self-verification of HOL light. In U. Furbach and N. Shankar, editors, *Proceedings of the third International Joint Conference (IJCAR)*, volume 4130 of *LNCS*. Springer-Verlag, 2006. doi:10.1007/11814771_17.

[11] H. Jiang, H. Liang, S. Xiao, J. Zha, and X. Feng. Towards certified separate compilation for concurrent programs. In K. S. McKinley and K. Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 111–125. ACM, 2019. doi:10.1145/3314221.3314595.

[12] R. Kumar, R. Arthan, M. O. Myreen, and S. Owens. Self-formalisation of higher-order logic - semantics, soundness, and a verified implementation. *Journal of Automated Reasoning (JAR)*, 56(3):221–259, 2016. doi:10.1007/s10817-015-9357-x.

[13] R. Kumar, E. Mullen, Z. Tatlock, and M. O. Myreen. Software verification with ITPs should use binary code extraction to reduce the TCB - (short paper). In J. Avigad and A. Mahboubi, editors, *Interactive Theorem Proving (ITP)*. Springer, 2018. doi:10.1007/978-3-319-94821-8_21.

[14] O. Lecarme, M. Pellissier, and M.-C. Thomas. Computer-aided production of language implementation systems: A review and classification. *Software: Practice and Experience*, 12(9):785–824, 1982. doi:10.1002/spe.4380120902.

[15] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. doi:10.1145/1538788.1538814.

[16] X. Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009. doi:10.1007/s10817-009-9155-4.

[17] The Coq development team. *The Coq proof assistant reference manual*, 2004. Version 8.0.

[18] J. S. Moore. Milestones from the pure Lisp theorem prover to ACL2. *Formal Asp. Comput.*, 31(6):699–732, 2019. doi:10.1007/s00165-019-00490-3.

[19] M. O. Myreen and S. Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *Journal of Functional Programming (JFP)*, 24(2-3), 2014. doi:10.1017/S0956796813000282.

[20] G. Neis, C. Hur, J. Kaiser, C. McLaughlin, D. Dreyer, and V. Vafeiadis. Pilsner: a compositionally verified compiler for a higher-order imperative language. In K. Fisher and J. H. Reppy, editors, *International Conference on Functional Programming (ICFP)*, 2015. doi:10.1145/2784731.2784764.

[21] S. Owens, M. O. Myreen, R. Kumar, and Y. K. Tan. Functional big-step semantics. In P. Thiemann, editor, *European Symposium on Programming (ESOP)*, LNCS. Springer, 2016. doi:10.1007/978-3-662-49498-1_23.

[22] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992. doi:10.1007/3-540-55602-8_217.

[23] T. Ramananandro, Z. Shao, S. Weng, J. Koenig, and Y. Fu. A compositional semantics for verified separate compilation and linking. In X. Leroy and A. Tiu, editors, *Certified Programs and Proofs (CPP)*, pages 3–14. ACM, 2015. doi:10.1145/2676724.2693167.

[24] J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22:1–22:50, 2013. doi:10.1145/2487241.2487248.

[25] K. Slind and M. Norrish. A brief overview of HOL4. In O. A. Mohamed, C. A. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008. doi:10.1007/978-3-540-71067-7_6.

[26] Y. Song, M. Cho, D. Kim, Y. Kim, J. Kang, and C. Hur. CompCertM: CompCert with C-assembly linking and lightweight modular verification. *Proc. ACM Program. Lang.*, 4(POPL):23:1–23:31, 2020. doi:10.1145/3371091.

[27] M. Sozeau, S. Boulier, Y. Forster, N. Tabareau, and T. Winterhalter. Coq Coq Correct! verification of type checking and erasure for Coq, in Coq. *Proc. ACM Program. Lang.*, 4(POPL), Dec. 2019. doi:10.1145/3371076.

[28] P.-Y. Strub, N. Swamy, C. Fournet, and J. Chen. Self-certification: Bootstrapping certified typecheckers in F* with Coq. In *Principles of Programming Languages (POPL)*. Association for Computing Machinery, 2012. doi:10.1145/2103656.2103723.

[29] Y. K. Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29, 2019. doi:10.1017/S0956796818000229.

[30] M. Wenzel, L. C. Paulson, and T. Nipkow. The isabelle framework. In O. A. Mohamed, C. A. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5170 of *Lecture Notes in Computer Science*, pages 33–38. Springer, 2008. doi:10.1007/978-3-540-71067-7_7.