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:

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
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:

\[ \text{compiler } \text{input} \overset{\text{def}}{=} \text{asm2str (codegen (parser (lexer input)))} \]

This compiler function has type \( \text{string} \rightarrow \text{string} \), and the internal functions have the following types:

- \( \text{lexer} : \text{string} \rightarrow \text{token list} \)
- \( \text{parser} : \text{token list} \rightarrow \text{prog} \)
- \( \text{codegen} : \text{prog} \rightarrow \text{asm} \)
- \( \text{asm2str} : \text{asm} \rightarrow \text{string} \)

Here \( \text{prog} \) is a datatype for the abstract syntax tree (AST) of source programs (which we will see in Section 3), and \( \text{asm} \) 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.

\[ \text{prog2str} : \text{prog} \rightarrow \text{string} \rightarrow \text{string} \]

We have proved that the lexer followed by the parser inverts \( \text{prog2str} \), regardless of the comments \( \text{coms} \):

\[ \vdash \text{parser (lexer (prog2str p coms))} = p \]

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

Before we can show the \( \text{codegen} \) 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 \( \text{prog} \)), we write:

\[ (\text{input}, p) \downarrow_{\text{prog}} \text{output} \]

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

\[ (\text{input}, a) \downarrow_{\text{asm}} \text{output} \]

to say that, with \( \text{input} \) to be read on stdin, target program \( a \) successfully terminates and produces \( \text{output} \) on stdout.

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

\[ \vdash (\text{input}, p) \downarrow_{\text{prog}} \text{output}_1 \land (\text{input}, \text{codegen} \ p) \downarrow_{\text{asm}} \text{output}_2 \Rightarrow \text{output}_1 = \text{output}_2 \]

The operational semantics mentioned above, i.e. \( \downarrow_{\text{prog}} \) and \( \downarrow_{\text{asm}} \), 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 \( \text{codegen} \).

**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 \textit{function type}: \( \text{string} \rightarrow \text{string} \). It would be type incorrect to attempt to apply compiler directly to compiler, since the input type, \text{string}, does not match the type of the argument, \text{string} \rightarrow \text{string}.

The key to this puzzle is to define a new constant, which we call \( \text{compiler} \_\text{prog} \), that is the compiler represented as source AST, i.e. a value of type \( \text{prog} \):

\[ \text{compiler}_\text{prog} : \text{prog} \]

We define \( \text{compiler}_\text{prog} \) in such a way that we can prove that it implements the compiler function:

\[ \vdash (\text{input}, \text{compiler}_\text{prog}) \downarrow_{\text{prog}} \text{compiler input} \]

One should read the theorem above as saying: for any \( \text{input} \), execution of the \( \text{compiler}_\text{prog} \) program will always terminate and the output on stdout is the string produced by applying the compiler function to \( \text{input} \).

Section 4 describes how we produce \( \text{compiler}_\text{prog} \) and prove the correctness theorem above for it.

**Applying the compiler to itself.** Using \( \text{compiler}_\text{prog} \), we can define a concrete-syntax version of it, \( \text{compiler}_\text{str} \); a version expressed in assembly, \( \text{compiler}_\text{asm} \); and the string representation of the assembly version, \( \text{compiler}_\text{asm}_\text{str} \):

\[ \text{compiler}_\text{str} \overset{\text{def}}{=} \text{prog2str compiler}_\text{prog} \text{coms} \]
\[ \text{compiler}_\text{asm} \overset{\text{def}}{=} \text{codegen compiler}_\text{prog} \]
\[ \text{compiler}_\text{asm}_\text{str} \overset{\text{def}}{=} \text{asm2str compiler}_\text{asm} \]

Note that \( \text{compiler}_\text{asm} \) is applying part of the compiler, namely \( \text{codegen} \), to the compiler itself.

From the definition of \( \text{compiler}_\text{asm} \), the definitions above, and the correctness of \( \text{prog2str} \), we can prove an equation describing the result of applying the entire compiler to itself:

\[ \vdash \text{compiler}_\text{str} = \text{compiler}_\text{asm}_\text{str} \]

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.

\[ (\text{input}, \text{compiler_asm}) \downarrow_{\text{asm}} \text{output} \Rightarrow \text{output} = \text{compiler input} \]

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 \( \downarrow_{\text{asm}} \). 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 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.

**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 \( \nu \) as the following recursive datatype.

\[
\nu = \text{Pair } \nu \nu | \text{Num } \text{nart}
\]

**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):

\[
\text{prog} = \text{Program (dec list) exp} \quad \text{dec} = \text{Defun fname (vname list) exp}
\]

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:

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

We did not need a primitive for multiplication.

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

\[
(\text{input}, \text{Program } \text{funs } \text{main}) \downarrow_{\text{prog}} \text{output} \overset{\text{def}}{=} \exists s, r.
\]

\[
(\text{empty Env}, [\text{main}], \text{init state } \text{input } \text{funs}) \downarrow_{\text{exp}} (r, s) \land \text{output} = s, \text{output}
\]

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

\[
(\text{env}, \text{exp}, s) \downarrow_{\text{exp}} (\text{vals}, s')
\]

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.

\[
\text{state} = \{ \text{ input: char llist; output: string; funs: dec list; clock: nat } \}
\]

\(^1\text{Our definition of asm2str is specific to the syntax used by the GNU assembler and the Linux execution environment for the x86-64 architecture.}\)
We prove that theorem.

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

\[ \text{init} \_ \text{state} \quad \text{input} \quad \text{funs} \quad \text{def} \quad \{ \text{input} := \text{input}; \ \text{output} := ""; \ \text{funs} := \text{funs}; \ \text{clock} := 0 \} \]

The \( \downarrow_{\text{exp}} \) relation is defined inductively and is not particularly surprising. The rule for \( 1 \)-const:

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

Similarly, the rule for \( \text{Var} \) is simple:

\[ \text{env} \ n = \text{Some } v \]

\[ (\text{env},[\text{Var } n],s) \Downarrow_{\text{exp}} ([v],s) \]

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

\[ (\text{env},x,s) \Downarrow_{\text{exp}} (v,s_2) \quad \text{app } \text{name } vs \ s_2 \ (v,s_3) \]

\[ (\text{env},[\text{name } x],s) \Downarrow_{\text{exp}} ([v],s_3) \]

The app relation is defined as follows.

\[ \text{env} \_ \text{and} \_ \text{body } \text{name } vs \ s_1 = \text{Some } (\text{env},\text{body}) \quad (\text{env},[\text{body},s_1]) \Downarrow_{\text{exp}} ([v],s_2) \]

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

\[ (\text{env},x,s) \Downarrow_{\text{exp}} (v,s_2) \quad \text{eval} \_ \text{op } f \ vs \ s_2 = (\text{Res } v,s_3) \quad (\text{env},[\text{Op } f \ x],s) \Downarrow_{\text{exp}} ([v],s_3) \]

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

\[ \text{eval} \_ \text{op } \text{Cons } [x; \ y] \ s \ \text{def} \ (\text{Res } (\text{Pair } x \ y),s) \]

\[ \text{eval} \_ \text{op } \text{Head } [\text{Pair } x \ y] \ s \ \text{def} \ (\text{Res } x,s) \]

\[ \text{eval} \_ \text{op } \text{Tail } [\text{Pair } x \ y] \ s \ \text{def} \ (\text{Res } y,s) \]

\[ \text{eval} \_ \text{op } \text{Div } [\text{Num } n_1; \ \text{Num } n_2] \ s \ \text{def} \]

\[ \begin{cases} (\text{Res } (\text{Num } (n_1 \ \text{div} \ n_2)),s) & \text{if } n_2 \neq 0 \text{ then } \text{else } (\text{Err Crash},s) \end{cases} \]

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

4 The Compiler Expressed in AST

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

\[ \vdash (\text{input},\text{compiler} \_ \text{prog}) \Downarrow_{\text{prog}} \text{compiler } \text{input} \]

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.

\[ \text{even } n \ \text{def} \quad \text{if } n = 0 \text{ then } \text{T} \text{ else } \neg \text{even } (n - 1) \]

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

\[ (\text{env},x,s) \Downarrow_{\text{exp}} (\text{encoding } \text{tm},s) \]

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

\[ \text{Bool } \text{T} \ \text{def} \quad \text{Num } 1 \]

\[ \text{Bool } \text{F} \ \text{def} \quad \text{Num } 0 \]

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 \text{T} is:

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

and the lemma for producing Boolean negation (\( \neg \)) is:

\[ \vdash (\text{env},[x],s) \Downarrow_{\text{exp}} ([\text{Bool } b],s) \Rightarrow (\text{env},[\text{Op } \text{Sub } [\text{Const } 1; \ x]],s) \Downarrow_{\text{exp}} ([\text{Bool } (\neg b)],s) \]

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

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

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".

\[ \vdash \text{even} \text{ N} \equiv \text{Some} (\text{Num } n) \land (n \neq 0) \Rightarrow \text{app EVEN } [\text{Num } (n - 1)] s (\text{Bool } (\text{even } (n - 1)), s) \Rightarrow (\text{num} \text{.} \text{even} \text{.} \text{code} \text{.} s) \mid \text{exp} ((\text{Bool } (\text{even } n)), s) \]

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

\[ \vdash \text{lookup} \text{.} \text{fun EVEN } s.\text{funs} = \text{Some} (\text{[N],even} \text{.} \text{code}) \land (n \neq 0) \Rightarrow \text{app EVEN } [\text{Num } (n - 1)] s (\text{Bool } (\text{even } (n - 1)), s) \Rightarrow \text{app EVEN } [\text{Num } n] s (\text{Bool } (\text{even } n), s) \]

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:

\[ \vdash \text{lookup} \text{.} \text{fun EVEN } s.\text{funs} = \text{Some} (\text{[N],even} \text{.} \text{code}) \Rightarrow \text{app EVEN } [\text{Num } n] s (\text{Bool } (\text{even } n), s) \]

which is the theorem returned by our proof automation as a certificate that \text{even} \text{.} \text{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 \text{Exp} for the \text{exp} type. The point of \text{Exp} is to define how the recursive \text{exp} type is represented in our Lisp’s values. Here we use \text{LET} to abbreviate the number representation of the string “Let”.

\[
\text{Exp} \ (\text{Let } n \ x \ y) \overset{\text{def}}{=} \text{Pair } (\text{Num } \text{LET}) \\
(\text{Pair } (\text{Num } n) \\
(\text{Pair } (\text{Exp } x) \\
(\text{Pair } (\text{Exp } y) (\text{Num } 0)))
\]

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

\textbf{Impure functions}. No proof automation was developed for the impure functions because there are only a few of them. The \text{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 \text{let} is used for sequencing.

\[
(\text{defun print } (s) \\
(\text{if } (= s \ '0') \ '0' \\
(\text{let } (v (\text{write } (\text{head } s))) \\
(\text{print } (\text{tail } s)))))
\]

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

\textbf{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 \text{print} function shown above.

\[
(\text{print } (\text{asm} \text{2} \text{str } (\text{codegen } (\text{parser } (\text{lexer})))))
\]

Given a main expression, our automation assembles all of the generated pure functions and all of the parsed impure functions, and then defines the \text{compiler} \text{.} \text{prog} constant.

Once the constant compiler \text{.} \text{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 \text{.} \text{prog} will always do whatever compiler does.

\[
(\text{input} \text{.} \text{compiler} \text{.} \text{prog} \mid \text{input} \Rightarrow \text{compiler} \text{.} \text{prog} \text{.} \text{input})
\]

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.

\textbf{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.

\textit{Abstract syntax}. In our formalisation, an assembly program \text{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.

\[
\text{asm} = \text{inst list}
\]

\begin{verbatim}
| inst = Const reg (64 word) | Mov reg reg |
| Add reg reg | Sub reg reg | Div reg |
| Jump cond nat | Call nat | Ret |
| Pop reg | Push reg |
| Add_RSP nat | Call_RSP nat |
| Load reg reg (4 word) |
| Store reg reg (4 word) |
| GetChar | PutChar | Exit |
| Comment string |
\end{verbatim}

\[
\text{reg} = \text{RAX | RD1 | RBX | RBP | RDX |
| R12 | R13 | R14 | R15 |
\text{cond} = \text{Always | Less reg reg | Equal reg reg}
\]
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.

\[
\text{state = } \emptyset \\
\text{instructions : } \text{asm} ; \\
\text{pc : } \text{nat} ; \\
\text{regs : } \text{reg }\to\text{ 64 word option} ; \\
\text{stack : } \text{word}_\text{or_ret list} ; \\
\text{memory : } \text{64 word }\to\text{ 64 word option} ; \\
\text{input : } \text{char list} ; \\
\text{output : } \text{string} \\
\emptyset
\]

\[
\alpha \text{ option } = \text{None }\mid \text{Some }\alpha \\
\text{word_or_ret } = \text{Word (64 word) }\mid\text{ RetAddr nat}
\]

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

\[
\text{fetch } s \overset{\text{def}}{=} \text{lookup } s . \text{pc } s . \text{instructions}
\]

\[
\text{lookup } n [] \overset{\text{def}}{=} \text{None} \\
\text{lookup } n (x : \alpha :: \alpha s) \overset{\text{def}}{=} \\
\quad \text{if } n = 0 \text{ then Some } x \text{ else lookup } (n - 1) \alpha s
\]

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

\[
\text{step : s_or_h }\to\text{s_or_h }\to\text{ bool}
\]

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.

\[
\text{s_or_h } = \text{State state }\mid\text{ Halt (64 word) string}
\]

\[
^2\text{In Coq, the type of step would be: } \text{s_or_h }\to\text{s_or_h }\to\text{ Prop}
\]

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.

\[
\begin{align*}
\text{fetch } s &\overset{\text{def}}{=} \text{Some (Const r w)} \\
\text{step } (\text{State } s) (\text{State (write_reg r w (inc s))})
\end{align*}
\]

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

\[
\begin{align*}
\text{fetch } s &\overset{\text{def}}{=} \text{Some (Add r1 r2)} \\
\text{s regs } r1 &\overset{\text{def}}{=} \text{Some } w1 \\
\text{s regs } r2 &\overset{\text{def}}{=} \text{Some } w2 \\
\text{step } (\text{State } s) (\text{State (write_reg r1 (w1 + w2) (inc s))})
\end{align*}
\]

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

\[
\begin{align*}
\text{fetch } s &\overset{\text{def}}{=} \text{Some (Call n)} \\
\text{step } (\text{State } s) \\
\text{(State (set_pc n (set_stack RetAddr s.pc + 1): s.stack))}
\end{align*}
\]

\[
\begin{align*}
\text{fetch } s &\overset{\text{def}}{=} \text{Some Ret} \\
\text{s stack } &\overset{\text{def}}{=} \text{RetAddr n:rest} \\
\text{step } (\text{State } s) (\text{State (set_pc n (set_stack rest s))})
\end{align*}
\]

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.

\[
\begin{align*}
\text{fetch } s &\overset{\text{def}}{=} \text{Some Exit} \\
\text{s regs RDI } &\overset{\text{def}}{=} \text{Some exit_code} \\
\text{even (length s stack)} \\
\text{step } (\text{State } s) (\text{Halt exit_code s.output})
\end{align*}
\]

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 to be reachable from the initial state.

\[
\begin{align*}
(input,asm)_\|_{\text{asm output}} &\overset{\text{def}}{=} \\
\exists t . \\
\text{init_state_ok } t \text{ input asm }\land \text{ step* } (\text{State } t) \text{ (Halt 0 output)}
\end{align*}
\]

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

\[
\begin{align*}
\text{init_state_ok } t \text{ input asm } &\overset{\text{def}}{=} \\
\exists r14 r15 . \\
\text{t pc } = 0 \land \text{t instructions } = \text{asm }\land \text{t input } = \text{input }\land \text{t output } = "" \land \text{t stack } = [] \land \text{t regs R14 } = \text{Some } r14 \land \\
\text{t regs R15 } = \text{Some } r15 \land \\
\text{memory_writable } r14 r15 \land \\
\text{t memory}
\end{align*}
\]
Note that we use this \( \llbracket \text{asm} \rrbracket \) as an assumption in our compiler correctness theorems. As a result, the existential inside \( \llbracket \text{asm} \rrbracket \) can be read as a universal quantifier in these theorems:

\[
\vdash ((\text{input,asm}) \llbracket \text{asm} \rrbracket \text{output} \Rightarrow \text{prop}) \\
\iff \\
\forall t. \text{init\_state\_ok} \land \text{input\_asm} \land \\
\text{step}^* (\text{State} t) (\text{Halt} 0 \text{output}) \Rightarrow \text{prop}
\]

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

The memory field of the state record is the following:

记忆 : 64 word \( \rightarrow \) 64 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 the 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.

memory writable \( r_14 \leq+ r_15 \land \text{aligned16} r_14 \land \text{aligned16} r_15 \land r_14 \neq 0 \land \) 
\( \forall a. r_14 \leq+ a <+ r_15 \land \text{aligned8} a \Rightarrow m a = \text{Some None} \)

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.

\[
\vdash (\text{input},p) \llbracket \text{prog} \rrbracket \text{output}_1 \land \\
(\text{input},\text{codegen} p) \llbracket \text{asm} \rrbracket \text{output}_2 \Rightarrow \\
\text{output}_1 = \text{output}_2
\]

This section omits definitions that do not fit here.

### 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:

\[
\alpha \text{app\_list} = \\
| \text{List} (\alpha \text{list}) |
\]

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.

\[
\text{flatten} (\text{List} xs) \text{acc} \overset{\text{def}}{=} xs ++ \text{acc} \\
\text{flatten} (\text{Append} l_1 l_2) \text{acc} \overset{\text{def}}{=} \text{flatten} l_1 (\text{flatten} l_2 \text{acc})
\]

The \( \text{c\_defun} \) function, shown below, generates code for declarations. The generated code consists of a function preamble, generated by \( \text{c\_pushes} \), followed by the code for the body of the function, generated by \( \text{c\_exp} \).

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).

\[
\text{c\_defun} l fs (\text{Defun} n \\text{body}) \overset{\text{def}}{=} \\
\text{let} (c_0, vs l_0) = \text{c\_pushes} vs l \text{in} \\
\text{let} (c_1, l_1) = \text{c\_exp} T l_0 vs fs \text{body} \text{in} \\
(\text{Append} c_0 c_1, l_1)
\]

The function \( \text{c\_defun} \) calls \( \text{c\_exp} \) with \( T \) as the first argument. This indicates that the expression is to be compiled in tail-position. For the \text{Op} case, \( \text{c\_exp} \) calls the non-tail version, i.e. \( \text{c\_exp}\) F. Below \( \text{c\_exp}s \) is a list version of \( \text{c\_exp} \) that
that is defined in mutual recursion with \(c_{\text{exp}}\).

\[
c_{\text{exp}} \quad \text{T} \quad l \quad vs \quad fs \quad \text{(Op \ op \ xs)} \quad \overset{\text{def}}{=} \\
\text{make_ret} \quad vs \quad (c_{\text{exp}} \ F \quad l \quad vs \quad fs \quad \text{(Op \ op \ xs)})
\]

\[
c_{\text{exp}} \quad F \quad l \quad vs \quad fs \quad \text{(Op \ op \ xs)} \quad \overset{\text{def}}{=} \\
\text{let} \quad (c,l') = c_{\text{exp}} \ l \quad vs \quad fs \quad xs \quad \text{in} \\
\text{let} \quad \text{insts} = c \quad \text{op} \quad vs \quad l' \quad \text{in} \\
(\text{Append} \quad c \quad (\text{List} \quad \text{insts}),l' + \text{length} \quad \text{insts})
\]

If the primitive operation \(\text{op}\) is \(\text{Add}\) then the definitions lead us to the following code.

\[
c_{\text{op}} \quad \text{Add} \quad vs \quad l \quad \overset{\text{def}}{=} c_{\text{add}} \quad vs
\]

\[
c_{\text{add}} \quad vs \quad \overset{\text{def}}{=} \\
[\text{Pop} \quad \text{RDI}; \\
\text{Add} \quad \text{RAX} \quad \text{RDI}; \\
\text{Jump} \quad (\text{Less} \quad \text{R13} \quad \text{RAX}) \quad (\text{give_up} \quad (\text{even_len} \quad vs))]
\]

\[
\text{give_up} \quad b \quad \overset{\text{def}}{=} \text{if} \quad b \quad \text{then} \quad 14 \quad \text{else} \quad 15
\]

We can see that \(\text{Add}\) 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 according to our invariant (see \(\text{state_rel}\) 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 \text{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 \(\text{Add}\) 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. \(c_{\text{exp}}\).
state \( t \) represents such a stack \( xs \).

\[
\text{has_stack } t \; xs \; \overset{\text{def}}{=} \\
\exists \; ws. \\
xs = \text{Word } w::\!ws \land t.\text{regs} \; \text{RAX} = \text{Some } w \land \\
t.\text{stack} = ws
\]

In Figure 1, line 4 assumes that a stack is present that can be divided into a current stack frame, \( \text{curr} \), and the rest of the stack, \( \text{rest} \). On line 14, we see that the \( \text{is_tail} \) case requires execution to finish in a state where the \( \text{curr} \) has been dropped and only the return value \( w \) is left in front of \( \text{rest} \). In the non-tail case, on line 17, we see that the return value \( w \) has been pushed onto the stack, leaving \( \text{curr} + \text{rest} \) 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 \( \preceq \) 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 \( \text{Num } n \) 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 \( n2w \ n \).

\[
v_{inv} \; t \; (\text{Num } n) \; w \; \overset{\text{def}}{=} \\
\quad \quad \quad n < 2^{63} \land w = n2w \ n
\]

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.)

\[
v_{inv} \; t \; (\text{Pair } x_1 \; x_2) \; w \; \overset{\text{def}}{=} \\
\quad \exists \; w_1 \; w_2. \\
\quad \quad \text{read_mem } (w + 0) \; t = \text{Some } w_1 \land v_{inv} \; t \; x_1 \; w_1 \land \\
\quad \quad \text{read_mem } (w + 8) \; t = \text{Some } w_2 \land v_{inv} \; t \; x_2 \; w_2 \land \\
\quad \quad \quad w \neq 0
\]

The \( v_{inv} \) relation is used in definition of \( \text{env_ok} \), which relates the source semantics environment \( env \), the current stack frame \( \text{curr} \), and the compiler’s model of the current stack frame \( vs \). We define \( \text{env_ok} \) to say that the model of the stack frame \( vs \) must be exactly the same length as the current stack frame \( \text{curr} \). Furthermore, for any variable binding that exists in the source-level environment \( env \), it must be possible to look up (using \( \text{find} \)) 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.

\[
\text{state_rel } fs \; s \; t \; \overset{\text{def}}{=} \\
\quad s.\text{input} = t.\text{input} \land s.\text{output} = t.\text{output} \land \\
\quad \text{code_rel } fs \; s.\text{funs} \; t.\text{instructions} \land \\
\quad \exists \; r_{14} \; r_{15}. \\
\quad \quad \quad t.\text{regs} \; \text{R12} = \text{Some } 16 \land t.\text{regs} \; \text{R13} = \text{Some } (2^{63} - 1) \land \\
\quad \quad \quad t.\text{regs} \; \text{R14} = \text{Some } r_{14} \land t.\text{regs} \; \text{R15} = \text{Some } r_{15} \land \\
\quad \quad \quad \text{memory_writable} \; r_{14} \; r_{15} \; t.\text{memory}
\]

\[
\text{code_rel } fs \; s.\text{funs} \; \text{instructions} \; \overset{\text{def}}{=} \\
\text{init_code_in} \; \text{instructions} \land \\
\forall \; n \; \text{params.body}.
\]

\[
\text{lookup_fun } \; f \; n \; funs \; = \text{Some } (\text{params.body}) \Rightarrow \\
\exists \; pos. \\
\quad \text{lookup } f s \; n \; = \text{Some } pos \land \\
\quad \text{code_in} \; pos \;
\]

(flatten

\[\text{Defun } (c.\text{defun} \; f \; s \; (\text{Defun } \; n \; \text{params.body}))) \; []\] instructions

\[
\text{init_code_in} \; \text{instructions} \; \overset{\text{def}}{=} \\
\exists \; \text{start}. \\
\quad \text{code_in} \; 0 \; (\text{init} \; \text{start}) \; \text{instructions}
\]

\[
\text{init start} \; \overset{\text{def}}{=} \\
[\text{Const } \text{RAX} \; 0; \text{Const } \text{R12} \; 16; \text{Const } \text{R13} \; (2^{63} - 1); \\
\text{Call } \text{start}; \text{Const } \text{RDI} \; 0; \text{Exit}; \text{Comment } \text{"cons"}; \\
\text{Jump } (\text{Equal } \text{R14} \; \text{R15}) \; 14; \text{Store } \text{RDI} \; \text{R14} \; 0; \\
\text{Store } \text{RAX} \; \text{R14} \; 8; \text{Mov } \text{RAX} \; \text{R14}; \text{Add } \text{R14} \; \text{R12}; \text{Ret}; \\
\text{Comment } \text{"exit 1"}; \text{Push } \text{R15}; \text{Const } \text{RDI} \; 1; \text{Exit}]
\]

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

\[
\text{env_ok } \; env \; vs \; \text{curr} \; t \; \overset{\text{def}}{=} \\
\forall \; n \; v. \\
\quad \text{env} \; n \; = \text{Some } v \Rightarrow \\
\quad \text{find } n \; vs \; 0 < \text{length } \text{curr} \land \\
\quad \exists \; w. \; \text{el}(\text{find } n \; vs \; 0) \; \text{curr} = \text{Word } w \land v_{inv} \; t \; v \; w
\]

\[
\text{find } n \; (\text{None} \; vs) \; k \; \overset{\text{def}}{=} \\
\quad \text{find } n \; vs \; (k + 1)
\]

\[
\text{find } n \; (\text{Some } v \; vs) \; k \; \overset{\text{def}}{=} \\
\quad \text{if } v = \text{then } k \text{ else find } n \; vs \; (k + 1)
\]

Lines 3 and 10 of Figure 1 are still to be explained. Line 3 requires that \( \text{state_rel} \) 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 \( \text{state_rel} \) 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 ($\parallel_{asm}$) and the source program is well-defined (i.e. does not crash), then the source program also diverges ($\parallel_{prog}$). Furthermore, the non-terminating execution agrees on the potentially infinite stream of output that is produced.

$$\vdash prog\text{ avoids\_crash } input \\ prog \land (input,codegen\ prog) \parallel_{asm} output \Rightarrow (input,prog) \parallel_{prog} output$$

The new notation, i.e. $\parallel_{asm}$ and $\parallel_{prog}$, 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.

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

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.

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

Here LUB has type string set $\rightarrow$ char list, where list 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, $(input,asm) \parallel_{asm} output$, 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.

$$\vdash (input,asm) \parallel_{asm} output \quad \text{def} = \exists\ t.\ \text{init\ state\ ok} \ t \ input \ asm \land ((\forall\ k.\ \exists\ t'.\ \text{step}^k\ (State\ t)\ (State\ t')) \land output = \text{LUB}\ \{\ t'.output\ \mid\ \text{step}^*\ (State\ t)\ (State\ t')\ \})$$

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 $\parallel_{asm}$ 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 $k$.

$$\text{eval\ from\ } k \ \text{input\ } p \quad \text{def} = \text{eval\ empty\ env\ main} \quad \text{init\ state\ input\ } \text{funs\ with\ clock\ } := \ k$$
A source program diverges if the program times out for every initial clock value. The output is captured by the least upper bound (LUB) of all partial outputs.

\[
\text{prog\_timeout}\ k\ \text{input\ prog} \triangleq \exists s.\ \text{eval\ from\ input\ prog} = (\text{Err \ TimeOut}, s)
\]

\[
\text{prog\_output}\ k\ \text{input\ prog} \triangleq \text{let}\ (\text{res}, s) = \text{eval\ from\ input\ prog}\ \text{in}\ s.\text{output}
\]

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.

\[
\text{(input, prog)} \uparrow_{\text{prog\_output}} \triangleq \left(\forall k.\ \text{prog\_timeout}\ k\ \text{input\ prog}\right) \land \text{output} = \text{LUB}\ \{(\text{prog\_output}\ k\ \text{input\ prog}) \mid k \in \mathbb{N}\}
\]

**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.

\[
\text{eval env (Const } n) \triangleq \text{return (Num } n)\\
\text{eval env (Var } n) \triangleq \begin{cases} \text{case } env\ n\ \text{of}\ \text{| fail} & \Rightarrow \text{fail} \\ \text{Some } v & \Rightarrow \text{return } v \\ \text{end} & \\ \text{eval env (Op } f\ x) \triangleq \begin{cases} \text{do } vs \leftarrow \text{evals env } x; \text{eval_op } f\ vs & \text{od} \\ \text{eval env (Let } v\ name\ x\ y) \triangleq \begin{cases} \text{do } v \leftarrow \text{eval env } x; \\ \text{eval env } (v\ name \mapsto \text{Some } v) & \text{od} \\ \text{eval env (If } test\ x\ y) \triangleq \begin{cases} \text{do } vs \leftarrow \text{evals env } x; \\ b \leftarrow \text{take\_branch } test\ vs; \\ \text{eval env } (\text{if } b\ \text{then } y\ \text{else } z) & \text{od} \\ \text{eval env (Call } f\ name\ x) \triangleq \begin{cases} \text{do } vs \leftarrow \text{evals env } x; \\ (fenv, body) \leftarrow \text{get\_env\_and\_body } f\ name\ vs; \\ \text{eval fenv body} & \text{od} \\ \text{evals env } [] \triangleq \text{return } [] \\ \text{evals env } (x::xs) \triangleq \begin{cases} \text{do } v \leftarrow \text{eval env } x; \\ vs \leftarrow \text{evals env } xs; \\ \text{return } (v::vs) & \text{od} \\ \text{return } v\ s \triangleq (\text{Res } v, s) \\ \text{fail } s \triangleq (\text{Err \ Crash}, s) \\ \text{monad\_bind } f\ g\ s \triangleq \text{case } f\ s\ \text{of}\ (\text{Res } v, s_1) \Rightarrow g\ v\ s_1 | (\text{Err } e, s_1) \Rightarrow (\text{Err } e, s_1) \\ \end{cases}
\end{cases}
\end{cases}
\end{cases}
\end{cases}
\]

8 Bootstrapping Results and Proof Scripts

This section concludes our description of the bootstrapping work by recappping 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:

\[
\vdash (\text{input, compiler\_asm}) \downarrow_{\text{asm\_output}} \Rightarrow \text{output} = \text{compiler\_input}
\]

The second theorem is an evaluation of the application of the asm2str function to the compiler_asm assembly program (where compiler_asm \(\triangleq\) codegen compiler_prog).

\[
\vdash \text{asm2str compiler\_asm} = "..."
\]

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

\[ \text{prog2str compiler_prog coms} = \ldots \]

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:

\[
\begin{align*}
$ gcc -o compiler compiler_asm.s \\
$ time ./compiler < compiler_prog.txt > output \\
\end{align*}
\]

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.

\[
\begin{array}{ccc}
\text{group of HOL4 files} & \text{# lines} \\
\hline
\text{source syntax, semantics, lemmas} & 1154 \\
\text{x64 assembly syntax, semantics, lemmas} & 609 \\
\text{code generator and its proofs} & 2225 \\
\text{parsing, printing and their proofs} & 1415 \\
\text{compiler_prog, automation and proofs} & 2166 \\
\text{top-level compiler, evaluation and proofs} & 79 \\
\text{total} & 7648 \\
\end{array}
\]

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 compiler 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 mechanically 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 bootstrapped compiler for which bootstrapping was done inside an ITP. CakeML’s compiler bootstrapping works in much the same way as the method described in this paper. However, the CakeML bootstrap 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 explored: (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


