**Journal of Automated Reasoning manuscript No.**
(will be inserted by the editor)

# Proof-grounded bootstrapping of a verified compiler

## Producing a verified read-eval-print loop for CakeML

**Ramana Kumar · Magnus O. Myreen ·
Scott Owens · Yong Kiam Tan**

Received: date / Accepted: date

**Abstract** Compiler verification aims to provide strong quality guarantees about the correctness of compilers. Ultimately, compiler verification aims to remove the need to trust the compiler implementation, i.e. to remove it from the trusted computing base (TCB) of other verification projects. Previous compiler verification projects have, however, not gone far enough towards removing compilers from TCBs. The reason is that previous projects require the use of unverified software for compiling the verified compiler, and thus a significant unverified compiler is still included in the TCB. In this paper, we describe a technique, called *proof-grounded bootstrapping*, for reducing the TCB further. In particular, we explain how one can produce a verified machine-code implementation of a verified compiler by applying the verified compiler to itself. This self-application of the verified compiler is done with proof within the logic of the theorem prover used for verification of the compiler. We show how such a verified compiler can be packaged and used as part of a larger machine-code program. Our example is a verified implementation of a read-eval-print loop (REPL) for the source language. The TCB for this REPL implementation includes only the operating system, the hardware, and the theorem prover: assumptions about the compiler and runtime are replaced by proof. We demonstrate our technique by producing a verified REPL for CakeML, a substantial subset of Standard ML, in x86-64 machine code.

The first author was supported by the Gates Cambridge Trust. The second author was partially supported by the Royal Society UK and the Swedish Research Council. The third author was funded in part by the EPSRC (grant number EP/K503769/1).

R. Kumar
Computer Laboratory, University of Cambridge
E-mail: Ramana.Kumar@cl.cam.ac.uk

M. O. Myreen
CSE Department, Chalmers University of Technology
E-mail: Myreen@chalmers.se

S. Owens
School of Computing, University of Kent
E-mail: S.A.Owens@kent.ac.uk

Y. K. Tan
Computer Laboratory, University of Cambridge
E-mail: ykt23@cam.ac.uk

**Keywords** Compilers · formal verification · bootstrapping

## 1 Introduction

Verified software may exhibit unexpected behaviour when the assumptions of its verification are not satisfied. These assumptions, the trusted computing base (TCB), typically cover the entire production process and execution environment: theorem prover, compiler, runtime, operating system, and hardware. In recent years, there has been much interest [1,8,19,21, 22] in verifying artefacts like compilers that are required to run a wide range of applications, since they are so often included in the TCB. However, if a verified part of the stack can only be run with the aid of unverified components of a similar complexity, the verification story is undermined.

In this paper we present a technique, *proof-grounded bootstrapping*, to close the gap between verifying algorithms for compilation (and garbage collection, etc.) and actually removing the compiler and runtime from the TCB.[1] Is closing the gap worthwhile? Just verifying an algorithm does increase our confidence in its implementation, even if the correspondence between the verified algorithm and the low-level implementation that actually runs needs to be trusted. However, most of the implementation—and its correctness proof—is produced automatically from the verified algorithm when using proof-grounded bootstrapping, making the cost of closing the gap well worth consideration.

We have applied proof-grounded bootstrapping to a large example: the verified compiler for CakeML [21], a programming language and implementation designed for high-assurance applications. We believe CakeML has the smallest TCB of any verified compiler. Although the CakeML compiler was designed with bootstrapping in mind, both its implementation and its correctness theorem are not particularly esoteric. We believe the technique can be used for other verified compilers.

In a nutshell, the idea of proof-grounded bootstrapping is to (automatically) derive a bootstrapping theorem, which states the result of applying the verified compilation algorithm to itself. This bootstrapping theorem includes a low-level implementation of the compiler—the output of running the compiler—in its theorem statement. Composing the bootstrapping theorem with the theorem asserting the algorithm is verified, we conclude that the low-level implementation of the compiler is also verified. Thus the TCB no longer needs to include unverified tools to compile the verified compiler: we can use the verified low-level implementation directly.

To apply proof-grounded bootstrapping, one needs a compiler that satisfies three requirements:

– the compilation algorithm is verified;
– as with ordinary compiler bootstrapping, the compiler is written in its own source language, or, more generally, something that can be translated to its source language; and,
– the compilation algorithm can be computed in the logic used for its verification (for example, the definition can be characterised using rewrite rules and hence computed by rewriting).

Our focus is on how to achieve proof-grounded bootstrapping once these requirements are satisfied, as is the case for CakeML.

[1] We refer to the compiler and runtime used for the verified software itself. Note that the theorem provers that was used for the verification also contains a compiler and runtime. We do not remove the theorem prover's compiler and runtime from the TCB.

The main contribution of this paper is an explanation of proof-grounded bootstrapping, a technique that can be applied directly to a verified compilation algorithm and results in a low-level verified implementation of that compiler (§3).

The second contribution is a demonstration of how such a verified implementation of a compiler can be used as part of a larger verification context. We show how bootstrapping can be used to verify a REPL. We describe the use of bootstrapping to produce a verified REPL for CakeML (§5–§7), and the subtleties involved in mixing bootstrapped and non-bootstrapped code in a single verified implementation (§8–§9).

This paper concludes with our assessment of the trusted computing base (TCB) of the result (§10.1), and a comparison with other compiler verification ideas.

This journal paper gives significantly more detail on CakeML work that was previously published at a conference [21]. The proofs described in this paper have been conducted in the HOL4 theorem prover [39], and are available online from `https://cakeml.org`. s (§10.2).

## 2 Verified algorithms

We make a distinction between algorithms and implementations, which is not always present in other work on verification. For us, an algorithm is a formally specified procedure whose semantics is implicit and mathematical. Using terminology introduced by Boulton et. al. [7, Section 4], an algorithm is a *shallow embedding*, which might be modelled by a function that is defined in the logic of a theorem prover and inherits the semantics of the logic. Implementations, on the other hand, are *deeply embedded*: they are syntax with an explicit formal semantics, for example the operational semantics of a programming language or the next-state relation of a processor model. We make this distinction because the technique we introduce moves from verified algorithms to verified implementations, which we see as finishing the task intended by the algorithm verification in the first place.

Our aim is to explain how we use bootstrapping to turn a verified compilation algorithm into a verified low-level compiler implementation. Although the overall bootstrapping process only applies to a compiler, some of the techniques apply to verified algorithms in general, and are easier to understand on their own terms first. Thus, we begin with an introduction to two techniques: evaluation by rewriting in the logic [3], and proof-producing translation from shallow to deep embeddings [35]. At the end of this section, we turn our attention back to compilers, looking both at what we mean by compiler verification and how the two techniques work when a compiler is involved.

Evaluation in the logic (henceforth “evaluation”) and proof-producing translation to a deep embedding (henceforth “translation”[2]) are both examples of proof automation that can be implemented in the context of a general-purpose theorem prover such as HOL4 [39] (which we use), Isabelle [43], or Coq [5]. Theorem provers (like those three) written in the LCF style [24] produce theorems only by checking the proof steps in a small “kernel” that implements the primitive inference rules of the logic. Sophisticated proof automation, like evaluation or translation, does not demand additional trust since any theorems produced by the automation have been pushed through the theorem prover’s kernel.[3]

[2] Although compilation is a kind of translation (from a high- to a low-level language), we reserve the term “translation” in this paper for moving from a shallow to a deep embedding.

[3] The kernels of these systems vary, however, in size. Coq, for example, includes some facilities for evaluation within the kernel that would need to be implemented outside in other systems.

### 2.1 Evaluation in the logic

Let us begin with an example of the kind of proof task we mean to be solved by evaluation. Given input $\mathsf{map}\ \mathsf{length}\ [[1;\ 1];\ [2];\ []]$, we wish to produce the theorem

$$\vdash \mathsf{map}\ \mathsf{length}\ [[1;\ 1];\ [2];\ []] = [2;\ 1;\ 0]$$

by evaluation using the definitions of map and length. The key characteristic is that the right-hand side of the theorem contains no more reducible expressions: it is a normal form in the rewriting system consisting of the function definitions and beta-conversion. The theorem should be produced automatically and efficiently.

The solution, introduced to HOL4 by Barras [3], is to interpret the equations characterising functions like map (shown below) as they would be by an interpreter for a functional programming language.

$$\begin{array}{l} \mathsf{map} f\ [] = [] \\ \mathsf{map} f\ (h{::}t) = f\ h{::}\mathsf{map} f\ t \end{array}$$

Each reduction step performed by such an interpreter can be justified by a (derived) rule of inference and replayed in the inference kernel, thanks to the kernel's semantics of equality and support for beta-conversion. Logically speaking, evaluation is no more sophisticated than rewriting (or simplification) as described, for example, by Paulson [36]. The difference is in the order in which rewrite rules are applied (bottom-up versus top-down) and in the book-keeping done to make the process more efficient. Although Barras's evaluation supports variables, for our purposes we need only consider evaluation problems, like the one above, where the input term has no free variables.

The equations characterising map above have the same status (proven theorems) as the theorem produced by evaluation. The fact that they can be viewed as *defining* equations does not distinguish them, in HOL, from any other equations. Indeed, any suitable rewrites that have been proved about a function can be used in the evaluation of that function. The resulting theorems, produced by evaluation, are proved using only the normal rules of the inference kernel, without any recourse to evaluation or compilation outside the logic, or purpose-built[4] inference rules for normalisation.

### 2.2 Translation from shallow to deep embeddings

The defining equations for map in the previous subsection are an example of a *shallow embedding* of a functional program in logic. The constant map is a function in HOL with type $(\alpha \to \beta) \to \alpha\ \mathit{list} \to \beta\ \mathit{list}$. Despite the evaluation machinery just described, the semantics of map is not operational; map is a mathematical function and has semantics according to the semantics of HOL. Indeed, there are HOL functions[5] that do not have any operational characterisation.

For functions like map which do have equations suitable for evaluation, in the sense of functional programming, there is an alternative way to model the function in logic. That alternative is to use a *deep embedding*: to model the function as a piece of syntax, animated by an explicit evaluation relation describing the operational semantics of a programming language. In our examples, the programming language for deep embeddings is always CakeML [21] although the ideas apply in general.

[4] One feature was added to the kernel, when evaluation was implemented, to improve the performance: the kernel datatype implementing HOL terms supports lazy substitution.

[5] For example, the existential quantifier over an uncountable type.

Consider the following definition of the syntax for the map function (this is CakeML abstract syntax; it is pretty-printed underneath):

```
map_dec =
 Letrec
  [("map","v3",
   Fun "v4"
    (Mat (Var "v4")
      [(Pcon "nil" [],Con "nil" []);
       (Pcon "::" [Pvar "v2"; Pvar "v1"],
        Con "::"
          [App [Var "v3"; Var "v2"]; App [App [Var "map"; Var "v3"]; Var "v1"]])]))]
```

The syntax is more readable as pretty-printed concrete syntax:

```
fun map v3 v4 =
  case v4
  of [] => []
  |  v2::v1 => (v3 v2::(map v3 v1))
```

The type of map_dec in HOL is *dec* (a CakeML declaration). Thus, it is not a HOL function and does not get its functional semantics that way. Rather, the semantics is given explicitly by an evaluation relation EvalDec $env_1$ *dec* $env_2$ that relates a declaration *dec* and an initial environment $env_1$ (e.g., containing the datatype declaration for lists) to a resulting environment $env_2$. The resulting environment for the map_dec declaration will include a binding of a new variable, called "map", to a function value (i.e., a closure).

If we want to prove something about map, working directly with the syntax and evaluation relation (operational semantics) is much more cumbersome than using the defining equations of the shallow embedding directly. However the extra machinery of the deep embedding (e.g., the environment and the explicit evaluation steps) make it a more realistic formalisation of map as a functional program. Fortunately, we can do our reasoning on the shallow embedding and carry any results over to the more realistic deep embedding automatically using a technique [35] that we call (proof-producing) *translation*.

Translation synthesises a deep embedding following the structure of the shallow embedding's equations and simultaneously proves a *certificate theorem* about the synthesised implementation. Synthesis happens in a bottom-up manner, using the certificate theorems for previously translated code as required. The certificate theorem is proved automatically, using the shallow embedding's induction theorem (typically proved automatically when the shallow embedding is defined) and relates the behaviour of the synthesised implementation to its shallow counterpart.

To explain certificate theorems, let us work through understanding the following one for map by taking it apart.

*Example 1 (Certificate theorem for* map*)*

$$\begin{array}{l}\vdash \exists env\ c.\\ \quad \mathsf{EvalDec\ InitEnv\ map\_dec}\ env \wedge \mathsf{Lookup}\ \texttt{"map"}\ env = \mathsf{Some}\ c\ \wedge \\ \quad ((a \longrightarrow b) \longrightarrow \mathsf{ListTy}\ a \longrightarrow \mathsf{ListTy}\ b)\ \mathsf{map}\ c\end{array}$$

There are two important concepts contained in such a certificate theorem: refinement invariants (e.g., ListTy $a$) and the operational semantics (EvalDec). A refinement invariant specifies the relationship between between a shallowly-embedded value (a HOL term) and a deeply-embedded one (a CakeML value). For example, $\mathsf{ListTy}\ \mathsf{BoolTy}\ [\mathsf{F}]\ v$ holds when $v$ is a CakeML value implementing the singleton list containing the HOL constant $\mathsf{F}$ (falsity) according to the refinement invariant $\mathsf{ListTy}\ \mathsf{BoolTy}$. Expanding out what this means explicitly, we have the following theorem. Here, $\mathsf{Conv}\ \mathit{name}\ \mathit{args}$ represents a deeply-embedded constructor value.

$$\begin{array}{l}\vdash \mathsf{ListTy}\ \mathsf{BoolTy}\ [\mathsf{F}]\ v \iff \\ \quad v = \\ \quad\ \ \mathsf{Conv}\ (\texttt{"::"},\mathsf{TypeId}\ \texttt{"list"}) \\ \quad\quad [\mathsf{Conv}\ (\texttt{"false"},\mathsf{TypeId}\ \texttt{"bool"})\ [];\ \mathsf{Conv}\ (\texttt{"nil"},\mathsf{TypeId}\ \texttt{"list"})\ []]\end{array}$$

Since lists are polymorphic, $\mathsf{ListTy}$ takes as an argument a refinement invariant to govern the type of the list elements. In the certificate theorem for $\mathsf{map}$ above (Example 1), there are free variables $a$ and $b$ standing for refinement invariants for the input and output list elements. The free variables show us that the certificate theorem applies to every instance of the polymorphic $\mathsf{map}$ function.

The full refinement invariant for $\mathsf{map}$ includes several instances of the refinement invariant, $x \longrightarrow y$, for functions (there are several instances because $\mathsf{map}$ is both higher-order and curried). Given a HOL function $f$ and refinement invariants $x$ and $y$ intended to describe the input and output types of $f$, the $(x \longrightarrow y)\, f\, c$ invariant holds when $c$ is a CakeML closure that implements $f$. More specifically, whenever $x\ v_1$ holds, then application of the closure $c$ to $v_1$ will, according to the CakeML operational semantics, terminate with a value $v_2$ that satisfies $y\ (f\ x)\ v_2$. Looking back at the refinement invariant for $\mathsf{map}$ in its certificate theorem, we see that $\mathsf{map}$ is implemented as a closure which, when given CakeML values satisfying $(a \longrightarrow b)\, f$ and $\mathsf{ListTy}\ a\ l$ as inputs will terminate and produce a CakeML value satisfying $\mathsf{ListTy}\ b\ (\mathsf{map}\, f\ l)$.

The certificate theorem for $\mathsf{map}$ is written in terms of the operational semantics, namely $\mathsf{EvalDec}$. In general $\mathsf{EvalDec}\ \mathit{env}_1\ \mathit{dec}\ \mathit{env}_2$ is the assertion that the CakeML declaration $\mathit{dec}$ evaluates in environment $\mathit{env}_1$ successfully and without side-effects[6] to produce the extended environment $\mathit{env}_2$. Thus for $\mathsf{map}$, we see that in initial environment $\mathsf{InitEnv}$, the $\mathsf{map\_dec}$ declaration will succeed and the resulting environment, $\mathit{env}$, will bind the variable `"map"` to a closure, $c$, implementing $\mathsf{map}$. It is not particularly important that we start in the $\mathsf{InitEnv}$ environment, which contains only CakeML primitives: a more general form of the certificate theorem (not shown) allows us to derive a similar result for any starting environment.

The proof-producing translation technique includes support for user-defined datatypes as well as the primitive datatypes of CakeML (Booleans, lists, etc.). The result of defining an algebraic data type in HOL provides enough information to synthesise refinement invariants (like $\mathsf{ListTy}\ a$) for new types. There is also some support (mainly namespace management) for translation into a named CakeML module. Details on the workings of the proof automation behind translation can be found in our previous paper [35]. Proof-producing translation plays a key role in bootstrapping the CakeML compiler, which is itself written in HOL but whose input language is CakeML.

[6] Certificate theorems for programs with side-effects are more complicated, but will not concern us until Section 8.

$$\begin{tikzcd} \boxed{env_1} \arrow[r, "dec"] \arrow[d, "\mathsf{Inv}"'] & \boxed{env_2} \arrow[d, dashed, "\mathsf{Inv}"] \\ \boxed{bs_1} \arrow[r, dashed, "bc"'] & \boxed{bs_2} \end{tikzcd}$$

**Fig. 1** Compiler correctness (Lemma 1) illustrated as a commuting diagram. On the top is evaluation of *dec* in the CakeML operational semantics. On the bottom is evaluation of bytecode, *bc*, resulting from compiling *dec*. Lemma 1 states that the dashed lines exist whenever the solid ones do.

## 2.3 Compiler verification

In this section we look at compilation as a verified algorithm: what its correctness theorem looks like, and how the algorithm interacts with both evaluation in the logic and proof-producing translation to a deep embedding. For now, we focus on compilation from CakeML abstract syntax (as seen in map_dec in the previous section) to CakeML bytecode (Section 6), an assembly-like language that operates over structured data and is our main stepping stone on the way to real machine code. When we package the compiler in a read-eval-print loop (REPL), we will add verified parsing from CakeML concrete syntax (i.e., a string), and further verified compilation to x86-64 machine code (i.e., numbers).

A compiler is a program for translating code from a high-level language to a low-level language, and the property usually considered to constitute its correctness is *semantics-preservation*. We define the CakeML compiler as a function in the logic, since that is the natural place for carrying out verification; the compilation algorithm is defined as a shallow embedding like map in the previous section. Such a shallow embedding, together with a correctness theorem, is what is typically meant by a "verified compiler", for example the CompCert verified compiler [23] is a verified algorithm in our terminology. CompCert is run by being extracted to OCaml (which is unverified). Using bootstrapping we eventually verify a much more concrete implementation of the CakeML compiler.

To verify the compiler, we need semantics for both the high-level- and low-level languages. We have seen examples of the semantics for CakeML in the previous section, in particular EvalDec $env_1$ $dec$ $env_2$ which specifies the evaluation of a declaration. The semantics of CakeML bytecode is given as a state-transition system, $bs_1 \to^* bs_2$, which means bytecode-machine state $bs_1$ transitions to state $bs_2$ in zero or more steps. The bytecode-machine states (explained more thoroughly in Section 6) contain code and a program counter, as well as the current state of the memory.

A call to the compiler looks like this: CompileDec $cs_1$ $dec = (cs_2,bc)$, where $cs_1$ and $cs_2$ are the compiler's internal state and $bc$ is the generated bytecode. Because we eventually want to call the compiler multiple times in succession (for the REPL), we prove preservation not just of semantics of the input program but of an invariant, Inv $env$ $cs$ $bs$, between the environment $env$ in the CakeML semantics, the compiler's state $cs$, and the bytecode-machine state $bs$. This is an example of *forward simulation* [11].

The compiler correctness theorem states that if the invariant holds for an environment $env_1$, and the semantics of $dec$ in that environment produces $env_2$, then the compiled code for $dec$ will run to completion and the invariant will hold again in $env_2$. The statement is illustrated in Figure 1, and printed formally below.

**Lemma 1 (Correctness of CompileDec for successful declarations)**

$$\begin{array}{l}\vdash \mathsf{Inv}\ env_1\ cs_1\ bs_1 \wedge \mathsf{EvalDec}\ env_1\ dec\ env_2 \wedge \mathsf{CompileDec}\ cs_1\ dec = (cs_2,bc) \Rightarrow \\ \quad \exists bs_2.\ (\mathsf{AddCode}\ bs_1\ bc) \to^* bs_2 \wedge \mathsf{Halted}\ bs_2 \wedge \mathsf{Inv}\ env_2\ cs_2\ bs_2\end{array}$$

This form of compiler correctness theorem is only suitable for source programs that terminate successfully. For bootstrapping, that is the important case, since we prove that the CakeML compiler always terminates successfully. The CakeML compiler is, however, also verified for the cases of diverging and failing input programs; we will reason about these cases when we want to run the verified compiler at runtime in a read-eval-print loop (REPL, Section 4). We will not go further into the details of the invariant, except to say that it embodies data refinement from CakeML source values to CakeML bytecode values with more than enough fidelity for verified printing.

Now we have seen the compiler, CompileDec, as a verified algorithm. Let us look at some examples of applying our verified algorithm techniques, evaluation and translation, to the compiler. Firstly, we can evaluate applications of the compiler to CakeML programs in the logic, for example to map_dec. Applying evaluation to the input term CompileDec InitCS map_dec, we obtain the following theorem, where MapCS stands for the concrete compiler state that results.

*Example 2 (Evaluating the compilation of* map*)*

$$\begin{array}{l}\vdash \mathsf{CompileDec\ InitCS\ map\_dec} = \\ \quad (\mathsf{MapCS}, \\ \quad\ [\mathsf{Jump\ (Lab\ 12);\ Label\ 10;\ Stack\ (PushInt\ 0);\ Stack\ (PushInt\ 1);\ Ref;} \\ \quad\ \ \mathsf{PushPtr\ (Lab\ 11);\ Stack\ (Load\ 0);\ Stack\ (Load\ 5);\ Stack\ (PushInt\ 1);} \\ \quad\ \ \mathsf{Stack\ (Cons\ 0);\ Stack\ (\ldots\ldots);\ \ldots\ldots;\ \ldots;\ \ldots\,])}\end{array}$$

Thus we can see that evaluation results in a theorem that produces a concrete list of bytecode for map_dec, to which the conclusion of the correctness theorem for CompileDec (Lemma 1) applies.

In addition to evaluating the compiler as a function in the logic, we can also use translation to produce an implementation of the compiler as a deep embedding. In other words, just as we produced map_dec plus its certificate theorem from the map algorithm, we can produce syntax and a certificate theorem from the compilation algorithm (the shallow embedding CompileDec). Since compilation is a rather more involved algorithm than map, it is split into 247 declarations of auxiliary functions and datatypes. We use translation to produce a CakeML module (called "C" below) containing all these declarations (called CompileDec_decs below). Just as for map, the certificate theorem for CompileDec shows that the generated CakeML code runs successfully in the initial environment to produce an environment, abbreviated as CompEnv, containing a closure that implements CompileDec.

**Lemma 2 (Certificate theorem for CompileDec)**

$$\begin{array}{l}\vdash \exists c. \\ \quad \mathsf{EvalDec\ InitEnv\ (Struct\ \texttt{"C"}\ CompileDec\_decs)\ CompEnv} \wedge \\ \quad \mathsf{LookupMod}\ \texttt{"C"}\ \texttt{"compiledec"}\ \mathsf{CompEnv} = \mathsf{Some}\ c \wedge \\ \quad (\mathsf{CompStateTy} \longrightarrow \mathsf{DecTy} \longrightarrow \mathsf{PairTy\ CompStateTy\ (ListTy\ BCInstTy)})\ \mathsf{CompileDec}\ c\end{array}$$

The result of translating CompileDec includes CakeML syntax for the compiler, namely CompileDec_decs. A natural question is what happens if we use evaluation of CompileDec on the syntax for CompileDec produced by translation. What can we conclude about the resulting bytecode? This question is the idea behind proof-grounded bootstrapping, to which we now turn.

## 3 Proof-grounded bootstrapping

The aim of bootstrapping is to obtain a verified low-level implementation of a compiler directly from the verified compilation algorithm, and to thereby remove the need to trust the process by which the verified compilation algorithm gets compiled. Let us see how we obtain this verified low-level implementation automatically through a combination of the proof-producing-translation and evaluation-by-rewriting proof automation techniques.

Via translation we have obtained CakeML syntax for the compiler (CompileDec_decs). Now, we use evaluation to calculate the application of the compiler to its syntax. This is analogous to Example 2 but instead of using map_dec as input, we use the module declaring the compiler. The result of this evaluation is what we call the *bootstrapping theorem*.

**Lemma 3 (Bootstrapping theorem for CompileDec)**

$$\vdash \mathsf{CompileDec}\ \mathsf{InitCS}\ (\mathsf{Struct}\ \texttt{"C"}\ \mathsf{CompileDec\_decs}) = (\mathsf{CompCS}, \mathsf{CompileDec\_bytecode})$$

The bootstrapping theorem contains a concrete list of bytecode instructions that is the code generated by the compiler for the CompileDec_decs module, which we have abbreviated as CompileDec_bytecode.

Three theorems come together to create proof-grounded bootstrapping. Each corresponds to a different level of concreteness for the compiler, namely, the algorithm, the high-level implementation in CakeML, and the low-level implementation in bytecode. They can be described as follows:

– Correctness theorem: the output of the compiler implements the input, for all inputs. This theorem is about the compilation algorithm (shallow embedding), and corresponds to Lemma 1.
– Certificate theorem: the syntax for the compiler (CompileDec_decs) implements the compiler. This theorem is about the high-level implementation of the compiler produced by translation, and corresponds to Lemma 2.
– Bootstrapping theorem: the output of the compiler when given its syntax as input is low-level code for the compiler (CompileDec_bytecode). This theorem contains the low-level implementation of the compiler produced by evaluation, and corresponds to Lemma 3.

Instantiating the correctness theorem with the bootstrapping theorem, then composing it with the certificate theorem, we obtain the desired result that the low-level code for the compiler implements the compiler. That is the method behind proof-grounded bootstrapping.

The essence of proof-grounded bootstrapping is a consideration for the three levels of concreteness: algorithm (CompileDec), syntax (CompileDec_decs), and low-level code (CompileDec_bytecode). It is bootstrapping because the syntax happens to be syntax for the compiler. The approach can be generalised by using any other certified syntax instead. We call the general approach *proof-grounded compilation*. The generalisation of the bootstrapping theorem is a *compilation theorem* since it captures the result of a particular compilation. For the CakeML REPL (Section 4), we apply proof-grounded compilation to a certificate theorem covering not just CompileDec_decs but also syntax for a verified parser and type inferencer.

In the sketch above, we used the word "implements" loosely. Let us look now at precisely what we obtain by following the bootstrapping method, and what assumptions remain undischarged. The compiler correctness theorem, repeated below, has three antecedents: the invariant, evaluation of the semantics, and an application of the compiler.

**Lemma 1 (Correctness of CompileDec for successful declarations)**

$$\begin{array}{l}\vdash \mathsf{Inv}\ env_1\ cs_1\ bs_1 \wedge \mathsf{EvalDec}\ env_1\ dec\ env_2 \wedge \mathsf{CompileDec}\ cs_1\ dec = (cs_2,bc) \Rightarrow \\ \quad \exists bs_2.\ (\mathsf{AddCode}\ bs_1\ bc) \rightarrow^* bs_2 \wedge \mathsf{Halted}\ bs_2 \wedge \mathsf{Inv}\ env_2\ cs_2\ bs_2\end{array}$$

Following the bootstrapping method, we instantiate Lemma 1 so that the application of the compiler matches the bootstrapping theorem (Lemma 3). Evaluation of the semantics come from the certificate theorem (Lemma 2). To establish the initial invariant we can easily construct a bytecode machine state, InitBS, that only contains the primitives and satisfies the invariant:

**Lemma 4 (Initial invariant)**

$$\vdash \mathsf{Inv}\ \mathsf{InitEnv}\ \mathsf{InitCS}\ \mathsf{InitBS}$$

After instantiating the correctness theorem and proving its hypotheses as just described, we are left with a conclusion that states that CompileDec_bytecode runs to completion and the resulting bytecode state satisfies the invariant at CompEnv, the environment containing the compiler:

**Lemma 5 (Result of bootstrapping)**

$$\begin{array}{l}\vdash \exists bs_2.\\ \quad (\mathsf{AddCode}\ \mathsf{InitBS}\ \mathsf{CompileDec\_bytecode}) \rightarrow^* bs_2 \wedge \mathsf{Halted}\ bs_2 \wedge \\ \quad \mathsf{Inv}\ \mathsf{CompEnv}\ \mathsf{CompCS}\ bs_2\end{array}$$

In other words, according to the semantics of bytecode execution, we can produce a bytecode machine state, $bs_2$ above, that implements CompEnv. The certificate theorem (Lemma 2) tells us that CompEnv contains a closure (bound by the variable `"compiledec"` in the `"C"` structure) that implements the CompileDec function according to the refinement invariants of translation. Thus, the bytecode machine state asserted to exist above ($bs_2$) contains a low-level implementation of the compiler, CompileDec, as promised.

The usefulness of Lemma 5 depends on the strength of the refinement invariant of translation ($\mathsf{CompStateTy} \longrightarrow \mathsf{DecTy} \longrightarrow \ldots \ldots$) connecting the implementation in CakeML to the shallow embedding, and the invariant (Inv) connecting the implementation in bytecode to the implementation in CakeML. What Lemma 5 provides is a closure implementing the compiler according to the refinement invariants. In fact, the invariants are strong enough for any use of the closure that depends only on its functional (i.e., input/output) behaviour. To support this claim about the usefulness of Lemma 5, we detail a particular approach to using a bootstrapped compiler in a larger context. Our approach is to package the verified compiler within a read-eval-print loop (REPL). In the next section, we motivate and describe the REPL, starting with an introduction to how verified compilers can be used in general.

## 4 Packaging a bootstrapped compiler as a REPL

A compiler can be used as a standalone application, which does no more than take high-level code as input and produce low-level code as output. We call this kind of application a *standalone compiler*. If the compiler is verified, there will be a correctness theorem about running the low-level code under particular conditions. The correctness theorem is vacuous unless its assumptions are met. For example, Lemma 1 states that the low-level code $bc$ output by CompileDec preserves the Inv invariant, which assumes the invariant holds in the first place. Lemma 4 states that InitBS satisfies the invariant, so it is sufficient to load the output of the compiler into InitBS before it is run. For a standalone compiler, it is up to the user to run the output of the compiler in such a way that satisfies the conditions of the correctness theorem if they want to leverage the verification.

With a view to reducing the trusted computing base (TCB), there is an extension to a standalone compiler that we call a *packaged compiler*, where the compiler is included within a larger verified program that always runs the compiler's output in a way that satisfies the assumptions of the compiler's correctness theorem. A packaged compiler does more than compilation: it compiles, loads, and runs code. And because it is self-contained, a verified package has a simpler correctness theorem than a verified standalone compiler. It allows us to focus our trust in the operating system and hardware on a single point: correct execution of the machine-code implementation of the whole package.

One way to package a compiler is as a *one-shot* package, which always uses the initial compiler state (for CompileDec that is InitCS) and loads the result of compilation into a fresh initial machine state (for bytecode that is InitBS) for execution. For a one-shot package, the *wrapper* (i.e., non-compiler) code reads the input (high-level program), feeds it to the compiler, loads the output (low-level program) into an appropriate runtime environment, then jumps to the loaded program. A one-shot package is not interactive: the entire program and its input is prepared before compilation, and any further interaction is via input/output (I/O) primitives called from within the program.

By putting the wrapper into a loop, however, we obtain a *read-eval-print loop* (REPL), which is inherently interactive. A REPL intersperses execution of the compiler with execution of its output and retains state between calls to the compiler, thus later input code can depend on the results of previously input code. Since CakeML does not presently have I/O primitives, a REPL is essential for interaction; it is also a more interesting example of a packaged compiler since the compiler can be called multiple times.

To verify a machine-code implementation of a packaged compiler, it is necessary to have a machine-code implementation of the compiler itself whose correctness theorem is strong enough to support execution of the compiler at (package) runtime. A verified compilation algorithm on its own is not enough to produce a verified REPL in machine code. It is the push from verified algorithms down to a verified implementation that enables production of such machine-code programs that contain the verified compiler.

Our goal now is to explain how, using proof-grounded bootstrapping, we were able to produce machine code that is verified to implement a REPL for CakeML. Each piece of verified machine code comprising the REPL is obtained by one of two methods. The first method is bootstrapping, which provides code for most of the compiler. The second method is *decompilation into logic* [28,33,34] (henceforth "decompilation"), which is used for the rest of the compiler and the wrapper code.

Decompilation is a tool-assisted but manual procedure for verifying programs written either directly in assembly code or as functions in the logic in a particular tail-recursive style. Because of the effort required—compared to the fully automated bootstrapping method—we use it only for those parts of the REPL that must be implemented at a low level in our design, such as the garbage collector and the (simple) compiler from CakeML bytecode to x86-64 machine code.

The REPL and its verification comprise three layers as shown in Figure 2. At the top of the figure is the semantics of the REPL (ReplSem), which builds on the semantics for CakeML programs (EvalDec) that we have already seen. We describe the semantics of the REPL in Section 5. In the middle of the figure is a description of the REPL as a function in the logic (ReplFun), which replaces the CakeML operational semantics with the seman-

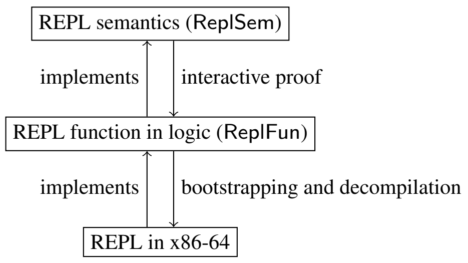

**Fig. 2** Overview of verified REPL construction.

tics for CakeML bytecode by packaging a verified compiler, called ParseInferCompile (Section 7), from concrete syntax to bytecode. This middle layer is almost an algorithm for the REPL, but deals with divergence non-algorithmically in terms of traces in the bytecode semantics. At the bottom of the figure is the implementation of the REPL package in machine code, which is produced and verified by a combination of the bootstrapping and decompilation techniques.

The REPL function in the logic acts as an intermediary between the semantics of the REPL and the machine code that is ultimately produced. It is treated like an implementation of the REPL semantics, but acts as a specification for the machine-code implementation. The specification is of the entire REPL package, that is, both the compiler and the wrapper. The compiler, ParseInferCompile, used inside ReplFun extends the CompileDec compiler we have already seen with the addition of a verified parser from concrete syntax and verified type checking. We describe the definition and verification of ReplFun in Section 7.

To produce the final machine-code implementation, most of ReplFun is bootstrapped and the remaining code is produced more manually using decompilation into logic. In connecting the bootstrapped and non-bootstrapped code, we face the issue of *using* the bootstrapped function—in particular, giving input and retrieving output—touched on at the end of the previous section. We need to be able to call the closure asserted to exist after bootstrapping, and know that it will behave correctly. For this purpose, we bootstrap not just the definition of the compiler but also a declaration of a call to the compiler. We explain this small extension to the bootstrapping idea in Section 8.

The non-bootstrapped code comes in two categories: firstly, there is the lexer and the loop that calls the compiler on the result of lexing and jumps to its output; secondly, there is the runtime that implements a CakeML bytecode machine, which includes additional (previously verified) machine-code libraries for garbage collection [29] and arbitrary-precision integer arithmetic [30]. The main subtlety in producing a packaged compiler by bootstrapping in this way is that there are logically two distinct bytecode machine states to consider: one for running bootstrapped code, and another simulating the bytecode machine that is explicitly mentioned in ReplFun and runs user code. We describe the construction and verification of this final layer in Section 9.

## 5 Semantics of the REPL

Recall the operational semantics of CakeML declarations: EvalDec $env_1$ *dec* $env_2$ holds when the semantics of processing the declaration *dec* in environment $env_1$ is to produce a new environment $env_2$. The semantics for a read-eval-print loop (REPL) is to read and evaluate declarations in a loop, printing the additional bindings in the new environment after each one. For simplicity, we have ignored the specification of stateful and failing programs: in the real CakeML semantics, the result of processing a declaration is a new store together with either an exception or a new environment as above. However, since our main concern is what is printed as a result of processing a declaration, and what the new state of the semantics is for the next declaration, we continue to ignore the details of stateful and failing computations.

We must, however, account for the possibility of divergence. We do so by ensuring the semantics covers all non-diverging possibilities with explicit errors where required, so that if a declaration has no semantics it must diverge. Additionally, in CakeML we have a small-step operational semantics where divergence can be specified in the normal way (as an infinite trace), and we have proved that small-step divergence is equivalent to a failure to be related by the big-step semantics.

We model the output of the REPL using the following type, which encodes a list of result strings ending in either termination or divergence.

$$\mathit{repl\_result} \;=\; \mathsf{Terminate} \mid \mathsf{Diverge} \mid \mathsf{Result}\ \mathit{string}\ \mathit{repl\_result}$$

Each result is the output from a declaration: it could indicate a parse error, a type error, an exception, or some new bindings. If some declaration diverges, the REPL result ends there with Diverge; otherwise it continues until there are no more declarations and ends with Terminate.

We model the input of the REPL as a string containing all user input. In reality, later parts of the user input are likely to depend on the REPL's output for earlier parts. But since we do not model the user at all, apart from the input they actually produce, it is convenient to assume we have all the input up front, akin to an oracle.

The concrete syntax for CakeML requires that every declaration end with a semicolon. Consequently, the input string can be split, after lexing, into lists of tokens each representing a declaration. To specify the semantics of lexing, we have executable specifications (Lex and SplitSemicolons) of the conversion to tokens and splitting at semicolons. For the semantics of parsing, the (non-executable) function Parse checks whether there exists a parse tree for a declaration in the CakeML grammar whose fringe is the given list of tokens, and returns Some *dec* if so, otherwise None. The semantics of the entire REPL, shown below, can thus be factored through a semantics (AstReplSem) that operates on abstract syntax.

$$\mathsf{ReplSem}\ \mathit{state}\ \mathit{input} = \mathsf{AstReplSem}\ \mathit{state}\ (\mathsf{map}\ \mathsf{Parse}\ (\mathsf{SplitSemicolons}\ (\mathsf{Lex}\ \mathit{input})))$$

Let us look now at the AstReplSem relation, of which the signature is shown below.

$$\mathsf{AstReplSem}\ \mathit{state}\ \mathit{dec\_options}\ \mathit{repl\_result}$$

The first argument is the state of the REPL semantics, in particular that means the state of the type system (the types declared so far) and of the operational semantics (the current environment and store). As we saw above, ReplSem is parameterised by an initial state thereby allowing a basis program before user input.

With our model of what a REPL result looks like, the definition of AstReplSem is a straightforward loop down the list of input declarations. For each declaration in the list *dec_options*:

1. if it is None accumulate a parse-error result, otherwise
2. if the declaration is not well-typed according to the type system, accumulate a type-error result, otherwise

$$
\begin{array}{lcl}
\mathit{bc\_inst} & ::= & \mathsf{Stack}\ \mathit{bc\_stack\_op} \mid \mathsf{PushExc} \mid \mathsf{PopExc} \\
 & \mid & \mathsf{Return} \mid \mathsf{CallPtr} \mid \mathsf{Call}\ \mathit{loc} \\
 & \mid & \mathsf{PushPtr}\ \mathit{loc} \mid \mathsf{Jump}\ \mathit{loc} \mid \mathsf{JumpIf}\ \mathit{loc} \\
 & \mid & \mathsf{Ref} \mid \mathsf{Deref} \mid \mathsf{Update} \mid \mathsf{Print} \mid \mathsf{PrintC}\ \mathit{char} \\
 & \mid & \mathsf{Label}\ n \mid \mathsf{Tick} \mid \mathsf{Stop} \mid \ldots \\
\mathit{bc\_stack\_op} & ::= & \mathsf{PushInt}\ \mathit{int} \mid \mathsf{Pop} \mid \mathsf{Pops}\ n \\
 & \mid & \mathsf{Load}\ n \mid \mathsf{Store}\ n \\
 & \mid & \mathsf{Cons}\ n \mid \mathsf{El} \mid \mathsf{TagEq}\ n \mid \mathsf{IsBlock} \mid \mathsf{LengthBlock} \\
 & \mid & \mathsf{Equal} \mid \mathsf{Less} \mid \mathsf{Add} \mid \mathsf{Sub} \mid \mathsf{Mult} \mid \mathsf{Div} \mid \mathsf{Mod} \\
\mathit{loc} & ::= & \mathsf{Lab}\ n \mid \mathsf{Addr}\ n \\
n & = & \mathit{num}
\end{array}
$$

**Fig. 3** CakeML bytecode instructions.

3. if the operational semantics of the declaration is to diverge, end with the Diverge result, otherwise
4. accumulate the (exceptional or normal) result of the operational semantics of the declaration, update the state (with the new results from the operational semantics and type system), and continue.

This specification of the REPL semantics corresponds to the top layer of Figure 2. The middle layer, the ReplFun function in the logic, describes an implementation strategy for the REPL that mimics the loop above but replaces the CakeML operational semantics with execution of compiled code according to CakeML bytecode semantics. Before we turn our attention to ReplFun, let us take a look at CakeML bytecode in more detail.

## 6 CakeML bytecode

The purpose of CakeML bytecode is to abstract over the details of data representation as machine words, and to hide the garbage collector, while being sufficiently low level for most of its instructions to map directly to small snippets of x86-64 machine code. bytecode is the low-level code produced by the REPL function in logic (ReplFun) in the middle layer of Figure 2. All production and verification of real machine code is isolated in the bottom layer.

In support of data abstraction, bytecode values do not explicitly model pointers into the heap but instead provide structured data (Cons packs multiple bytecode values *vs* into Block *tag vs*) on the stack. Similarly, the bytecode provides mathematical integers (Number *i*) as values on the stack, abstracting over the representation as either small integers (that fit in a machine word) or pointers to heap-allocated big integers. Apart from blocks and integers, the only other bytecode values are special-purpose pointers into the heap (RefPtr *p*, for implementing references), into the code heap (CodePtr *p*, for closures and dynamic jumps), and into the stack (StackPtr *p*, for implementing exceptions).

The bytecode semantics is a deterministic state machine, operating over bytecode machine states, *bs*, that contain code (*bs*.code), a program counter (*bs*.pc), and a list of bytecode values (*bs*.stack). The state transition relation, $bs_1 \to bs_2$, fetches the instruction in the program counter and updates the state according to its semantics. A selection[7] of bytecode instructions are shown in Figure 3, and a selection of their semantics in Figure 4.

[7] Not shown, for simplicity, are instructions supporting additional primitive types (characters and byte arrays) and global variables.

$$\frac{\mathit{fetch}\ \mathit{bs} = \mathsf{Stack}\ (\mathsf{Cons}\ t) \quad \mathit{bs}.\mathsf{stack} = \mathsf{Number}\ n{::}\mathit{vs}\ @\ \mathit{xs} \quad \mathsf{length}\ \mathit{vs} = n}{\mathit{bs} \to (\mathsf{bump}\ \mathit{bs})\{\mathsf{stack} = \mathsf{Block}\ t\ (\mathsf{reverse}\ \mathit{vs}){::}\mathit{xs}\}}$$

$$\frac{\mathit{fetch}\ \mathit{bs} = \mathsf{Return} \quad \mathit{bs}.\mathsf{stack} = x{::}\mathsf{CodePtr}\ \mathit{ptr}{::}\mathit{xs}}{\mathit{bs} \to \mathit{bs}\{\mathsf{stack} = x{::}\mathit{xs};\ \mathsf{pc} = \mathit{ptr}\}}$$

$$\frac{\mathit{fetch}\ \mathit{bs} = \mathsf{CallPtr} \quad \mathit{bs}.\mathsf{stack} = x{::}\mathsf{CodePtr}\ \mathit{ptr}{::}\mathit{xs}}{\mathit{bs} \to \mathit{bs}\{\mathsf{stack} = x{::}\mathsf{CodePtr}\ (\mathsf{bump}\ \mathit{bs}).\mathsf{pc}{::}\mathit{xs};\ \mathsf{pc} = \mathit{ptr}\}}$$

$$\frac{\mathit{fetch}\ \mathit{bs} = \mathsf{PushExc} \quad \mathit{bs}.\mathsf{stack} = \mathit{xs}}{\mathit{bs} \to (\mathsf{bump}\ \mathit{bs})\{\mathsf{stack} = \mathsf{StackPtr}\ \mathit{bs}.\mathsf{handler}{::}\mathit{xs};\ \mathsf{handler} = \mathsf{length}\ \mathit{xs}\}}$$

$$\frac{\mathit{fetch}\ \mathit{bs} = \mathsf{PopExc} \quad \mathit{bs}.\mathsf{handler} = \mathsf{length}\ \mathit{ys} \quad \mathit{bs}.\mathsf{stack} = x{::}\mathit{xs}\ @\ \mathsf{StackPtr}\ h{::}\mathit{ys}}{\mathit{bs} \to (\mathsf{bump}\ \mathit{bs})\{\mathsf{stack} = x{::}\mathit{ys};\ \mathsf{handler} = h\}}$$

**Fig. 4** Examples of semantics of CakeML bytecode instructions. The helper function *fetch bs* fetches the next instruction according to the program counter *bs*.pc, and bump *bs* updates the program counter to the next instruction.

Since $bs_1 \to bs_2$ (and hence $bs_1 \to^* bs_2$) is deterministic, we can define a function in the logic, EvalBC $bs_1$, that returns the result of repeatedly stepping the semantics until no further step is possible, which occurs when there is no applicable rule for the next instruction either because the machine was mis-configured or the next instruction is Stop. If bytecode evaluation of $bs_1$ eventually stops, then EvalBC $bs_1$ = Some $bs_2$ for the unique final state $bs_2$. If, however, there is no final state and evaluation of $bs_1$ diverges, then EvalBC $bs_1$ = None.

Data refinement from CakeML source-level values to bytecode values must encode all CakeML values as bytecode Blocks and Numbers. The overall refinement relation decomposes into a series of relations that mirror each phase of compilation. The most complicated part of data refinement is for closures; at a high level, our strategy encodes each closure as Block closure_tag (CodePtr *ptr*::*env*), where the code pointer *ptr* points to the result of compiling the body of the closure, which must exist in the bytecode machine state's code field. For first-order values, since bytecode blocks are structured and bytecode integers are mathematical integers, data refinement is not much more complicated than assigning tags to blocks to distinguish different types of value and following a straightforward encoding scheme. Data refinement to bytecode is part of the Inv *env cs bs* invariant seen previously asserting that the semantics, the compiler, and the bytecode machine state are in correspondence.

Now we have a basic understanding of the target language for ReplFun, the middle layer of Figure 2. To connect to the bottom layer, where the final target is x86-64 machine code, we write a simple compiler from bytecode as described in Section 9. To connect from the top layer, let us now see how the verified compiler from CakeML source to bytecode enables us to prove that ReplFun implements ReplSem.

## 7 REPL implementation specified as a function in logic

When we looked at the example of bootstrapping CompileDec, we evaluated CompileDec on syntax implementing CompileDec itself. When bootstrapping for the REPL we will still evaluate CompileDec but on syntax implementing a larger function. We combine parsing, type inference, and compilation to bytecode together as:

$$\mathsf{ParseInferCompile}\ \mathit{tokens}\ s$$

which is called on a list of tokens, *tokens*, produced by the lexer and the state, *s*, of the REPL implementation[8]. This function returns either Failure $(\mathit{msg},\mathit{sf})$ if there is a parse- or type-error, or Success $(\mathit{code},\mathit{ss},\mathit{sf})$ with bytecode *code* that executes the declaration represented by *tokens* and new REPL states to be installed if running *code* terminates normally $(\mathit{ss})$ or raises an exception $(\mathit{sf})$[9]. In addition to the parser and type inferencer, we also include an initial program—the CakeML Basis Library—to be loaded in the REPL when it starts. The combined function representing almost[10] all the code to be bootstrapped is ReplStep, and is defined at the top of Figure 5. Bootstrapping affords us the ability to produce low-level implementations of the parser and type inferencer automatically after verifying their shallow embeddings: we simply include them (via ParseInferCompile) in this function to be bootstrapped.

The remainder of the REPL (the lexer, the main loop, and the runtime that executes the compiler-generated code) is not generated by bootstrapping, so requires a more manual treatment. However, we specify the entire REPL implementation, including the non-bootstrapped parts, as a function in the logic. That function is ReplFun, and its definition is shown in Figure 5. The majority of the code in the REPL implementation is hidden inside the ParseInferCompile algorithm inside ReplStep, but since this part is produced by bootstrapping we only need to know that the algorithm is correct and not how it is implemented. By comparison, the details of the implementation of MainLoop are important for constructing the final machine-code implementation, but there are only a handful of them.

The correctness theorem for ReplFun states that it produces exactly the same *repl_result*, *output*, for a given *input* as is specified by the semantics ReplSem (modulo an additional empty result at the front corresponding to the basis library).

**Theorem 1 (Correctness of ReplFun)**

$$\begin{array}{l}\vdash \forall \mathit{input}.\\ \quad \exists \mathit{output}.\ \mathsf{ReplFun}\ \mathit{input} = \mathsf{Result}\ \texttt{""}\ \mathit{output} \wedge \mathsf{ReplSem}\ \mathsf{Basis}\ \mathit{input}\ \mathit{output}\end{array}$$

Theorem 1 is proved by complete induction on the length of the input string (which corresponds to the number of declarations made by the user), and follows the model of invariant preservation. The invariant used for the REPL extends the Inv invariant from Section 2.3 with information about type inference. It connects the semantics, the compiler, and the bytecode, ensuring that: the state of the type system in the semantics is consistent with itself and with the state of the inferencer, and the state of the operational semantics is consistent with the state of both the compiler and the values in the bytecode machine. In each iteration of MainLoop, we combine the correctness theorems for the parser, type inferencer, and compiler to conclude that either the correct error message is produced or the generated bytecode, when evaluated, correctly diverges or correctly stops in a bytecode machine state that again satisfies the invariant.

ReplFun implements ReplSem, so we have reduced our task to implementing ReplFun in machine code. The function divides neatly into two parts, the part called ReplStep, and the

[8] The state includes the compiler's and type inferencer's memory of previous declarations, whose results may be referred to in later declarations.

[9] Different states are required since not all bindings might persist if an exception is raised, and exceptions are not statically predictable.

[10] All that is missing is an extra interface function used to make a call to ReplStep, described in Section 8.

```
ReplStep None = Success BasisCodeAndStates
ReplStep (Some (tokens,s)) = ParseInferCompile tokens s

MainLoop prev bs input =
 case ReplStep prev of
  Success (code,ss,sf) ⇒
   (let bs₁ = AddCode bs code
    in
     case EvalBC bs₁ of
      None ⇒ Diverge
     | Some bs₂ ⇒
       Result bs₂.output
        (case LexUntilSemicolon input of
          None ⇒ Terminate
         | Some (tokens,input₂) ⇒
           (let s₂ = TestException bs₂ (ss,sf)
            in
              MainLoop (Some (tokens,s₂)) bs₂ input₂)))
 | Failure (msg,sf) ⇒
   Result msg
    (case LexUntilSemicolon input of
      None ⇒ Terminate
     | Some (tokens,input₂) ⇒ MainLoop (Some (tokens,sf)) bs input₂)

ReplFun input = MainLoop None EmptyBS input
```

**Fig. 5** REPL implementation specified as a function in the logic, ReplFun, which is partitioned into a part to be bootstrapped (ReplStep) that includes the parser, type inferencer, compiler, and initial program, and a part to be verified using decompilation (the rest of MainLoop).
The particular functions that need the manual decompilation treatment can be seen in the definition of MainLoop, they are: AddCode to install new code in the code heap, EvalBC that simulates bytecode execution, LexUntilSemicolon that reads and lexes new input, and TestException that checks whether bytecode simulation ended with success or failure and returns the corresponding new REPL state.
The main loop takes the last read declaration (*tokens*) and current state (*s*) as an argument, *prev*, so that the first thing it does is call the ReplStep function: this way of structuring the loop makes it easier to include the bootstrapped code in the final machine-code implementation.

part called MainLoop that does case-analysis on the result of ReplStep. To produce machine code for ReplStep, we put it through the proof-grounded bootstrapping process described in Section 3. In the next section, we look at bootstrapping ReplStep more carefully and address the question of using the bootstrapped compiler at package runtime, that is, providing it input and retrieving its output. Then, in Section 9 we turn to verifying machine code for the rest of ReplFun and putting the two together.

## 8 Bootstrapping a function call

To bootstrap ReplStep, we follow the strategy described in Section 3, where we bootstrapped CompileDec by evaluating compilation of CompileDec_decs. Which declarations should we use in place of CompileDec_decs? To answer this, consider how we will use the result of bootstrapping which, analogous to Lemma 5, produces a bytecode machine state containing the declared values. Since our main loop makes a call to ReplStep, we want those values to include ReplStep, but we also want to be able to call ReplStep on input and obtain its output. To make the interface between the bootstrapped and non-bootstrapped code as simple as possible, we define one extra function, CallReplStep, that calls ReplStep and does I/O via CakeML references. Thus, the declarations we want to bootstrap, called REPL_decs, are:

```
...; fun replstep x = ...;
val input = ref NONE;
val output = ref NONE;
fun callreplstep _ =  output := (replstep (!input));
```

The first line represents 428 declarations (for the parser, type inferencer, compiler, and all dependencies) generated automatically by proof-producing translation of ReplStep, and the last three declarations are added by hand.

The important feature of CallReplStep is that its type in CakeML is *unit* $\rightarrow$ *unit*, which means it can be called multiple times uniformly. The certificate theorem for CallReplStep will be used in the same way each time around the loop of the REPL. We use references for I/O so we do not have to reason about an endless sequence of calls to CallReplStep, but instead prove a single theorem (Theorem 2 below) that is strong enough to apply to each call.

To call ReplStep, the non-bootstrapped machine code need only do three things: update the `"input"` reference, run the following declaration, called call_dec:

```
val () = REPL.callreplstep ()
```

and read the `"output"` reference. We now have two declarations serving different roles. The first is REPL_decs, which is used to declare CallReplStep and all its dependencies (including the compiler). The second is call_dec which does not declare anything (it returns *unit*), but has the side-effect of calling CallReplStep and updating the I/O references. We apply bootstrapping to both declarations, because we need verified low-level implementations of both. The first step is to produce certificate theorems.

Most of the syntax for REPL_decs is generated by proof-producing translation of ReplStep, which generates certificate theorems automatically. We use them to prove some extended certificate theorems that mention the I/O-related declarations we added. Our extended certificate theorems, shown below, say that the semantics of REPL_decs is to produce an environment, called ReplEnv, and whenever the call_dec declaration is made in ReplEnv, it has the sole effect of updating the `"output"` reference with the result of applying ReplStep to the contents of the `"input"` reference.

**Theorem 2 (Certificate theorems for REPL_decs and call_dec)**

$$
\begin{array}{l}
\vdash \mathsf{EvalDec}\ \mathsf{InitEnv}\ (\mathsf{Struct}\ \texttt{"REPL"}\ \mathsf{REPL\_decs})\ \mathsf{ReplEnv} \\
\vdash \forall x\ inp\ out_1 .\\
\quad \mathsf{InpTy}\ x\ inp \Rightarrow \\
\quad\ \exists out_2 .\\
\quad\quad \mathsf{OutTy}\ (\mathsf{ReplStep}\ x)\ out_2 \land \\
\quad\quad \mathsf{EvalDec}\ (\mathsf{UpdRefs}\ inp\ out_1\ \mathsf{ReplEnv})\ \mathsf{call\_dec}\ (\mathsf{UpdRefs}\ inp\ out_2\ \mathsf{ReplEnv})
\end{array}
$$

As usual, there are refinement invariants (in this case InpTy and OutTy) mediating the connection between HOL values ($x$ and ReplStep $x$) and CakeML values (*inp* and $out_2$). The helper function above, UpdRefs *inp out* ReplEnv, denotes an instance of ReplEnv where nothing has changed except for the contents of the two references which are now *inp* and *out*.

Now for the bootstrapping theorems. We use the same compiler as before (CompileDec), and apply evaluation in the logic to our two declarations to obtain bytecode (REPL_bytecode and Call_bytecode) that implements them. The compiler needs to know how to compile the variable lookup for `"REPL.callreplstep"` when compiling call_dec, so we use the compiler state (ReplCS) that resulted from compiling the REPL declarations when compiling call_dec.

**Theorem 3 (Bootstrapping theorems for the REPL)**

$$\begin{array}{l}\vdash \mathsf{CompileDec\ InitCS}\ (\mathsf{Struct}\ \texttt{"REPL"}\ \mathsf{REPL\_decs}) = (\mathsf{ReplCS}, \mathsf{REPL\_bytecode}) \\ \vdash \mathsf{CompileDec\ ReplCS\ call\_dec} = (\mathsf{CallCS}, \mathsf{Call\_bytecode})\end{array}$$

Let us review the three theorems used for bootstrapping, and what results from following the method.

- Correctness theorem: since we are still using CompileDec as our compilation algorithm, we continue to use its correctness theorem, Lemma 1.
- Certificate theorem: Theorem 2 states that the semantics of call_dec is to make a call to ReplStep via I/O references.
- Bootstrapping theorem: Theorem 3 contains the bootstrapped bytecode, REPL_bytecode and Call_bytecode, that comes from evaluating the compiler.

Instantiate the correctness theorem with the bootstrapping theorem, then apply the certificate theorem. For REPL_decs, we get a result stating that we can produce a bytecode machine state, ReplBS, implementing ReplEnv.

**Theorem 4 (Result of bootstrapping REPL_decs)**

$$\begin{array}{l}\vdash (\mathsf{AddCode\ InitBS\ REPL\_bytecode}) \to^* \mathsf{ReplBS} \land \mathsf{Halted\ ReplBS} \land \\ \quad \mathsf{Inv\ ReplEnv\ ReplCS\ ReplBS}\end{array}$$

The first thing the non-bootstrapped machine code for the REPL does is load REPL_bytecode into InitBS and run it. By Theorem 4, this produces the ReplBS bytecode machine state, which will be used for all subsequent calls to ReplStep. The invariant governing these calls to ReplStep is a specialised version of the Inv invariant, which fixes everything except the I/O references, so that it can be re-established after each call. Specifically, the specialised invariant is InvIO *inp out bs*. This invariant means that *bs* is AddCode ReplBS Call_bytecode modulo I/O references, and Inv (UpdRefs *inp out* ReplEnv) ReplCS holds for *bs* before Call_bytecode is added.

If we write the result of bootstrapping the call using this InvIO invariant, it is clear that if the non-bootstrapped code sets the input reference correctly, it can run Call_bytecode after which the output reference will be set to the result of calling ReplStep. The function ResetPC *bs* sets the program counter back to the beginning of Call_bytecode, in preparation for the next iteration of the REPL.[11]

**Theorem 5 (Result of bootstrapping call_dec)**

$$\begin{array}{l}\vdash \mathsf{InvIO}\ inp\ out_1\ bs_1 \land \mathsf{InpTy}\ x\ inp \Rightarrow \\ \quad \exists out_2\ bs_2. \\ \quad\ \mathsf{OutTy}\ (\mathsf{ReplStep}\ x)\ out_2 \land bs_1 \to^* bs_2 \land \mathsf{Halted}\ bs_2 \land \\ \quad\ \mathsf{InvIO}\ inp\ out_2\ (\mathsf{ResetPC}\ bs_2)\end{array}$$

Theorem 5 lets the non-bootstrapped part of the REPL implementation call the bootstrapped compiler. This result is simply the bytecode-level version of Theorem 2 (the certificate theorem describing this process at the level of the operational semantics). Together, Theorems 4 and 5 represent the results of bootstrapping for the REPL. We turn now to the non-bootstrapped parts of the REPL, and putting the whole package together.

[11] The conclusion of Theorem 5 is not used immediately again as its hypothesis. First the *inp* parameter is changed by the non-bootstrapped code.

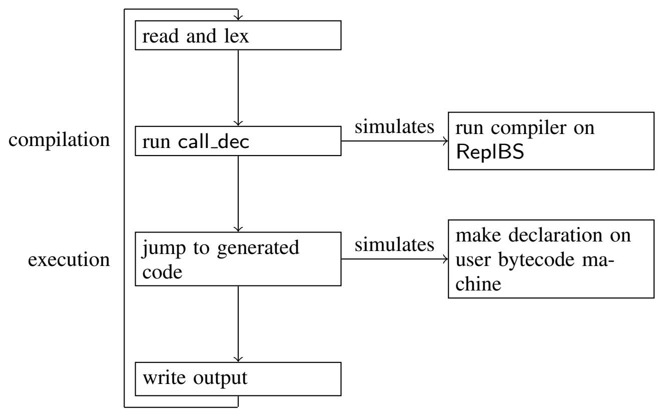

**Fig. 6** The two bytecode machines simulated by the final REPL implementation. The left half of the figure is specified by MainLoop (Figure 5). The call to EvalBC within MainLoop happens via simulation of the main bytecode machine (for user code). The call to ReplStep within MainLoop happens via simulation of another bytecode machine, which stays in ReplBS (modulo references), and runs the bootstrapped compiler.

## 9 Producing verified machine code

The machine-code implementation of the REPL for CakeML does the following steps in a loop: read and lex the next declaration (LexUntilSemicolon), compile the declaration to bytecode (ReplStep), evaluate the compiled bytecode (EvalBC) by first compiling to x86-64 then jumping to the new code, print the result and continue. These steps can be seen in the specification of the REPL main loop in Figure 5. The most involved part of each iteration is compilation to bytecode, but we have verified bootstrapped code for ReplStep to do that part. The next most complicated part is compilation and evaluation of bytecode (EvalBC).

Because it includes bootstrapped code, the final REPL implementation depends on the two separate sessions of the REPL semantics, and simulates two bytecode machines. The first session, for the compiler, is the one that is initialised with REPL_decs and thereafter stays in ReplBS (with input/output references updated each iteration). The second session, for the user, is the one that runs the user's input on the bytecode machine state *bs* passed around in the definition of MainLoop. Figure 6 illustrates how these fit together.

To simulate each bytecode machine, we write a simple compiler from bytecode instructions (Section 6) to snippets of x86-64 machine code. For the semantics of x86-64 machine code, we use the model developed by Sarkar et. al. [38] and updated for the verified Lisp runtime, Jitawa [31]. We verify the compiler using the technique of decompilation into logic [34]. The most difficult part of this verification is devising the invariant that holds between a bytecode machine state and an x86-64 machine state, which also includes data refinement from bytecode values to immediate values or pointers into the x86-64 heap. We do not delve into the details of this invariant here, since they are not especially relevant to packaging bootstrapped code.

There are a handful of bytecode instructions (e.g., structural equality) and helpers (e.g., lexing) that are implemented by machine-code routines that are larger than the snippets used for most instructions. Also, instructions which do allocation or arithmetic make use of separately verified machine-code routines for garbage collection [29] and arbitrary-precision integer arithmetic [30]. In each case, the larger routine is verified using decompilation and plugged into the overall correctness proof. Producing the non-bootstrapped parts of

MainLoop, including the runtime for simulating bytecode execution, is an example of machine-code verification as used in previous work [31] verifying the Jitawa runtime for Lisp.

To use the bootstrapped code, it is sufficient to establish the InvIO invariant, since we can then apply Theorem 5. We prove that the InvIO invariant holds after REPL_bytecode runs when the REPL starts, and then continues to hold when the input reference is updated with the result of lexing. Theorem 5 lets us preserve the invariant across calls to the compiler, and therefore throughout execution of the main loop.

Our interface to the x86-64 machine semantics is via predicates that apply to sequences of steps (traces) made by the x86-64 state machine. The kinds of predicates we use are inspired by temporal logic. The assertion TemporalX64 *code A* states that if *code* is loaded in memory then the temporal predicate *A* is satisfied by all runs of the machine. Satisfaction of a temporal predicate by a run, *s*, is defined as follows:

- Now $P$ is satisfied by $s$ if $P\ (s\ 0)$.
- Holds $p$ is satisfied by $s$ if $p$ is true. ($p$ does not depend on the machine state).
- $\Diamond A$ is satisfied by $s$ if $A$ is satisfied by $\lambda n.\ s\ (n+k)$ for some $k$.
- $\Box A$ is satisfied by $s$ if $A$ is satisfied by $\lambda n.\ s\ (n+k)$ for all $k$.
- $A \wedge B$ is satisfied by $s$ if $A$ is satisfied by $s$ and $B$ is satisfied by $s$. Similarly for $A \vee B$, $A \Rightarrow B$, and $\exists x.\ A\ x$.

The final correctness theorem we obtain is about a single machine-code program (a list of bytes), which we abbreviate as ReplX64, and is phrased as a temporal assertion about running that program. It states that: if at some time the machine state is appropriately initialised, then either it will eventually run out of memory, or it will eventually diverge or terminate with output according to the CakeML REPL semantics.

**Theorem 6 (Correctness of REPL implementation in x86-64)**

$$\begin{array}{l}
\vdash \mathsf{TemporalX64}\ \mathsf{ReplX64} \\
\quad (\mathsf{Now}\ (\mathsf{InitialisedX64}\ ms) \Rightarrow \\
\quad\ \ \Diamond\ \mathsf{Now}\ (\mathsf{OutOfMemX64}\ ms) \vee \\
\quad\ \ \exists output. \\
\quad\ \ \ \ \mathsf{Holds}\ (\mathsf{ReplSem}\ \mathsf{Basis}\ ms.\mathsf{input}\ output) \wedge \\
\quad\ \ \ \ \mathtt{if}\ \mathsf{Diverges}\ output\ \mathtt{then}\ \Box\ \Diamond\ \mathsf{Now}\ (\mathsf{RunningX64}\ output\ ms) \\
\quad\ \ \ \ \mathtt{else}\ \Diamond\ \mathsf{Now}\ (\mathsf{TerminatedX64}\ output\ ms))
\end{array}$$

The helper function Diverges *repl_result* tests whether *repl_result* ends in termination or divergence (the *repl_result* type is described in Section 5). There are four predicates on machine states *ms* that encode our invariants and conventions concerning the x86-64 machine as it simulates a bytecode machine.

- InitialisedX64 *ms* states that the machine is initialised. The heap invariant is satisfied, there is a return pointer on the stack, and the machine's output stream is empty.
- OutOfMemX64 *ms* states that the machine has aborted execution and is out of memory.
- RunningX64 *output ms* states that the heap invariant is satisfied and the output stream is equal to the concatenation of results in *output*.
- TerminatedX64 *output ms* states that the machine is about to jump to the return pointer and the output stream is equal to the concatenation of results in *output*.

Theorem 6 thus connects execution of an x86-64 machine loaded with the verified code produced by bootstrapping and decompilation back to the CakeML REPL semantics, completing the picture shown in Figure 2.

## 10 Conclusion

Unverified compilers usually contain bugs [44]. To reduce dependence on unverified tools, we suggest bootstrapping a verified compiler—compiling it with itself—in a proof-grounded way so that the correctness theorem applies to the final implementation that runs without further compilation. The proof-grounded bootstrapping method is mostly automatic. It uses proof automation techniques (proof-producing translation from shallow to deep embeddings, and evaluation in the logic) to push hard-won results about the correctness of compilation algorithms down to the level of real implementations.

The theorems that result from proof-grounded bootstrapping let us package a verified compiler implementation inside a larger machine-code program and prove a correctness theorem about the combined system. We used bootstrapping to eliminate compilation from the trusted computing base (TCB) of a read-eval-print loop (REPL) for CakeML, a machine-code program that contains the verified compiler for CakeML and calls it repeatedly at runtime.

### 10.1 Trusted computing base

What is in the trusted computing base for the CakeML REPL? The correctness theorem for the final implementation, Theorem 6, is written in terms of the semantics of x86-64. It has an assumption that the x86-64 machine starts in a correctly initialised state, and concludes that its behaviour implements the semantics of the CakeML REPL. To run the REPL implementation, we need to create the initial state, then we simply run the verified machine code. The TCB, therefore, consists of three things:

1. Verification: the software that checked the proof of Theorem 6, and our method for extracting the verified code, ReplX64, from the theorem statement.
2. Initialisation: the code used to create an initial machine state that satisfies the assumption of Theorem 6.
3. Execution: the hardware and operating system that runs the verified implementation. Our x86-64 semantics needs to capture the execution environment accurately.

What have we removed from the TCB by bootstrapping and packaging the compiler? Without bootstrapping, there would have been an additional item, after initialisation, about compilation from a verified algorithm to an executable, and the execution item would additionally include the language runtime. Without packaging, if we had merely verified a standalone compiler, there would have been additional initialisation and execution steps for running the output of the compiler. Thus, we have succeeded in removing trust in the compiler and runtime for running CakeML applications.

Now let us look more closely at what is left in the TCB, starting at the bottom with the execution environment. The x86-64 semantics we use is naive in two ways:

1. The semantics only covers user-mode instructions, and only a subset of them. This is particularly important for I/O: we simply assume it is possible to make system calls to read and write characters.
2. The semantics has a flat view of memory. We do not model virtual memory.

Trust in hardware is unavoidable, but it can be reduced with more accurate models. The hardware model can be made more accurate independently of the bootstrapping technique, which sits above it.

What we require of the operating system (if any) and memory subsystem is transparency: we leave them out of formal assumptions and thereby trust them to keep up the illusion of running on the hardware directly and without virtual memory. These items, together with initialisation code, represent realistic opportunities for more accurate modelling.

The initialisation code represents work traditionally done by a boot loader, or by a linker and loader. In theory, we could produce a boot loader to initialise a machine with the CakeML REPL implementation, which would then run “on bare metal”. In practice, we write our initialisation code in a small (30 SLOC) C wrapper program, which includes the CakeML REPL machine code as inline assembly. We compile this C program with standard (unverified) tools. In this setup, the initialisation part of the TCB includes the C compiler and linker, and the operating system’s loader. While we have theoretically avoided trusting a C compiler, we would need to formalise and verify linking and loading to produce a practical alternative to using a C compiler for initialisation.

Finally, we trust the theorem prover, and its execution environment (compiler, runtime, etc.), that we use to produce our verified implementation and to check its correctness theorem. Trust in the theorem prover is a methodological hazard of formal verification. However, it is not as bad as it sounds, because the real products of verification are *proofs* that can be checked independently. We must trust a theorem prover, but we are not constrained to a single one.

The question is again of practicality. In theory, we can export our proofs from HOL4 using OpenTheory [14] or similar technology, to be checked independently by OpenTheory itself or another theorem prover such as HOL Light. Such proofs are the sequences of primitive inferences that pass through the LCF-style kernel of HOL4. In practice, the proofs generated by automation like evaluation in the logic and translation from shallow to deep are extremely large and would require improved infrastructure to export. Possible directions for making independent checking more practical include compressing proofs as or after they are exported, or exporting proofs at intermediate levels rather than expanding everything out as primitive inferences. The latter demands greater sophistication from the independent checker.

Proofs about realistic software are too large to be checked by hand. Can the required machine assistance itself be verified? In other work [20,32], we have considered verifying implementations (in machine code) of theorem provers and proving that they only produce theorems that are true according to the semantics of their logic. To avoid an infinite regress of trust, one might consider self-verification of a theorem prover, that is, a theorem prover that can verify its own implementation. There are obvious parallels to compiler bootstrapping, but although such a self-verifying prover would be an interesting and impressive achievement it does not eliminate our need to trust something altogether. There always remains the possibility that a self-verifying theorem prover is unsound in a way that causes it to incorrectly verify itself.

Ken Thompson, in his Turing award lecture, Reflections on Trusting Trust [42], describes a method by which a Trojan horse—deliberate mis-compilation of certain programs—can be inserted into a bootstrapped compiler while leaving no trace in the compiler’s source code. The trick is to make the compiler introduce the Trojan horse, and code for introducing the Trojan horse, whenever it recognises that it has been given its own source code as input. Can a similar trick be used to introduce a Trojan horse into a compiler produced by proof-grounded bootstrapping? The crux of Thompson’s example is that the compiler executable used to re-compile the compiler is already contaminated; in proof-grounded bootstrapping we do not use a compiler executable to compile the compiler, rather, we use evaluation in the logic of the theorem prover. Thus, to insert a Trojan horse we would need to contaminate the theorem prover to recognise when it is being asked to evaluate our compiler in the logic, or, perhaps simpler, when it is being asked to export the result of bootstrapping at which point it could substitute malicious machine code instead. Thompson's example simply reinforces the need to trust the verification tools we use, and is mitigated as explained above by independent checking of proofs.

### 10.2 Related work

*Verified compilers* CompCert [23] is foremost amongst verified realistic compilers, being a compiler for C that includes verified optimisations and is deployed in the real world. Recent improvements to CompCert include validated parsing [16] and there are versions with the ability to do (verified) separate compilation [40]. CompCert is an algorithm that is verified in Coq, and the implementation of the compiler is extracted from Coq as an OCaml program before it is compiled and run. The correctness theorem covers the compilation algorithm for compiling whole programs down to assembly code. The trusted computing base for running the compiler includes the OCaml compiler and runtime and other build tools.

Unfortunately, it is not possible to immediately apply proof-grounded bootstrapping to CompCert to obtain a correctness theorem about its implementation. The reason is that the source and implementation languages of the compiler are very different, so it does not satisfy the second prerequisite for bootstrapping: that the compiler is written in its own source language. It may be possible to create a kind of proof-producing translation from Coq to C to fill this gap and enable proof-grounded bootstrapping of CompCert, but such a tool would be bridging a much larger gap (from Coq to C) than our proof-producing translator from HOL to CakeML does.

Considering verified compilers for higher-order functional languages, the Lambda Tamer project [8] precedes our work on CakeML. Lambda Tamer includes a compilation algorithm, verified in Coq, from an ML-like language to an idealised assembly language, and emphasises clever choices of representations for formalisation that lead to highly automatic proofs. The definition of the compiler uses dependent types, which are not present in its source language. Therefore, to bootstrap this compiler one would need a more sophisticated proof-producing translator that can translate away dependent types.

Another example of verified compilation for a high-level language came out of the VerifiCard project [4], aimed at formalising a subset of Java as used on smartcards. This work predates CompCert, and highlighted the approach of generating executable code from a verified algorithm by using the "code extraction" facility of the theorem prover (in their case, Isabelle/HOL). While it provides a convenient route for producing executable code, which can further be integrated with code developed separately, this approach requires trust in the code extraction facility and in the compiler used on the extracted output. By contrast, with proof-grounded bootstrapping one need only trust "extraction" (really just printing) of machine code, where preservation of semantics is a simpler claim.

Proof-grounded bootstrapping is a technique that promotes end-to-end verification: correctness from source code to machine code within a single theorem prover. On this theme, there have been several impressive projects, perhaps the first of which was the "verified stack" of Computational Logic, Inc. [6,26], which in the late 1980's produced a verified stack from applications down to hardware (i.e., below the machine code that has been our lowest level). Moore [27] writes that this project was very ambitious for its time, and the results fall short on realism and usefulness; however, he also notes that "the CLI stack was a *technology driver*", and indeed the now industrial-strength theorem prover, ACL2 [17], was one of its products.

More recently, Chlipala's Bedrock [9] framework emphasises building end-to-end proofs from high level languages down to assembly code using modular interfaces, and comes with a great deal of proof automation. The specifications, proofs, and automation are all implemented in the Coq theorem prover. A recent example of the use of Bedrock covers end-to-end verification of web applications [10]. Gu et. al. [13] describe another modularity-focused approach to end-to-end correctness, also in Coq; their main application example is an operating-system kernel. The Verified Software Toolchain [1,2] is also geared towards end-to-end correctness. The particular approach is to build a program logic, in Coq, above the subset of C accepted by CompCert, enabling verification of source-level C programs that can then be compiled by CompCert. Since the program logic, Verifiable C, is proved sound with respect to CompCert C's semantics, properties proved about the source programs carry down to the generated assembly code.

The closest work to the CakeML compiler and theorem prover is the verified Milawa theorem prover that runs on a verified Lisp compiler on the verified Jitawa runtime [31]. As with CakeML, the correctness theorems for Milawa and Jitawa are about implementations in machine code. However, the Lisp compiler used in Jitawa is not bootstrapped; rather, the whole compiler is verified using the decompilation techniques that in CakeML were used only for smaller libraries (garbage collector, lexer, etc.). The machine-code verification techniques (decompilation in particular) used in verifying the implementation of CakeML bytecode were also used recently in binary validation of the seL4 verified operating system microkernel [19]. Other work on machine code verification includes Jensen, Benton, and Kennedy's [15,18] work that makes heavy use of dependent types for modelling machine code, and develops a higher-order separation logic above the model which is in some ways similar to our machine-code Hoare logic [28] used in decompilation.

*Verified bootstrapping* Bootstrapping of verified compilers is less common in the literature than verified compilation in general. However, it is not without precedent.

An early reference can be found in work on the Verifix project (e.g., Goerigk and Hoffman [12]), which describes a bootstrapping process that is closer to traditional compiler bootstrapping than the proof-grounded bootstrapping method described in this thesis: the bootstrapping is used to introduce implementations of new language features into the compiler and thus there are many phases of bootstrapping. Goerigk and Hoffman's approach includes manual review of the output of bootstrapping, and considers trusting execution inside the theorem prover risky. We take the opposite view, and consider execution of the compiler inside the logic a much more trustworthy process than execution outside the logic followed by manual review. The Verifix view may have been influenced by the state of the art of theorem provers at the time, which may not have supported efficient execution that nevertheless produces theorems checked by a small kernel.

More recently, Strub et. al. [41] have proposed self-certification of type checkers, and demonstrated it with a bootstrapping type checker for F* (itself implemented in Coq). The idea is to write a type checker in the type system whose specifications it checks, and to give the type checker the specification that it correctly implements the type system. Then, run the type checker on itself to produce a certificate and check that certificate in the theorem prover. The point is to only check a single application of the type checker (application to itself) in the theorem prover and thereafter use the verified typechecker without need of the theorem prover. This is analogous to proof-grounded bootstrapping (of a compiler): we execute the compiler once (on itself) in the logic and thereafter can run the verified machine-code implementation outside of the theorem prover.

### 10.3 Future work

As mentioned in Section 10.1, a promising line for future work is to integrate the correctness theorem for a packaged compiler implementation with verified tools for linking and loading. Such an integration would let us replace the initialisation code in the trusted computing base with a formal semantic model of linking and loading, and our compiler would produce an executable (e.g., an ELF) verified against this semantics rather than raw machine code. Similarly, we are considering what it would take to run the CakeML REPL as a verified user application on the seL4 verified operating system [19] and thereby remove the operating system from the execution part of the TCB.

Because CakeML does not support I/O primitives directly, we had to resort in Section 8 to tricks using references to give the REPL I/O at the top level. More seriously, to mix bootstrapped and non-bootstrapped code we had to use the subtle method of simulating two different bytecode machine states. We are currently investigating a more straightforward approach to producing a packaged compiler that adds both I/O and dynamic installation of new code as primitives to the source language, thereby allowing a REPL implementation to be written entirely in the source language.

## References

1. Andrew W. Appel. Verified software toolchain - (invited talk). In Gilles Barthe, editor, *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, volume 6602 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2011.
2. Andrew W. Appel. *Program Logics for Certified Compilers*. Cambridge University Press, 5th ed. edition, 2014.
3. Bruno Barras. Programming and computing in HOL. In Mark Aagaard and John Harrison, editors, *TPHOLs*, volume 1869 of *Lecture Notes in Computer Science*, pages 17–37. Springer, 2000.
4. Stefan Berghofer and Martin Strecker. Extracting a formally verified, fully executable compiler from a proof assistant. *Electr. Notes Theor. Comput. Sci.*, 82(2):377–394, 2003.
5. Yves Bertot. A short presentation of Coq. In Mohamed et al. [25], pages 12–16.
6. William R. Bevier, Warren A. Hunt Jr., J Strother Moore, and William D. Young. An approach to systems verification. *J. Autom. Reasoning*, 5(4):411–428, 1989.
7. Richard J. Boulton, Andrew D. Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In Victoria Stavridou, Thomas F. Melham, and Raymond T. Boute, editors, *Theorem Provers in Circuit Design, Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience, Nijmegen, The Netherlands, 22-24 June 1992, Proceedings*, volume A-10 of *IFIP Transactions*, pages 129–156. North-Holland, 1992.
8. Adam Chlipala. A verified compiler for an impure functional language. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 93–106. ACM, 2010.
9. Adam Chlipala. The bedrock structured programming system: combining generative metaprogramming and hoare logic in an extensible program verifier. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 391–402. ACM, 2013.
10. Adam Chlipala. From network interface to multithreaded web applications: A case study in modular program verification. In Rajamani and Walker [37], pages 609–622.

11. Willem P. de Roever and Kai Engelhardt. *Data Refinement: Model-oriented Proof Theories and their Comparison*, volume 46 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.
12. Wolfgang Goerigk and Ulrich Hoffmann. Rigorous compiler implementation correctness: How to prove the real thing correct. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullmann, editors, *Applied Formal Methods - FM-Trends 98, International Workshop on Current Trends in Applied Formal Method, Boppard, Germany, October 7-9, 1998, Proceedings*, volume 1641 of *Lecture Notes in Computer Science*, pages 122–136. Springer, 1998.
13. Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In Rajamani and Walker [37], pages 595–608.
14. Joe Hurd. The OpenTheory standard theory library. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2011.
15. Jonas Braband Jensen, Nick Benton, and Andrew Kennedy. High-level separation logic for low-level code. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 301–314. ACM, 2013.
16. Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. Validating LR(1) parsers. In Helmut Seidl, editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 397–416. Springer, 2012.
17. Matt Kaufmann and J Strother Moore. An ACL2 tutorial. In Mohamed et al. [25], pages 17–21.
18. Andrew Kennedy, Nick Benton, Jonas Braband Jensen, and Pierre-Évariste Dagand. Coq: the world's best macro assembler? In Ricardo Peña and Tom Schrijvers, editors, *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*, pages 13–24. ACM, 2013.
19. Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1):2, 2014.
20. Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. Self-formalisation of higher-order logic. *J. Autom. Reasoning*, 2015. To appear.
21. Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *POPL*, pages 179–192. ACM, 2014.
22. Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
23. Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4):363–446, 2009.
24. Robin Milner. LCF: A way of doing proofs with a machine. In Jirí Becvár, editor, *Mathematical Foundations of Computer Science 1979, Proceedings, 8th Symposium, Olomouc, Czechoslovakia, September 3-7, 1979*, volume 74 of *Lecture Notes in Computer Science*, pages 146–159. Springer, 1979.
25. Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors. *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*. Springer, 2008.
26. J Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5:461–492, 1989.
27. J Strother Moore. A grand challenge proposal for formal methods: A verified stack. In Bernhard K. Aichernig and T. S. E. Maibaum, editors, *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *Lecture Notes in Computer Science*, pages 161–172. Springer, 2002.
28. Magnus O. Myreen. *Formal verification of machine-code programs*. PhD thesis, University of Cambridge, 2008.
29. Magnus O. Myreen. Reusable verification of a copying collector. In Gary T. Leavens, Peter W. O'Hearn, and Sriram K. Rajamani, editors, *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings*, volume 6217 of *Lecture Notes in Computer Science*, pages 142–156. Springer, 2010.
30. Magnus O. Myreen and Gregorio Curello. Proof pearl: A verified bignum implementation in x86-64 machine code. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, volume 8307 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 2013.
31. Magnus O. Myreen and Jared Davis. A verified runtime for a verified theorem prover. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem*

*Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*, volume 6898 of *Lecture Notes in Computer Science*, pages 265–280. Springer, 2011.

32. Magnus O. Myreen and Jared Davis. The reflective Milawa theorem prover is sound - (down to the machine code that runs it). In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 421–436. Springer, 2014.
33. Magnus O. Myreen and Michael J. C. Gordon. Function extraction. *Sci. Comput. Program.*, 77(4):505–517, 2012.
34. Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. Decompilation into logic - improved. In Gianpiero Cabodi and Satnam Singh, editors, *FMCAD*, pages 78–81. IEEE, 2012.
35. Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *J. Funct. Program.*, 24(2-3):284–315, 2014.
36. Lawrence C. Paulson. A higher-order implementation of rewriting. *Sci. Comput. Program.*, 3(2):119–149, 1983.
37. Sriram K. Rajamani and David Walker, editors. *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. ACM, 2015.
38. Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 379–391. ACM, 2009.
39. Konrad Slind and Michael Norrish. A brief overview of HOL4. In Mohamed et al. [25], pages 28–32.
40. Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional compcert. In Rajamani and Walker [37], pages 275–287.
41. Pierre-Yves Strub, Nikhil Swamy, Cédric Fournet, and Juan Chen. Self-certification: bootstrapping certified typecheckers in F* with Coq. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 571–584. ACM, 2012.
42. Ken Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8):761–763, 1984.
43. Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle framework. In Mohamed et al. [25], pages 33–38.
44. Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294. ACM, 2011.