CakeML: A Verified Implementation of ML

Ramana Kumar * [1]   Magnus Myreen † [1]   Michael Norrish [2]   Scott Owens [3]

[1] Computer Laboratory, University of Cambridge, UK
[2] Canberra Research Lab, NICTA, Australia ‡
[3] School of Computing, University of Kent, UK

Abstract

We have developed and mechanically verified an ML system called CakeML, which supports a substantial subset of Standard ML. CakeML is implemented as an interactive read-eval-print loop (REPL) in x86-64 machine code. Our correctness theorem ensures that this REPL implementation prints only those results permitted by the semantics of CakeML. Our verification effort touches on a breadth of topics including lexing, parsing, type checking, incremental and dynamic compilation, garbage collection, arbitrary-precision arithmetic, and compiler bootstrapping.

Our contributions are twofold. The first is simply in building a system that is end-to-end verified, demonstrating that each piece of such a verification effort can in practice be composed with the others, and ensuring that none of the pieces rely on any over-simplifying assumptions. The second is developing novel approaches to some of the more challenging aspects of the verification. In particular, our formally verified compiler can bootstrap itself: we apply the verified compiler to itself to produce a verified machine-code implementation of the compiler. Additionally, our compiler proof handles diverging input programs with a lightweight approach based on logical timeout exceptions. The entire development was carried out in the HOL4 theorem prover.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness proofs, Formal methods; F.3.1 [*Logics and meanings of programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification, Specification techniques, Invariants

Keywords Compiler verification; compiler bootstrapping; ML; machine code verification; read-eval-print loop; verified parsing; verified type checking; verified garbage collection.

1. Introduction

The last decade has seen a strong interest in verified compilation; and there have been significant, high-profile results, many based on the CompCert compiler for C [1, 14, 16, 29]. This interest is easy to justify: in the context of program verification, an unverified compiler forms a large and complex part of the trusted computing base. However, to our knowledge, none of the existing work on verified compilers for general-purpose languages has addressed all aspects of a compiler along two dimensions: one, the compilation algorithm for converting a program from a source string to a list of numbers representing machine code, and two, the execution of that algorithm as implemented in machine code.

Our purpose in this paper is to explain how we have verified a compiler along the full scope of both of these dimensions for a practical, general-purpose programming language. Our language is called CakeML, and it is a strongly typed, impure, strict functional language based on Standard ML and OCaml. By verified, we mean that the CakeML system is ultimately x86-64 machine code alongside a mechanically checked theorem in higher-order logic saying that running that machine code causes an input program to yield output or diverge as specified by the semantics of CakeML.

We did not write the CakeML compiler and platform directly in machine code. Instead we write it in higher-order logic and synthesise CakeML from that using our previous technique [22], which puts the compiler on equal footing with other CakeML programs. We then apply the compiler to itself, *i.e.*, we *bootstrap* it. This avoids a tedious manual refinement proof relating the compilation algorithm to its implementation, as well as providing a moderately large example program. More specifically,

- we write, and can run, the compiler as a function in the logic, and we synthesise a CakeML implementation of the compiler inside the logic;
- we bootstrap the compiler to get a machine-code implementation inside the logic; and
- the compiler correctness theorem thereby applies to the machine-code implementation of the compiler.

Another consequence of bootstrapping is that we can include the compiler implementation as part of the runtime system to form an interactive read-eval-print loop (REPL). A verified REPL enables high-assurance applications that provide interactivity, an important feature for interactive theorem provers in the LCF tradition, which were the original motivation for ML.

Contributions

- Semantics that are carefully designed to be simultaneously suitable for proving meta-theoretic language properties and for supporting a verified implementation. (Section 3)
- An extension of a proof-producing synthesis pathway [22] originally from logic to ML, now to machine code (via verified compilation). (Sections 4–6, 10)

* supported by the Gates Cambridge Trust

† supported by the Royal Society, UK

‡ NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

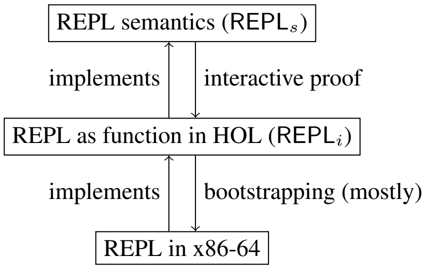
POPL '14, January 22–24, 2014, San Diego, CA, USA..
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2544-8/14/01…$15.00.
http://dx.doi.org/10.1145/2535838.2535841

- A lightweight method for proving divergence-preservation using a clock and timeout exceptions — these only appear in the proof, not in the implementation. It allows us to do all of our compiler proofs by induction and in the direction of compilation. (Section 7)
- To the best of our knowledge, this is the first bootstrapping of a formally verified compiler. The bootstrapping is done within the logic so that the compiler correctness theorem can be instantiated to apply directly to the compiler's output. (Section 9)

Result The result of this work is an end-to-end verified REPL implementation in 64-bit x86, including lexing, parsing, type inference, compilation, garbage collection, printing, and arbitrary-precision integer arithmetic. The entire formal development was carried out in the HOL4 theorem prover [8]. All of our definitions and proofs are available at `https://cakeml.org`.

2. Approach

The CakeML system and verification divides into three layers.



At the top is the semantic specification of the REPL. At the bottom is machine code implementing the REPL. In between is an implementation of the REPL as a function in logic. We describe the components of each layer in more detail below, then give an overview of the proof that each layer implements the one above it.

The REPL Specification The semantics for the CakeML REPL is given by a relation

$$\mathsf{REPL}_s : \mathsf{bool\ list} \to \mathsf{string} \to \mathsf{repl\_result} \to \mathsf{bool}$$

between a string representing the read input and a repl_result, which is a list of strings representing the printed output, ending in a nil value that indicates whether the REPL has terminated or diverged. The bool list argument indicates which input declarations have type errors (needed because we have not yet verified completeness for the type inferencer; see Section 5).

The input string is treated as a series of declarations separated by semicolons. Each declaration might yield an output string describing a result value or reporting a lex, parse, or type error; or it might diverge.

In the definition of REPL_s, we apply an executable lexer specification to the entire input string and split the resulting list of tokens at top-level semicolons, yielding a list of possibly invalid declarations.[1] For each declaration, we pick a valid parse tree according to the CakeML grammar, if one exists. If none exists, the result for that declaration is a parse error. We then translate away some syntactic sugar, and resolve the scoping of types and datatype constructors.

CakeML has a declarative (non-algorithmic) typing relation in the standard style. If the declaration has a type, we move on to the operational semantics. If it does not have a type, then the semantics gets stuck or signals a type error depending on the corresponding value in the bool list.

The operational semantics is given as a big-step relation, on which our compiler correctness proofs can all proceed by induction since both the compiler and big-step semantics are defined inductively over CakeML abstract syntax. To precisely define divergence, and to support a syntactic type-soundness proof [31], we also give a small-step semantics for expressions as a CEK machine [5], and prove that it is equivalent to the big-step relation.

If the declaration diverges, REPL_s relates it to the repl_result divergence-nil value. If it terminates normally or with an unhandled exception, or if a parse or type error occurred, REPL_s updates its state, converts the result to a string and conses it to the result of evaluating the rest of the input.

The REPL Implementation The REPL_s specification is implemented by a HOL function

$$\mathsf{REPL}_i : \mathsf{string} \to \mathsf{repl\_result}$$

from an input string to a list of result strings ending in either termination or divergence (as above). The function is total; however, it is not computable, since it correctly ends the output list with divergence when necessary: being a function in the logic, it is able to ask whether there exist a number of steps in which execution of code for the next declaration would terminate.

We define REPL_i as a loop that consumes the input string while transforming the state of a low-level virtual machine for CakeML Bytecode (Section 6.3) and accumulating results. The loop is split into three parts:

1. ($\mathsf{REPL}_i\mathsf{\_step}$: Read and compile) The first part reads and lexes the input until the next semicolon, parses the resulting tokens, performs type inference on the resulting syntax and then compiles it to produce bytecode.
2. (Evaluate) The second part returns the result, if it exists, of executing the bytecode simulator on the bytecode, or else signals divergence.
3. (Print and Loop) If the simulator terminates, the last part continues the loop with the new machine state after accumulating the results from the new state. If the simulator diverges, or if all input has been read, the loop ends. If any step fails, the loop continues after accumulating the error message.

The REPL in Machine-Code The bottom layer is a machine-code implementation of the REPL, created mostly by bootstrapping. Specifically, we translate the HOL function, $\mathsf{REPL}_i\mathsf{\_step}$, the main part of REPL_i, into CakeML declarations using proof-producing synthesis [22]. We then evaluate the compiler in the logic on those declarations to produce a verified bytecode implementation of $\mathsf{REPL}_i\mathsf{\_step}$. This bytecode is mapped via a verified translation into x86-64 machine code.

To implement the other parts of REPL_i, we separately synthesise a small amount of x86-64 machine code to jump to the code produced by the compiler, and to tie the loop together. We also synthesise verified machine code for the lexer, printer, garbage collector and arbitrary-precision integer package, which complete the verified implementation.

2.1 Correctness and Verification

The correctness of the x86-64 machine-code implementation of the CakeML REPL is stated with respect to our machine model for x86-64. The theorem (which appears in more detail as Theorem 25) can informally be read as follows:

Theorem 1. If the machine is loaded with the CakeML REPL code and started with accessible memory for the data and code heaps, and provided with a finite input character stream i, then one of the following happens.

- The machine terminates successfully with an output stream $output$, and there exist l and o such that $\mathsf{REPL}_s\ l\ i\ o$ holds, o

[1] We use an executable specification instead of a regular expression-based one because ML comments are not expressible with regular expressions.

ends with termination, and out $o = output$. (Here out converts a repl_result to a string by joining the results.)

- The machine diverges after producing a finite output stream $output$, and there exist l and o such that REPL$_s$ l i o holds, o ends with divergence, and out $o = output$.
- The machine terminates with an out-of-memory error. The output stream is terminated with an error message.

We prove the correctness theorem using two methods: interactive proof, to establish a connection between REPL$_s$ and REPL$_i$ and to put the final result together, and a combination of interactive proof and proof-producing synthesis (including bootstrapping) to construct verified machine code that implements REPL$_i$.

The main lemma proved interactively is that for every $input$, there is an l such that REPL$_s$ l $input$ (REPL$_i$ $input$) holds. We prove this (Section 8) by composing proofs that relate each of the components of REPL$_i$ to the corresponding component of REPL$_s$.

Since REPL$_i$ reads its input one character at a time, whereas REPL$_s$ splits the whole input string up front, we first prove, for the lexer, that the resulting list of declarations is the same either way. We then prove our parser sound and complete with respect to CakeML's grammar (Section 4), which means the results of parsing in REPL$_i$ match the choice of valid parse tree made by REPL$_s$.

We prove soundness for the type inference algorithm: if a type is inferred, the type system allows that type. We verify the rest of the loop in REPL$_i$ under the assumption that type inference succeeded, and hence the operational semantics does not encounter a type error. Internally, REPL$_s$ is defined by inductive relations parameterised by an environment and store. We maintain an invariant between these semantic objects, the state of the compiler, and the bytecode machine state.

Our first compiler correctness theorem (Section 6) is for when REPL$_s$ says the declaration yields a new environment and output string. We carry our invariant around an iteration of the REPL$_i$ loop with this theorem, which says running the compiled code for the declaration leads the virtual machine to produce the same output string and to terminate in a state that satisfies the invariant updated with the new environment.

In the case of divergence, we have a second compiler correctness theorem (Section 7) that says that if REPL$_s$ says a declaration diverges, then a run of the compiled code for that declaration also diverges, causing the entire REPL to diverge.

The x86-64 machine-code implementation is constructed using forward proof, most of which is automated. The bulk of the verified machine code, *i.e.* that implementing REPL$_i$_step, is produced via compiler bootstrapping:

1. We start by translating the HOL function REPL$_i$_step into CakeML abstract syntax, a list of CakeML declarations (ml_repl_step_decls). The translation is automatic and proof-producing: it proves that the generated CakeML declarations implement the HOL functions.
2. Once we have the REPL$_i$_step in CakeML abstract syntax, we evaluate the verified compiler on this abstract syntax. The evaluation happens *inside the logic* and proves a theorem:

   $$\text{compile\_decs init ml\_repl\_step\_decls} = (\text{bytecode}, \text{cstate})$$

   By the compiler correctness theorems, we prove that the generated code, *i.e.* bytecode, accurately implements the CakeML declarations produced in the previous step.
3. In order to construct verified and executable machine code from bytecode, we apply a verified translation from bytecode instructions to x86-64 machine code.

```
id    ::=  x | Mn.x
cid   ::=  Cn | M.Cn
t     ::=  int | bool | unit | α | id | t id | (t(,t)*)id
       |   t * t | t -> t | t ref | (t)
l     ::=  i | true | false | () | []
p     ::=  x | l | cid | cid p | ref p | _ | (p(,p)*) | [p (,p)*]
       |   p :: p
e     ::=  l | id | cid | cid e | (e,e (,e)*) | [e (,e)*]
       |   raise e | e handle p => e (| p => e)*
       |   fn x => e | e e | ((e ;)* e) | uop e | e op e
       |   if e then e else e | case e of p => e (| p => e)*
       |   let (ld|;)* in (e ;)* e end
ld    ::=  val x = e | fun x y+ = e (and x y+ = e)*
uop   ::=  ref | ! | ~
op    ::=  = | := | + | - | * | div | mod | < | <= | > | >= | <> | ::
       |   before | andalso | orelse
c     ::=  Cn | Cn of t
tyd   ::=  tyn = c (| c)*
tyn   ::=  (α(,α)*) x | α x | x
d     ::=  datatype tyd (and tyd)* | val p = e
       |   fun x y+ = e (and x y+ = e)*
       |   exception c
sig   ::=  :> sig (sl |;)* end
sl    ::=  val x : t | type tyn | datatype tyd (and tyd)*
top   ::=  structure Mn sig? = struct (d |;)* end; | d; | e;
```

where x and y range over identifiers (must not start with a capital letter), α over SML-style type variables (*e.g.*, 'a), Cn over constructor names (must start with a capital letter), Mn over module names, and i over integers.

Figure 1. CakeML source grammar

The steps outlined above produced verified x86-64 code for part of REPL$_i$, namely REPL$_i$_step. We construct the machine code for all other parts, including garbage collector and bignum library, using a different form of proof-producing synthesis of machine code [23]. The top-level theorem is proved interactively by plugging together theorems describing the behaviour of the components.

3. The Specification of CakeML

CakeML (Figure 1) is a subset of Standard ML (SML) [18], including datatypes and pattern-matching, higher-order functions, references, exceptions, polymorphism, and modules and signatures. CakeML integers are arbitrary precision. The main features not included are records, input/output, and functors.

All CakeML programs should run on an off-the-shelf SML implementation with the same result;[2] however, we have not proved that fact with respect to the *Definition of Standard ML.* Thus, any formal reasoning about CakeML programs must be carried out with respect to the CakeML semantics, rather than SML's. We chose to be faithful to SML so that potential CakeML users do not have to learn a new programming language from scratch, but our main focus is not on SML *per se*, but rather on a call-by-value, impure, strongly typed, functional language. We choose SML over OCaml for technical reasons (Section 12); on our subset, the difference is mostly in the syntax.

Section 2 summarises the specification: as a lexer, context-free grammar, de-sugarer, elaborator, type system, and big-step operational semantics wrapped in a REPL. Most of these are typical, so here we focus on their properties, and on our design decisions.

CakeML's concrete syntax is specified by a context-free grammar similar to Figure 1's, but that unambiguously encodes the various parsing precedences and resolves the "dangling case" ambiguity. The transformation to abstract syntax removes syntactic sugar.

[2] CakeML does not enforce equality types, so an SML implementation will statically reject some programs that CakeML accepts. Our implementation of equality raises an exception if one of its arguments contains a closure.

For example, a Let expression with multiple val bindings is desugared into nested Lets, each with one binding.

Our abstract syntax uses a first-order representation of binding where variables, constructors, etc. are represented as strings. This straightforward representation fits well with our semantics, which uses closures rather than substitutions, and it also enables bootstrapping, because the AST closely corresponds to a CakeML datatype (CakeML does not support strings, but they become lists of integers during synthesis).

The elaborator walks the AST and replaces uses of types and constructors in patterns and expressions with fully qualified names, *i.e.*, with the full module path to the constructor. Thus, each constructor is canonically named, and the type system and operational semantics do not have to generate unique stamps at datatype declarations in order for a type preservation lemma to hold.

A key design principle for the operational semantics is for it to raise a distinguished exception Rtype_error whenever something goes wrong, instead of getting stuck or continuing on anyway. By raising Rtype_error instead of getting stuck, the big-step semantics has a precise correspondence between divergence and failure of an expression to relate to a result. This is a key part of our technique for verifying the compiler for diverging input programs (Section 7).

The Rtype_error behaviour of the semantics forms the interface between the type soundness proof and the compiler proof, as the compiler proof assumes that programs do not raise Rtype_error, and type soundness theorem guarantees this. Even though these choices only affect programs that do not type check, they are how the compiler proof can use the fact that these situations do not arise. For example, when evaluating a pattern match, the value's constructor might be from a different type than the first pattern's, as in `case true of None => 0 | true => 1`. The semantics could just move on to the next pattern since the value's and first-pattern's constructors are not equal. Instead the semantics detects this case and raises Rtype_error.

The semantics relies on the following definitions of values v; environments for variables $envE$, constructors $envC$, and modules $envM$; stores s; and results r. The constructor environment records the arity of the constructor and what type of values it constructs.

$$
\begin{array}{lcl}
v & ::= & loc \mid l \mid cid \mid cid\ v \mid \texttt{(}v\texttt{,}v\texttt{(,}v\texttt{)}^*\texttt{)} \mid \mathsf{Closure}\langle E, x, e\rangle \\
 & \mid & \mathsf{RecClosure}\langle E, (\texttt{fun}\ x\ y \texttt{ = } e\ (\texttt{and}\ x\ y \texttt{ = } e)^*), x\rangle \\
loc & ::= & \mathsf{num} \\
envE, E & = & (x, v)\ \mathsf{list} \\
envC, C & = & (cid, \mathsf{num}{*}id)\ \mathsf{list} \\
envM, M & = & (id, E)\ \mathsf{list} \\
s & = & loc\ \mathsf{list} \\
r & ::= & \langle s, C, \langle M, E\rangle\rangle \mid \langle s, C, ex\rangle \mid \langle s, C, \mathsf{Rtype\_error}\rangle
\end{array}
$$

The type system relies on typing contexts for variables tE, constructors tC, modules tM, and stores tS. We then have relations for evaluation, divergence, and typing for top-level declarations.

$$
\begin{array}{l}
\mathsf{evaluate\_top} : M \rightarrow C \rightarrow s \rightarrow E \rightarrow top \rightarrow r \rightarrow \mathsf{bool} \\
\mathsf{top\_diverges} : M \rightarrow C \rightarrow s \rightarrow E \rightarrow top \rightarrow \mathsf{bool} \\
\mathsf{type\_top} : tM \rightarrow tC \rightarrow tE \rightarrow top \rightarrow tM \rightarrow tC \rightarrow tE \rightarrow \mathsf{bool}
\end{array}
$$

We use closures for function values, rather than performing substitutions at function applications, to keep a close connection between the semantics of functions and their implementation strategy.

The REPL repeatedly calls evaluate_top, and must update its state between each call, installing new bindings, using the new store, etc. After completing a module declaration, the REPL updates the type system's state with the module's bindings according to its signature, since future declarations must not refer to hidden bindings or constructors. However, the operational semantics' state must be updated because references to the hidden constructors will generally be reachable from the environment or from the store.

The last remaining subtlety lies in module declarations that raise an un-handled exception part-way through. When any declaration raises an exception, its bindings do not take effect, and so the type environments are not updated. However, the operational semantics might need its constructor information updated, since the module might have placed a locally defined constructor in the store first before raising the exception.

We have two options, either updating the constructor environment with just the constructors seen, or with all of the constructors from the module body. Both are sound, and neither affect the programmer, since they are not in the type environments. We choose the latter, to match how the compiler implements modules.

Metatheory

Theorem 2 (Determinism)**.** If evaluate_top $M\ C\ s\ E\ top\ r_1$ and evaluate_top $M\ C\ s\ E\ top\ r_2$ then $r_1 = r_2$.

Proof sketch. By induction on the big-step relation. □

Theorem 3 (Totality)**.** $(\exists r.\ \mathsf{evaluate\_top}\ M\ C\ s\ E\ top\ r)$ iff $\neg\mathsf{top\_diverges}\ M\ C\ s\ E\ top$.

Proof sketch. First, we establish, by induction along a small step trace, that the small-step semantics cannot get stuck. Then we use the big-step/small-step equivalence theorem to transfer the result to the big-step semantics. □

The invariant of the type soundness theorem existentially quantifies constructor and module type environments that represent the declarations that are hidden by module signatures. The update_type_sound_inv function updates the state depending on whether the result is an exception or not following the discussion above.

type_sound_inv$(tM, tC, tE, M, C, E, s) =$
$\exists\ tS\ tM\_no\_sig\ tC\_no\_sig.$
the environments are well-formed and consistent with each other,
MAP FST tM_no_sig = MAP FST tM,
tM_no_sig weakens tM,
tC_no_sig weakens tC, and
the constructors in tC_no_sig but not tC are all in modules in the domain of tM.

Theorem 4 (Type soundness)**.**
Let $state = \langle tM, tC, tE, M, C, E, s\rangle$ be given.
For all top, if type_sound_inv $state$ and
type_top $tM\ tC\ tE\ top\ tM'\ tC'\ tE'$ then either
top_diverges $M\ C\ s\ E\ top$ or there exists s', C', and r such that
$r \neq$ Rtype_error,
evaluate_top $M\ C\ s\ E\ top\ \langle s', C', r\rangle$, and
type_sound_inv
(update_type_sound_inv $top\ state\ tM'\ tC'\ tE'\ s'\ C'\ r$)

Proof sketch. We prove the usual preservation and progress lemmas along a small-step trace. Then we use the big-step/small-step equivalence theorem to transfer the result to the big-step semantics. We then reason about declarations entirely with the big-step semantics. This poses no problem because declarations cannot diverge apart from their sub-expressions diverging. The preservation and progress lemmas were proved directly on our CEK-style small-step semantics, which uses closures and a context stack. To our knowledge no such proof appears in the literature. The main impact on the proof is to flatten out its structure (*e.g.*, no substitution lemma), and instead to require an invariant that types the continuation stack. The usual subtleties about type substitution and weakening were unchanged; we use de Bruijn indices for type variables. □

Part II: Verification and Implementation

We now turn to the implementation and verification of each part of the compiler. Afterward, we include a high-level discussion of CakeML's design and significance (Section 12).

4. Parsing

We implement the first phase of parsing with a Parsing Expression Grammar (PEG), following Koprowski and Binsztok's mechanisation in Coq [11]. We have a general inductive relation $\mathsf{peg\_eval}$, which takes a grammar, an input sequence of tokens, a PEG expression, and returns a verdict of either failure, or success with a returned value and the remaining input. We prove the results from Koprowski and Binsztok that the $\mathsf{peg\_eval}$ relation always has a value when the PEG is well-formed, and that it is deterministic. Thus, on well-formed PEGs (we prove the CakeML PEG is one such), $\mathsf{peg\_eval}$ specifies a function.

PEG expressions are accompanied by "semantic actions" that can transform the values associated with sub-parses into a combined return value. Our semantic actions, in HOL, cannot be dependently typed, as can be done in Coq where it is pleasant to have parses of different non-terminals return values of different types. Instead, all our PEG-expressions have semantic actions that simply return lists of CFG-parse-trees. In this way, the first phase of parsing concentrates on turning a linear sequence of tokens into a tree; later phases map different parts of these trees into different types.

Even though $\mathsf{peg\_eval}$ specifies a function, its definition is not well-suited to computation. Rather, we define a general PEG interpreter $\mathsf{peg\_exec}$ that is tail-recursive, written in a continuation-passing style. Whenever its PEG argument is well-formed, we prove the interpreter has a well-defined result, and one that is equal to the result demanded by the $\mathsf{peg\_eval}$ relation. It is this definition that is compiled into the bytecode that is eventually executed.

Theorem 5 (Parser Soundness)**.** Whenever the PEG successfully parses a non-terminal N, its result is a single CFG parse-tree with N at its head and whose fringe of tokens corresponds to the input consumed by the execution of the PEG.

Proof sketch. Induction on the lexicographic combination of the length of the input given to $\mathsf{peg\_eval}$, and the *rank* of the non-terminal. Each non-terminal is given a natural number rank such that if executing the PEG on non-terminal N can result in needing to execute non-terminal M without having consumed any input, then N's rank is greater than M's. The fact that this ranking is possible stems from the same argument that gives the well-formedness of the PEG. □

Theorem 6 (Parser Completeness)**.** If a parse tree exists for a given input, then the PEG implementation will find it.

Proof sketch. Induction on the lexicographic combination of the length of the parse tree's fringe, and the rank of the tree's head non-terminal. □

As PEG execution is deterministic, this also implies that our CFG is unambiguous.

5. Type Inference

Type inference is based on Milner's algorithm $\mathcal{J}$ (and hence $\mathcal{W}$) [17]. We represent unification variables as numbers, and increment a counter to generate fresh variables. Since the inferencer is written in higher-order logic (in a state-and-exception monad), we cannot use the common technique of doing substitutions by updating pointers in the types. Instead we re-use our previous formalisation (from the HOL4 examples directory) of *triangular* substitutions, which are not idempotent, but are instead designed to maintain sharing for greater efficiency in a pure functional setting [12].

Theorem 7 (Type Inferencer Soundness)**.**
If $\mathsf{infer\_top}\ tM\ tC\ tE\ top\ state =$
$(\mathsf{Success}(tM', tC', tE'), state')$ and tM, tC, tE contain no unification variables, then $\mathsf{type\_top}\ tM\ tC\ tE\ top\ tM'\ tC'\ tE'$.

Proof sketch. The soundness proof is long and tedious, but not particularly surprising. A large part of the proof is devoted to establishing invariants on the usage of type variables in the various environments and state components of the algorithm. □

There are two ways in which our inferencer verification falls short of what could be seen as an ideal for SML, but neither directly affects our end-to-end correctness theorem. First, the types of let-bound variables are not generalised, restricting polymorphism to top-level and module-top-level declarations;[3] and second, we have not verified completeness of the inferencer. In both cases, the potential for the CakeML system to behave badly is limited to signalling a type error on a declaration that the programmer expected to pass type checking. In particular, there is no possibility for the programmer to be deceived by the system appearing to returning an erroneous result of executing the definition.

We gave proving inferencer completeness low priority because, in practice, we find the inferencer does not fail on the programs we expect to have types. If it did, a completeness theorem would tell us that the bug is in CakeML's declarative type system, rather than possibly also in the inferencer.

6. Compilation

We now turn to translation from abstract syntax to CakeML Bytecode, our assembly language for a virtual stack machine, which is the final language before translation to x86-64 machine code. We describe the bytecode in detail in Section 6.3, but first take a high-level look at the compiler and its verification.

The main function is $\mathsf{compile\_top}$ which takes a top-level declaration and the compiler's state, including environments mapping variables to bytecode stack offsets, and returns code and two new compiler states. One is used if the code runs successfully and the other if it raises an un-handled exception. For the verification, we define a relation

$$\mathsf{compiler\_inv}\ envM\ envC\ s\ envE\ rs\ z\ rd\ bs$$

between the semantic context (the module, constructor, and value environments, and the store), the compiler's state rs, the bytecode machine state bs, and additional context indicating the size, z, of the stack before the last complete declaration, and information about references and closures in rd.

We explain the compiler along with its verification, beginning with the statement of the correctness theorem for terminating programs (for diverging programs see Section 7).

Theorem 8 (Compiler Correctness)**.** If
top is well-typed, $\mathsf{evaluate\_top}\ m\ c\ s\ e\ top\ res$,
$\mathsf{compile\_top}\ rs\ top = (rss, rsf, code)$, and,
$\mathsf{compiler\_inv}\ m\ c\ s\ e\ rs\ |bs.\mathsf{stack}|\ rd\ bs$,
then a run of bs with $code$ terminates with a new state bs', satisfying the following condition, depending on res:
If res is a successful result with new environments m', c', and e', and new store s', then there exists rd' such that
bs' has reached the end of $code$ (no next instruction),
$\mathsf{compiler\_inv}\ m'\ c'\ s'\ e'\ rss\ |bs'.\mathsf{stack}|\ rd'\ bs'$, and,
bs' has the result of printing the new bindings in its output.

[3] Vytiniotis et al. [30, Section 4.3] provide evidence that generalisation for let expressions is rarely used in practice.

Otherwise if res is an exception exc with a new store s', then there exists rd' and bv such that

the next instruction for bs' is Stop,
bs'.stack $= bv :: bs$.stack,
bv is a refinement of exc, and,
compiler_inv $m\ c'\ s'\ e\ rsf\ |bs.\mathsf{stack}|\ rd'\ bs''$,

where bs'' is bs' with bv popped. (The printing and popping of bv is done by the main REPL loop after the Stop instruction is reached.)

Proof sketch. The compile_top function is implemented using functions for compiling declarations and expressions, mirroring the structure of the big-step semantics. The proof for each layer is by induction on the appropriate big-step relation. □

Our compilation strategy has three phases: translation to an intermediate language, analysis for the purpose of compiling functions, and finally translation to bytecode. We proceed with a description of the implementation and verification of each phase.

6.1 Translation to Intermediate Language

The intermediate language (IL) simplifies the source language in the following ways:

- Pattern-matching is replaced by conditional expressions and constructor-tag equality tests.
- Named variables are replaced by de Bruijn indices. Constructor names are replaced by numeric tags.
- All functions are syntactically recursive, and may have zero or more arguments.
- The set of primitive operations is smaller (*e.g.* no greater-than, only less-than).

The translation to IL expressions is a straightforward recursion on the abstract syntax of CakeML. We also define a translation from CakeML values to IL values, which makes use of the expression translation since closures are amongst the values.

Verification We give a big-step operational semantics for the intermediate language, similar to the one for the source language. The correctness theorem for the translation to IL says whenever the CakeML semantics gives a program p a result r, the IL semantics gives the translated program $[p]$ a result r' that is related to $[r]$. The proof is by induction on the CakeML big-step evaluation relation.

We cannot prove identity between the translated CakeML result and the result produced by the IL semantics, because closure environments may differ. The translation to IL sometimes introduces fresh variables that will appear in an IL closure's environment but not in the translation of the source closure's. We need a coarser relation on IL closures, allowing their environments to differ.

For this purpose, we define a relation $\approx$ on IL expressions, values, and environments. The relation $V \vdash (z_1, e_1) \approx (z_2, e_2)$ relates an IL expression together with the size of its environment to another such pair. It is parameterised by a relation $V v_1 v_2$ indicating variables that are assumed to be bound to equivalent values. To relate values, the relation $v_1 \approx v_2$ needs no sizes or parameters, since closures carry their environments so the size can be computed.

We have proved $\approx$ reflexive and transitive (symmetry fails as explained in the next section) and the following two theorems.

Theorem 9. If the IL semantics says e_1 evaluates in environment env_1 to a result r_1; whenever $V\ v_1\ v_2$ holds, $env_1(v_1) \approx env_2(v_2)$ does also; and, $V \vdash (|env_1|, e_1) \approx (|env_2|, e_2)$; then there is a result r_2 to which e_2 evaluates in env_2 and $r_1 \approx r_2$.

Proof sketch. By induction on the IL evaluation relation. □

The main purpose of the $\approx$ relation is to enable closure environments to be manipulated. The second theorem supports the renumbering of variables that is required in a closure's body when its environment is changed.

Theorem 10. Let e' be the result of renumbering variables in e and suppose V relates indices under the same renumbering scheme, then $V \vdash e \approx e'$.

Proof sketch. By induction on syntax-directed renumbering. □

We use these theorems for verifying the translation to IL including the removal of pattern matching. We also use $\approx$ to relate IL function bodies to annotated IL closure bodies, described below.

6.2 Intermediate Language Closure Annotation

For every function body, we compute how to compile each occurrence of a variable, and how to build the closure environment. We use flat closure environments that only have bindings for variables that actually occur free in the function's body.

As an example, consider the following IL expression (written in concrete syntax with named variables, but recall that the IL actually uses de Bruijn indices):

```
let val a = e1 val b = e2 val c = e3
    fun f h x =
      if x = 6 then h 7 else (g x) + (f (x - a))
    and g x = f (fn y => x + y - b) (a - c)
in e4 end
```

Analysis of the bodies records this about the free variables:

for `f`:	for `g`:	for λ:
`x -> arg 0`	`f -> rec 0`	`x -> env 0`
`h -> arg 1`	`b -> env 0`	`y -> arg 0`
`g -> rec 1`	`a -> env 1`	`b -> env 1`
`f -> self`	`c -> env 2`	cl env = `[0,2]`
`a -> env 0`	cl env = `[1,2,0]`	
cl env = `[2]`		

The `env` annotation gives an index into the closure environment, which is itself a list of indices into the enclosing environment (so in the anonymous function's closure environment, `0` refers to `g`'s argument, and `2` refers to the first element of `g`'s environment, since `1` would refer to `g` itself).

We update the variables in function bodies so that they refer to the closure environment instead of the enclosing environment. For example, in the body of `f`, the de Bruijn index for `a` is decremented twice, because the intervening variables in the enclosing environment are omitted in the closure environment. CakeML's operational semantics also uses closures, making the proof of correspondence conceptually straightforward, but closures in the semantics contain the whole enclosing environment.

We generate, for each function body, a unique label that is used (Section 6.4) as a code pointer to bytecode implementing the body.

Verification We store the closure environment information as annotations on the function bodies within the expression. The IL semantics uses the closure (rather than enclosing) environment to evaluate a call when such an annotation exists. Similarly, the relation $V, U \vdash (z_1, bs_1) \approx (z_2, bs_2)$ allows the bodies in bs_1 and bs_2 to be annotated, and uses the closure environment rather than V and z as appropriate. The relation is directed (hence not symmetric): if a body in bs_1 is annotated, then the corresponding body in bs_2 must have the same annotation; however an unannotated body in bs_1 may be related to an annotated body in bs_2.

Theorem 11. If e' is the result of annotating e, and e is unannotated and has free variables all less than z, then $(=) \vdash (z, e) \approx (z, e')$, and e' is fully annotated with environments that cover the free variables of its bodies.

Proof sketch. By induction on syntax-directed annotation. □

$$
\begin{array}{lcl}
\mathit{bc\_inst} & ::= & \mathsf{Stack}\ \mathit{bc\_stack\_op} \mid \mathsf{PushExc} \mid \mathsf{PopExc} \\
 & \mid & \mathsf{Return} \mid \mathsf{CallPtr} \mid \mathsf{Call}\ \mathit{loc} \\
 & \mid & \mathsf{PushPtr}\ \mathit{loc} \mid \mathsf{Jump}\ \mathit{loc} \mid \mathsf{JumpIf}\ \mathit{loc} \\
 & \mid & \mathsf{Ref} \mid \mathsf{Deref} \mid \mathsf{Update} \mid \mathsf{Print} \mid \mathsf{PrintC}\ \mathsf{char} \\
 & \mid & \mathsf{Label}\ n \mid \mathsf{Tick} \mid \mathsf{Stop} \\
\mathit{bc\_stack\_op} & ::= & \mathsf{Pop} \mid \mathsf{Pops}\ n \mid \mathsf{Shift}\ n\ n \mid \mathsf{PushInt}\ \mathsf{int} \\
 & \mid & \mathsf{Cons}\ n\ n \mid \mathsf{El}\ n \mid \mathsf{TagEq}\ n \mid \mathsf{IsBlock}\ n \\
 & \mid & \mathsf{Load}\ n \mid \mathsf{Store}\ n \mid \mathsf{LoadRev}\ n \\
 & \mid & \mathsf{Equal} \mid \mathsf{Less} \mid \mathsf{Add} \mid \mathsf{Sub} \mid \mathsf{Mult} \mid \mathsf{Div} \mid \mathsf{Mod} \\
\mathit{loc} & ::= & \mathsf{Lab}\ n \mid \mathsf{Addr}\ n \\
n & = & \mathsf{num} \\
\mathit{bc\_value} & ::= & \mathsf{Number}\ \mathsf{int} \mid \mathsf{RefPtr}\ n \mid \mathsf{Block}\ n\ \mathit{bc\_value}^{*} \\
 & \mid & \mathsf{CodePtr}\ n \mid \mathsf{StackPtr}\ n \\
\mathit{bc\_state} & ::= \{ & \mathsf{stack} : \mathit{bc\_value}^{*}; \mathsf{refs} : n \mapsto \mathit{bc\_value}; \\
 & & \mathsf{code} : \mathit{bc\_inst}^{*};\ \mathsf{pc} : n; \mathsf{handler} : n; \\
 & & \mathsf{output} : \mathsf{string};\ \mathsf{names} : n \mapsto \mathsf{string}; \\
 & & \mathsf{clock} : n^{?} \quad \}
\end{array}
$$

Figure 2. CakeML Bytecode syntax, values, and machine state

$$\frac{\mathsf{fetch}(bs) = \mathsf{Stack}\ (\mathsf{Cons}\ t\ n) \quad bs.\mathsf{stack} = vs \mathbin{@} xs \quad |vs| = n}{bs \to (\mathsf{bump}\ bs)\{\mathsf{stack} = \mathsf{Block}\ t\ (\mathsf{rev}\ vs) :: xs\}}$$

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

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

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

$$\frac{\begin{array}{c}\mathsf{fetch}(bs) = \mathsf{PopExc} \quad bs.\mathsf{handler} = |ys| \\ bs.\mathsf{stack} = x :: xs \mathbin{@} \mathsf{StackPtr}\ h :: ys\end{array}}{bs \to (\mathsf{bump}\ bs)\{\mathsf{stack} = x :: ys;\ \mathsf{handler} = h\}}$$

Figure 3. CakeML Bytecode semantics (selection). The helper function fetch calculates the next instruction using the pc and code, and bump updates the pc to the next instruction.

6.3 CakeML Bytecode

The target language of the compiler is CakeML Bytecode (Figure 2), a low-level assembly language for a virtual machine with a single random-access[4] stack.

CakeML Bytecode was designed with three separate goals: to be (i) conveniently abstract as a target for the compiler and its proofs, and (ii) easy to map into reasonably efficient machine code that is (iii) possible to reason about and verify w.r.t. an operational semantics for x86-64 machine code. To support (i), the bytecode has no notion of pointers to the heap, and provides structured data (Cons packs multiple bytecode values into a Block) on the stack instead. Also, the bytecode Number values are mathematical integers; the x86-64 implementation includes a bignum library to implement the arithmetic instructions. For (ii), we ensure that most bytecode instructions map to one or two x86-64 machine instructions; and for (iii), the bytecode essentially only operates over a single 'stack', the x86-64 stack which we access using the normal stack and base pointers, `rsp` and `rbp` registers. (See Section 10 for the implementation of the bytecode in x86-64.)

The bytecode semantics is a deterministic state transition system: the relation $bs_1 \to bs_2$ fetches from code the instruction indicated by pc and executes it to produce the next machine state. We give some example clauses in Figure 3.

Our data refinement relation $l, r, Cv \models bv$ says bv is a bytecode value representing the IL value Cv. It is parameterised by two functions: l to translate labels to bytecode addresses, and r providing extra information about code pointers for closures.

The refinement of closures is most interesting. There are two components to a closure: its body expression, and its environment which may refer to other closures in mutual recursion. We use a correspondence of labels to link a code pointer to an annotated IL body, and for the environment, we assume the IL annotations correctly specify the closure environment. In the IL, a closure looks like CRecClos $env\ defs\ n$, where env is the enclosing[5] environment, and the body is the nth element of the bundle of recursive definitions $defs$. We say

$$l, r, \mathsf{CRecClos}\ env\ defs\ n \models \mathsf{Block}\ \mathsf{c}\ [\mathsf{CodePtr}\ a; \mathsf{Block}\ \mathsf{e}\ bvs]$$

holds (c and e are tags indicating closure and environment Blocks) when:

- $defs[n]$ has label lab and annotations ann, $l(lab) = a$, and $|ann| = |bvs|$;
- for every variable x with an `env` annotation in ann, the corresponding bytecode value bv in bvs satisfies $l, r, env(x) \models bv$; and,
- for every variable with a `rec` i annotation in ann, the corresponding bytecode value in bvs is RefPtr p, for some p, and there are env', $defs'$, and j such that $r(p) = (env', defs', j)$ and CRecClos $env\ defs\ i \approx$ CRecClos $env'\ defs'\ j$.

Thus, for a function in mutual recursion we assume it is behind the indirection of a RefPtr; the function r acts as an oracle indicating the closure that should be pointed to. To tie the knot, the inductive hypothesis in our compilation proof says whenever $r(p) = (env', defs', j)$ the bytecode machine refs binds p to a value bv satisfying l, r, CRecClos $env\ defs\ j \models bv$.

6.4 Translation to Bytecode

The main compilation algorithm takes an IL expression as input and produces bytecode instructions. Additional context for the compiler includes an environment binding IL variables to stack offsets, and a return context indicating the number of variables that need to be discarded before a jump if the expression is in tail position.

The correctness theorem for this phase is similar to Theorem 8 (whose proof uses this one as a lemma), assuming evaluation in the IL semantics rather than the source semantics. In particular, we have a relation called IL_inv that captures an invariant between the IL environment and store, the compiler state, the bytecode machine state, and proof information like the l and r functions. This relation is used in the definition of compiler_inv, which crosses the three languages (CakeML, IL, Bytecode). The theorem below depends only on the IL and the bytecode.

Theorem 12. If the IL semantics says $Cexp$ evaluates in environment $Cenv$ and store Cs to a new store Cs' and result $Cres$, and all the values in the context are fully annotated, then for all bytecode machine states bs satisfying IL_inv with $Cenv$ and Cs (and proof information including l and r), then

- If $Cres$ is a value, Cv, then
  - Running bs with code from compiling $Cexp$ in non-tail position leads the bytecode machine to terminate in a new state bs' such that IL_inv holds of $Cenv$, Cs', and bs', and $bs'.\mathsf{stack} = bv :: bs.\mathsf{stack}$ with $l, r', Cv \models bv$; and,

[4] Most operations work on the top of the stack, but Load n and Store n read/write the cell n places below the top, and LoadRev takes an index from the bottom.

[5] Annotations on $defs[n]$ build the closure environment from env.

  - Assuming $bs.\mathsf{stack} = lvs \mathbin{@} \mathsf{CodePtr}\ ret :: args \mathbin{@} st$, running bs with code from compiling $Cexp$ in tail position ready to discard $|lvs|$ and $|args|$ leads the machine to a state bs' satisfying the invariant as above, and also $bs'.\mathsf{pc} = ret$ and $bs'.\mathsf{stack} = bv :: st$, with $l, r', Cv \models bv$.
- Otherwise, if $Cres$ is an exception value Cx, and if $bs.\mathsf{stack} = vs \mathbin{@} \mathsf{StackPtr}\ h :: \mathsf{CodePtr}\ hdl :: st$, and $bs.\mathsf{handler} = |st| + 1$, then running bs with code for compiling $Cexp$ (in either call context) leads the machine to a state bs' satisfying the invariant, and with $bs'.\mathsf{pc} = hdl$, $bs'.\mathsf{stack} = bv :: st$, $l, r', Cx \models bv$, and $bs'.\mathsf{handler} = h$.
- Finally, if $Cres$ is a timeout exception, then a run of bs on code for $Cexp$ causes the machine to time out. (See Section 7 for details.)

Proof sketch. By induction on the big-step semantics for the IL. The invariant includes a condition on $bs.\mathsf{code}$: it must already contain code resulting from compiling all the function bodies appearing in $Cexp$ and for closures in Cs and $Cenv$. This assumption is justified by the compilation of function bodies described below. □

Function Bodies Before compiling the main expression, we compile the functions. For each body, the compilation environment is the closure environment and the return context is tail position (ready to discard just the arguments to the function but no local variables). We lay the resulting stretches of bytecode in sequence each preceded by the label annotating the closure's body.

Closure Creation As we saw in the definition of $l, r, Cv \models bv$, we represent a closure using a Cons block containing a code pointer and an environment block, which is built following the annotations on the closure body. For mutually recursive closures (the `rec` annotation), we build the closure environment using references, and update them with the appropriate closures once all the closures are created, thereby ensuring mutually recursive functions appear as RefPtrs in the closure environment.

Function Calls and Proper Tail Calls The generated code for a function call depends on whether it is tail position or not. In both cases, we first evaluate the closure and its arguments, and extract the code pointer from the closure. For a non-tail call, we use the CallPtr instruction, which generates a return pointer and jumps. For a tail call, since we are assuming a return pointer is already on the stack, we reorganise the stack, discarding the local variables and arguments, then use the Return instruction to make the call.

The key lemma enabling our proof by induction for the function call case says that if $bs.\mathsf{stack} = benv :: \mathsf{CodePtr}\ ret :: bvs \mathbin{@} \mathsf{Block}\ \mathsf{c}\ [p; benv]$ and $l, r, \mathsf{CRecClos}\ env\ defs\ n \models \mathsf{Block}\ \mathsf{c}\ [p; benv]$, then we can establish IL_inv for bs with the closure environment made from env and the annotations on $defs[n]$. Thus we can use the inductive hypothesis on the closure body even though it is not a subexpression of the original $Cexp$ in the theorem statement.

Declarations So far we have looked at the compilation of IL expressions. A CakeML program, however, is a sequence of declarations of types, values, or structures. Our IL does not have declarations, so to compile a value declaration `val` p `=` e, we construct the CakeML expression `case` e `of` p `=>` vs, where vs is a tuple of the variables appearing in p, translate this expression to the IL and compile it, and generate a bit of additional code to extract the new bindings from the tuple. For type declarations, we need not generate any code at all and simply update the compiler's state component mapping constructor names to bytecode block tags.

Modules Structure declarations are, from the compiler's perspective, just a sequence of type and value declarations. But they must be treated as a single unit, so there is a subtlety: if any of the declarations within a structure raises an un-handled exception, the bindings for the whole structure need to be discarded. Therefore, we must set up an exception handler around the whole sequence of declarations, enabling unwinding of the stack before proceeding to the next top-level declaration. If the sequence of declarations finishes successfully, we pop the exception handler and hence need to shift the stack offsets for the new bindings in the compiler's environment. We reuse this machinery and its proof for top-level declarations (treating them as unnamed structures with one declaration).

7. Diverging Programs

So far we have seen our compilation algorithm and how we prove that if a source declaration terminates then the bytecode terminates with the same result. By assuming termination, we might appear to admit a compiler that causes programs that should diverge to do anything at all, including returning a wrong result. Here we show how to establish that our compiler in fact preserves divergence.

The proofs of the previous section are all performed in the direction of compilation by induction on the big-step semantics. We would like to handle diverging programs in the same way, to avoid establishing a simulation from a small-step bytecode trace to a small-step source trace against the direction of compilation, and to avoid introducing a co-inductive big-step semantics [15]. Because the bytecode semantics is deterministic, all we have to do is show that the compiler maps diverging source expressions to diverging bytecode expressions.

First, we add optional clocks to the source big-step semantics and to the bytecode machine state. In the big-step semantics, the clock is decremented by 1 on each function call, and a timeout exception is raised if a function is called when the clock is 0. In the bytecode, the `Tick` instruction decrements the clock, and the semantics gets stuck if the clock is 0. The compiler emits a `Tick` instruction for each source function call, and we prove that if a program times out in the semantics with a certain clock, then the compiled version times out in the bytecode with the same clock. This is the core of the compiler proof for divergence, and it follows the same inductive approach as the rest of the compiler proof. The conclusion of Theorem 12 handles the case of a timeout exception, and thereby supports an analogue of Theorem 8 for diverging programs.

It remains to show how to establish our main divergence result when the source semantics ignores the clock and the `Tick` instruction is implemented as a no-op (and thus produces no x86-64 instructions). We sketch the proofs here with a simplified notation. We will write $c \vdash e \Downarrow v$ for convergence in the source language with clock c to a value (or non-timeout exception), and $c \vdash e \Downarrow \emptyset$ for a timeout exception. We use a clock of ∞ to indicate the version that ignores the clock.

Lemma (Big-step Clocked Totality)**.** For all clocks c and expressions e, either $c \vdash e \Downarrow \emptyset$ or $\exists v.\ c \vdash e \Downarrow v$.

Proof sketch. By well-founded induction on the lexicographic ordering of the clock and size of the expression. In all but one case, the applicable big-step rules have inductive premises that have the same or smaller clocks (because the clock is monotonically non-increasing) and smaller sub-expressions. Thus, by induction the results for the sub-expressions combine to give a result for the expression. (It is important here that all mis-applied primitives evaluate to an exceptional result.) The only case where the expression might be bigger is function application, but it decrements the clock first. □

Lemma (Big-step Clock/Unclock)**.** $c \vdash e \Downarrow v$ implies $\infty \vdash e \Downarrow v$ and, $\infty \vdash e \Downarrow v$ implies $\exists c.\ c \vdash e \Downarrow v$.

Proof sketch. Straightforward induction. □

The bytecode's operational semantics is small-step, so we define an evaluation relation in the standard way:

$$c \vdash bs \Downarrow_{bc} bs' \equiv bs\{\texttt{clock} = c\} \rightarrow^* bs' \wedge \forall bs''.\ \neg(bs' \rightarrow bs'')$$

We say the machine has timed out if it evaluates to a state with $\texttt{clock} = 0$ and next instruction `Tick`. A bytecode machine state diverges if it can always take another step.

Lemma (Bytecode Clock/Unclock)**.** $c \vdash bs \Downarrow_{\text{bc}} bs'$ implies $bs\{\texttt{clock} = \infty\} \rightarrow^* bs'\{\texttt{clock} = \infty\}$, and $\infty \vdash bs \Downarrow_{\text{bc}} bs'$ implies $\exists c.\ c \vdash bs \Downarrow_{\text{bc}} bs'\{\texttt{clock} = 0\}$.

Proof sketch. Straightforward induction. □

Lemma (Clocked Bytecode Determinism)**.** $c \vdash bs \Downarrow_{\text{bc}} bs'$ and $c \vdash bs \Downarrow_{\text{bc}} bs''$ implies $bs' = bs''$.

Proof sketch. The small-step relation is deterministic by inspection of the rules; the main result follows by induction on $\rightarrow^*$. □

Theorem 13. Evaluation of e diverges in the un-clocked semantics iff the compilation of e (loaded into a bytecode state bs) diverges in the un-clocked bytecode semantics.

Proof. For the forwards direction, we have $c \vdash e \Downarrow \emptyset$, for all clocks c, by the source language's determinism, and the totality and clock/unclock lemmas. Therefore by the compiler correctness result, we know for all clocks c there is a bs' such that $c \vdash bs \Downarrow_{\text{bc}} bs'$ and bs' is timed out. Now we must show that $bs\{\texttt{clock} = \infty\}$ diverges. Suppose, for a contradiction, there is some bs'' with $\infty \vdash bs \Downarrow_{\text{bc}} bs''$. Let c be one more than the number of `Tick` instructions on the trace from bs to bs'', which is unique by determinism. This contradicts the existence of a bs' above: if evaluation stops before reaching bs'', it will not have passed enough `Tick`s to deplete the clock, and if it reaches bs'' it stops without timing out.

The backwards direction follows easily from Theorem 8 and the clock/unclock lemmas. □

8. Read-Eval-Print Loop

To interact with our compiler, we build a REPL. We define this first as a logic function, REPL$_i$, that implements REPL$_s$. (In later sections, we describe how we produce an x86-64 machine code implementation of the REPL.)

```
loop $(bs, b)$ $input$ =
  case lex_until_toplevel_semicolon $input$ of
  | None → Terminate
  | Some $(tokens, rest\_of\_input)$ →
    case REPL$_i$_step$(tokens, s)$ of
    | Failure $error\_msg$ →
        Result $error\_msg$
          (loop $(bs, s)$ $rest\_of\_input$)
    | Success $(code, s, s_{exc})$ →
        let $bs$ = install_code $s$.cstate $code$ $bs$ in
          case bc_eval $bs$ of
          | None → Diverge
          | Some $bs$ →
              let $s$ = if success $bs$ then $s$ else $s_{exc}$ in
                Result $bs$.output
                  (loop $(bs, s)$ $rest\_of\_input$)

REPL$_i$ $input$ = loop initial_state $input$
```

On each iteration of its main loop REPL$_i$ lexes part of the input string up until the first top-level semicolon; it then calls the REPL$_i$_step function, which performs parsing, elaboration, type inference and then compilation to bytecode. Once the input has been turned into code, it installs the code into a bytecode state and starts execution (bc_eval) of the new bytecode state. If the generated code terminates, then it loops back to the top and starts again. Here bc_eval is a non-computable function which describes the small-step execution of the bytecode semantics. This function is total and returns None if the bytecode fails to terminate, otherwise it returns the resulting state inside Some.[6]

Theorem 14 (REPL$_i$ Correct)**.** For all inputs i and outputs o, if REPL$_i$ $i = o$ then REPL$_s$ (get_type_error_mask o) i o. Where REPL$_s$ is the REPL semantics described in Section 2 and get_type_error_mask picks out the inputs that output `"<type error>"`, to let the REPL semantics know where the type system should fail.

Proof sketch. By induction on the length of i. This theorem pulls together the parser soundness and completeness theorems, the inference soundness theorem, the type soundness theorem and the compiler correctness theorem. Most of the proof is devoted to showing that the various invariants required by the theorems are maintained across the iterations of loop. □

The verified REPL$_i$ function is well suited for simulating the implementation in the theorem prover: we can evaluate REPL$_i$ using the HOL4 prover's `EVAL` mechanism. For example,

```
EVAL ‘‘REPL$_i$ "fun f x = x + 3; f 2;"’’
```

automatically derives a theorem:

```
REPL$_i$ "fun f x = x + 3; f 2;" =
Result "val f = <fn>" (Result "val it = 5" Terminate)
```

Note that this evaluation of REPL$_i$ inside the logic by inference does not terminate if the bc_eval fails to terminate, *i.e.*, this evaluation won't return Diverge.

9. Bootstrapping

Most of the complexity in the definition of REPL$_i$ is contained within the definition of REPL$_i$_step, a function that combines parsing, elaboration, type inference and compilation to bytecode. The next section describes how we construct a verified x86-64 machine-code implementation of REPL$_i$, and therefore of REPL$_s$. In order to make the construction of x86-64 for REPL$_i$ an easier task, we use the verified compiler to compile REPL$_i$_step to bytecode. The REPL$_i$_step function contains the compiler itself, which means that this application of compilation amounts to *bootstrapping* the compiler. This section explains the bootstrapping method; the next section explains how we use the result.

Our starting point is REPL$_i$_step. We want to have bytecode which is proved to implement the REPL$_i$_step function. The REPL$_i$_step is defined in logic (HOL), where functions are specified using equations, *e.g.* it is easy to define a HOL constant, fac, that is characterised by the equations:

$$\begin{aligned} \mathsf{fac}\ 0 &= 1 \\ \mathsf{fac}\ (n+1) &= (n+1) \times \mathsf{fac}\ n \end{aligned}$$

In order to apply the verified compiler to functions defined in HOL, we need these equations to exists as CakeML abstract syntax for function definitions, *i.e.*, we need CakeML declarations defining REPL$_i$_step.

A previously developed proof-producing tool [22], implements exactly this kind of translation. Given a constant from HOL with equations that look sufficiently ML-like, the tool generates CakeML abstract syntax and proves that the generated CakeML implements the constant described by equations in HOL. When applied to REPL$_i$_step, this translation produces a long list of

[6] bc_eval is the $\Downarrow_{bc}$ relation from Section 7.

CakeML declarations:

$$\begin{array}{l}\mathsf{ml\_repl\_step\_decls} \;=\\ \quad [\texttt{datatype}\ \ldots,\ \texttt{datatype}\ \ldots,\\ \quad\ \texttt{fun}\ \ldots,\ \texttt{fun}\ \ldots,\ \ \ldots,\ \texttt{fun repl\_step x = }\ldots]\end{array}$$

The tool also proves two relevant theorems:

Theorem 15 (Evaluating ml_repl_step_decls)**.** Evaluation of the declaration list succeeds and produces a semantic environment, *i.e.*, a mapping from names to values. We will refer to this environment as ml_repl_step_env.

Theorem 16 (Evaluating `repl_step`)**.** The value associated with the name `repl_step` in ml_repl_step_env, is a closure which implements REPL_i_step w.r.t. refinement invariants input and output (relations between abstract values and CakeML semantic values). In the notation of Myreen and Owens [22]:

$$\begin{array}{c}\mathsf{Eval}\ \mathsf{ml\_repl\_step\_env}\ \texttt{"repl\_step"}\\ ((\mathsf{input} \to \mathsf{output})\ \mathsf{REPL}_i\mathsf{\_step})\end{array}$$

Proof sketch. By algorithm from Myreen and Owens [22] □

The CakeML-to-bytecode compiler is then applied to the declaration list. The result is a theorem produced by evaluation.

Theorem 17 (Compiling ml_repl_step_decls)**.** Given an initial compiler state init and the declaration list, the compiler produces bytecode and a new compiler state cstate.

$$\mathsf{compile\_decs\ init\ ml\_repl\_step\_decls} = (\mathsf{bytecode}, \mathsf{cstate})$$

Proof sketch. By evaluation in the logic (`EVAL` from above). □

Next, we instantiate the compiler correctness theorem (the compile_decs version of Theorem 8) for use with Theorems 15 and 16. Two further theorems are outlined below. We omit the definition of run_inv.

Theorem 18 (Running bytecode)**.** Executing the compiler generated bytecode terminates in some state bs such that this state contains ml_repl_step_env and run_inv $bs\ v$ is true, for some v.

Proof sketch. By instantiation of a lemma supporting Theorem 8. □

Theorem 19 (Calling `repl_step`)**.** For any bytecode state bs, semantics value v and x such that run_inv $bs\ v$ and input $x\ v$, running a snippet of bytecode which calls the code for `repl_step`, terminates in a new bytecode state bs' and v' such that run_inv $bs'\ v'$ and output $(\mathsf{REPL}_i\mathsf{\_step}\ x)\ v'$. In other words, calling the code for `repl_step` computes REPL_i_step.

Proof sketch. By instantiation of Theorem 12 and other lemmas supporting Theorem 8. □

10. Implementation in x86-64

The verified x86-64 implementation is constructed so that it exactly implements the in-logic-defined REPL_i function from above. Thanks to the bootstrapping, large parts of REPL_i, namely REPL_i_step, exist in the form of verified bytecode. In other words, much of the effort involved in constructing the x86-64 implementation of REPL_i boils down to producing an implementation of the bytecode that the compiler targets.

In order to verify the x86-64 code we need a semantics, a programming logic and proof automation for x86-64 machine code. For these, we build on previous work.

Semantics of x86-64 We use a conventional small-step operational semantics for x86 machine code. This semantics originates in a definition for 32-bit x86 that Sarkar et al. [28] validated against real hardware. The semantics was extended in Myreen [19] to handle self-modifying code and used in our previous work on a verified implementation of a REPL for Lisp [21]. The current semantics has a small coverage of the user-mode x86-64 instruction set. However, it tries to be as accurate as possible for the subset it describes.

Programming Logic For most proofs (manual and automatic), we use the machine-code Hoare logic for our previous work on self-modifying code [19]. However, for the current work, we had to define a new more expressive programming logic in order to state and prove the top-level theorem. The top-level theorem must be able to specify that the machine code diverges. Our previously developed Hoare logic is only able to specify that a state satisfying the postcondition is eventually reached (termination).

The new programming logic (also embedded in HOL), makes statements, using temporal logic, about the sequence of all possible executions of the underlying x86-64 machine. This new programming logic makes judgements of the form temporal $code\ \phi$. It is strictly more expressive than the Hoare triple.

Theorem 20 (Hoare Triple Instance of temporal)**.** The machine-code Hoare triple, $\{p\}\ code\ \{q\}$, is an instance of temporal:

$$\{p\}\ code\ \{q\} \iff \mathsf{temporal}\ code\ ((\mathsf{now}\ p) \Rightarrow \Diamond(\mathsf{now}\ q))$$

Proof sketch. Follows immediately from the definitions. □

We define temporal, now, $\Rightarrow$, $\Diamond$, *etc.* as follows. The definitions below take a few concepts from Myreen [19], in particular x86_seq $s\ t$ is true if t is a valid x86-64 trace starting from state s, and $p \preceq s$ is true if p holds for a copy of a state s with a less precise instruction cache [19]; for most purposes, simply read $p \preceq s$ as $p\ s$.

$$\begin{array}{l}\mathsf{temporal}\ code\ \phi \;=\\ \quad \forall s\ t\ r.\\ \quad\quad \mathsf{x86\_seq}\ s\ t \implies \phi\ (\lambda p\ s.\ (p * \mathsf{code}\ code * r) \preceq s)\ t\end{array}$$

$$\begin{array}{rcl}\mathsf{now}\ p &=& \lambda assert\ t.\ assert\ p\ (t\ 0)\\ \Box\,\phi &=& \lambda assert\ t.\ \forall k.\ \phi\ assert\ p\ (\lambda n.\ t\ (n+k))\\ \Diamond\,\phi &=& \lambda assert\ t.\ \exists k.\ \phi\ assert\ p\ (\lambda n.\ t\ (n+k))\\ \mathsf{later}\ \phi &=& \lambda assert\ t.\ \exists k.\ \phi\ assert\ p\ (\lambda n.\ t\ (n+k+1))\\ \phi \vee \psi &=& \lambda assert\ t.\ \phi\ assert\ t \vee \psi\ assert\ t\\ \phi \Rightarrow \psi &=& \lambda assert\ t.\ \phi\ assert\ t \implies \psi\ assert\ t\end{array}$$

Using these, we can specify divergence of a machine-code program. For example, from initial configuration p, *code* will always, at some some later point, reach a state satisfying q. In other words, q will be true infinitely many times.

$$\mathsf{temporal}\ code\ ((\mathsf{now}\ p) \Rightarrow \Box\,\Diamond(\mathsf{now}\ q))$$

In our theorems, we instantiate q to say that the compiled bytecode is still running. Divergence means that the program runs bytecode forever.

Bytecode Heap Invariant Central to our proofs is the invariant which specifies how bytecode states are concretely represented in the x86-64 machine state. This invariant is formalised as a state assertion bc_heap $bs\ aux\ s$, which relates bytecode state bs and auxiliary state aux to (part of) machine state s.

The formal definition of bc_heap is not shown due to its length. However, informally, this invariant states that:

- memory is split into two data heaps (of which only one is in use at any point in time, enabling stop-and-copy garbage collection), a code heap, the normal x86-64 stack and a separate global state record;
- registers `rax`–`rdx` hold bytecode values – in the case of a Block, a large (bignum) Number, or a RefPtr, this means they contain a pointer into the data heap;
- the top of the bytecode stack is stored in register `rax`,

- the rest of the bytecode stack is kept in the x86-64 stack, *i.e.*, all values in the x86-64 stack are roots for the garbage collector,
- the stack is accessed through the normal stack and base pointers, registers `rsp` and `rbp`;
- other registers and state keep track of temporary values, the state of the allocator and system configuration.
- output is produced via calls to a special code pointer, for which we have an assumption that each call to this code pointer puts a character onto some external stream (in practice we link to C's `putc` routine). Input is handled similarly (using `getc`).
- memory contains code for supporting routines: the verified garbage collector, arbitrary-precision arithmetic library *etc.*

The garbage collector updates the heap and the stack (*i.e.*, the roots for the heap), both of which can contain code pointers and stack pointers. In order for the garbage collector to distinguish between data pointers and code/stack pointers all code/stack pointers must have zero as the least significant bit (*i.e.*, appear to be small integers). We ensure that all code pointers end with zero as the least significant bit by making sure that each bytecode instruction is mapped into x86-64 machine code that is of even length.

Implementation of CakeML Bytecode Having formalised the representation of bytecode states, we define a function that maps CakeML Bytecode instructions into concrete x86-64 machine instructions (*i.e.* lists of bytes). Here i is the index of the instruction that is to be translated (i is used for the translation of branch instructions, such as Jump).

$$\begin{array}{rcl} \mathsf{x64}\ i\ (\mathsf{Stack\ Pop}) & = & [\mathtt{0x48}, \mathtt{0x58}] \\ \mathsf{x64}\ i\ (\mathsf{Stack\ Add}) & = & [\mathtt{0x48}, \ldots] \\ & \vdots & \end{array}$$

Entire bytecode programs are translated by $\mathsf{x64\_code}$:

$$\begin{array}{rcl} \mathsf{x64\_code}\ i\ [\,] & = & [\,] \\ \mathsf{x64\_code}\ i\ (x :: xs) & = & \mathsf{let}\ c = \mathsf{x64}\ i\ x\ \mathsf{in} \\ & & c \mathbin{@} \mathsf{x64\_code}\ (i + \mathsf{length}\ c)\ xs \end{array}$$

We prove a few key lemmas about the execution of the generated x86-64 machine code.

Theorem 21 (x64_code Implements Bytecode Steps)**.** The code generated by $\mathsf{x64\_code}$ is faithful to the execution of each of the CakeML Bytecode instructions. Each instruction executes at least one x86-64 instruction (hence later). Note that execution must either reach the target state or resort to an error ($\mathsf{out\_of\_memory\_error}$).

$$\begin{array}{l} bs \to bs' \implies \\ \mathsf{temporal}\ \{(base, \mathsf{x64\_code}\ 0\ bs.\mathsf{code})\} \\ \quad (\mathsf{now}\ (\mathsf{bc\_heap}\ bs\ (base, aux)) \Rightarrow \\ \quad\quad \mathsf{later}\ (\mathsf{now}\ (\mathsf{bc\_heap}\ bs'\ (base, aux)) \\ \quad\quad\quad \vee\ \mathsf{now}\ (\mathsf{out\_of\_memory\_error}\ aux))) \end{array}$$

Proof sketch. For simple cases of the bytecode step relation ($\to$), the proof was manual using the programming logic from Myreen [19]. More complex instruction snippets (such as the supporting routines) were produced using a combination of manual proof and proof-producing synthesis (*e.g.* [20]). □

Theorem 22 (x64_code Implements Terminating Bytecode Executions)**.** Same as the theorem above, but with $\Downarrow_{\mathsf{bc}}$ instead of $\to$.

Proof sketch. Induction on the number of steps. □

Theorem 23 (x86-64 Implementation of $\mathsf{REPL}_i\mathsf{\_step}$)**.** Executing the x64_code-generated code for the result of the bootstrapping (*i.e.* bytecode) and the bytecode snippet that calls `repl_step` has the desired effect w.r.t. bc_heap.

Proof sketch. Follows from theorems 18, 19 and 22. □

The only source of possible divergence in our x86-64 implementation of REPL_i is the execution performed by bc_eval. When the logic function bc_eval returns None, we want to know that the underlying machine gets stuck in an infinite loop and that the output stays the same. (Only the top-level loop is able to print output.)

$$\begin{array}{l} \mathsf{repl\_diverged}\ out\ aux\ = \\ \quad \Box\,\Diamond(\mathsf{now}\ (\exists bs.\ \mathsf{bc\_heap}\ bs\ aux * (bs.\mathsf{output} = out))) \end{array}$$

Theorem 24 (x86-64 Divergence)**.** For any bs, such that $\mathsf{bc\_eval}\ bs = \mathsf{None}$, we have:

$$\begin{array}{l} (\forall bs'.\ bs \to^* bs' \implies bs.\mathsf{output} = bs'.\mathsf{output}) \implies \\ \mathsf{temporal}\ \{(base, \mathsf{x64\_code}\ 0\ bs.\mathsf{code})\} \\ \quad (\mathsf{now}\ (\mathsf{bc\_heap}\ bs\ (base, aux)) \Rightarrow \\ \quad\quad \mathsf{later}\ (\mathsf{repl\_diverged}\ bs.\mathsf{output}\ aux) \\ \quad\quad\quad\quad \vee\ \mathsf{now}\ (\mathsf{out\_of\_memory\_error}\ aux))) \end{array}$$

Proof sketch. Theorem 21 and temporal logic. □

Top-level Correctness Theorem The top-level theorem for the entire x86-64 implementation is stated as follows.

Theorem 25 (x86-64 Implementation of REPL_s)**.** If the state starts from a good initial state (init), then execution behaves according to $\mathsf{REPL}_s\ l$ for some list l of type inference failures.

$$\begin{array}{l} \mathsf{temporal\ entire\_machine\_code\_implementation} \\ \quad (\mathsf{now}\ (\mathsf{init}\ inp\ aux) \Rightarrow \\ \quad\quad \mathsf{later}\ ((\exists l\ res.\ \mathsf{repl\_returns}\ (\mathsf{out}\ res)\ aux\ \wedge \\ \quad\quad\quad\quad\quad (\mathsf{REPL}_s\ l\ inp\ res \wedge \mathsf{terminates}\ res)) \\ \quad\quad\quad \vee \\ \quad\quad\quad (\exists l\ res.\ \mathsf{repl\_diverged}\ (\mathsf{out}\ res)\ aux\ \wedge \\ \quad\quad\quad\quad\quad (\mathsf{REPL}_s\ l\ inp\ res \wedge \neg\mathsf{terminates}\ res)) \\ \quad\quad\quad \vee \\ \quad\quad\quad \mathsf{now}\ (\mathsf{out\_of\_memory\_error}\ aux))) \end{array}$$

Here repl_returns states that control is returned to the return pointer of aux, and out and terminates are defined as follows.

$$\begin{array}{l} \mathsf{out\ Terminate}\ =\ \texttt{""} \\ \mathsf{out\ Diverge}\ =\ \texttt{""} \\ \mathsf{out}\ (\mathsf{Result}\ str\ rest)\ =\ str \mathbin{@} \mathsf{out}\ rest \\ \\ \mathsf{terminates\ Terminate}\ =\ \mathsf{true} \\ \mathsf{terminates\ Diverge}\ =\ \mathsf{false} \\ \mathsf{terminates}\ (\mathsf{Result}\ str\ rest)\ =\ \mathsf{terminates}\ rest \end{array}$$

Proof sketch. The execution of bytecode is verified as sketched above. The other parts of the x86-64 implementation (the setup code, the lexer and the code that ties together the top-level loop) was verified, again, using a combination of manual Hoare logic reasoning and proof-producing synthesis. Theorem 14 was used to replace REPL_i by the top-level specification REPL_s. □

11. Small Benchmarks

To run the verified x86-64 machine code, we inline the code into a 30-line C program, which essentially just allocates memory (with execute permissions enabled) then runs it (passing in function pointers for `getc` and `putc`).

The result of running a few benchmarks is shown below. Execution times are compared with interpreted OCaml: CakeML runs the Fibonacci example 2.2 times faster than interpreted OCaml.

	compiled OCaml	Poly/ML	CakeML
Fibonacci	7.9	4.6	2.2
Quicksort	3.1	10.0	0.6
Batched queue	2.0	12.9	0.4
Binary tree	4.3	5.6	0.6

The Fibonacci benchmark computes the 31st Fibonacci number using the naïve recursive definition; the second benchmark applies Quicksort to a list of length 2,000; the batched queue benchmark performs enqueue-enqueue-dequeue 1,000,000 times on a purely functional implementation of queues; the last benchmark constructs a 2,000-element binary tree and then flattens it. We used OCaml version 4.01 and Poly/ML 5.4.1.

12. Design and Context

Our overall goal for CakeML is to provide the most secure system possible for running verified software and other programs that require a high-assurance platform. Thus, our primary focus has been on reducing the trusted computing base, rather than on compiler optimisations or exotic language features.

12.1 Trusted Computing Base

Our correctness theorem relies on a machine code semantics for x86-64 and on a semantics for CakeML. If the machine code semantics does not soundly reflect actual processor behaviour, then the program might not behave as verified. Having a machine code semantics in the trusted computing base is intrinsic to the problem. The only alternative is to restrict execution to a verified processor, severely limiting the usefulness of the verified compiler compared to one that targets off-the-shelf hardware. However, the target machine code semantics only needs to cover, and be tested on, the instructions that the compiler actually generates, which in our case is significantly smaller than the entire x86-64 ISA.

If a programmer wants to understand what their CakeML program does, they currently have two strategies: one, is to reason about it relative to the CakeML semantics, and the other is to synthesise verified CakeML from higher-order logic (*e.g.*, using the same technique [22] that we use for bootstrapping). In the latter case, the CakeML semantics is not part of the trusted computing base, because the synthesised CakeML is verified with respect to the CakeML semantics. In this sense, the CakeML semantics is just the interface between the compiler, and the user's chosen verification approach. In the future, we hope to implement further (verified) ways to verify CakeML code, in the spirit of the Princeton "verified software toolchain" [1], which takes the same viewpoint, but for a C-like source language.

We also trust a small amount of code to set up the initial execution environment with functions to get a character from standard input and to write a character to standard output, because our machine model does not include these features.

A theorem prover or checker is also intrinsically part of the trusted computing base. In practice, our proofs can only be checked by automated means: too much of their content is in the minute details. Furthermore, the bootstrapping and machine code synthesis steps use verified computation in the prover itself to create the verified code: the proofs generated here are not human readable. One could apply a separate proof checking tool (*e.g.*, OpenTheory [9]), or simply trust HOL4 which follows the LCF approach and relies for its own soundness only on a small ($\approx$1000 lines) trusted kernel.

Lastly, we note that we do not rely on the correctness of another compiler in our trusted computing base (except perhaps as part of the proof checker or theorem prover).

12.2 Other Targets, Other Sources

CakeML currently translates from an SML-like language to x86-64 machine code; however, neither of those choices are mandated by our approach. In the future, we hope to extend CakeML to generate ARM machine code as well. Because CakeML compiles to a low-level, machine independent bytecode, and then to machine code, retargeting it only requires introducing a new bytecode to machine code level. This means that the amount of effort required to get a verified compiler for a second platform is much smaller than the effort required to build the system in the first place. Even though going through a machine-independent bytecode can potentially limit the compiler's ability to generate optimal code, we consider it well worth it in this context.

It would take more effort to adapt CakeML to new source languages: any of the lexer, parser, type checker and compilation to intermediate language (or even bytecode, if the source language is different enough) might have to change. However, this work would not be of a different character – it would just be in re-doing proofs of different parsing/typechecking/etc. algorithms following the strategy laid out here. In particular, the use of clocks (Section 7) to handle divergence and maintaining equivalent small-step and big-step semantics will be helpful in building a verified compiler for any similar language.

What about OCaml? For the features that CakeML supports, SML and OCaml are very similar, and CakeML follows OCaml in several ways including the capitalisation restrictions on variables and constructors, and our lack of equality types and overloading. However, OCaml lacks two things that are important to our development: deterministic evaluation order and an equality function that always terminates (OCaml's follows reference cells into their contents, instead of just comparing them for identity). Thus, in order to be a semantic subset of an existing language, SML is our only choice. However, retargeting CakeML to an OCaml-like syntax (albeit with these small differences) would just be a matter of providing a new lexer and parser.

13. Related Work

The Jitawa verified Lisp implementation [21] has a similar goal of end-to-end verification about the compiler running as machine code. However, the source language of Jitawa is simpler than ours, with much simpler parsing, no type system, no modules, no handleable exceptions, no pattern matching, and no higher-order functions. Thus, CakeML demonstrates a substantial scaling-up of the kind of language that can be supported in a very high assurance setting. Furthermore, Jitawa does not bootstrap itself, and its top-level correctness theorem assumes that every input program terminates.

The CompCert compiler [14] and projects based on it, including CompcertTSO [29] and the Princeton "verified software toolchain" [1], focus on a C-like source language in contrast to our ML-like language. The emphasis is variously on optimisations, concurrency and program logics. Whereas our emphasis is on end-to-end correctness and a very small trusted computing base.

The VLISP project [7], which produced a *rigorously* – not formally – verified implementation of Scheme, emphasised end-to-end verification and did bootstrap their Scheme compiler to produce a verified implementation of their compiler.

Chlipala [4] is the most closely related work on verification of compiling impure functional programs with higher-order functions. Chlipala's compiler has simpler source and target languages, and its proofs do not address divergence. He emphasises the use of parametric higher-order abstract syntax (PHOAS) instead of the more conventional substitution semantics. For CakeML, we made the closure environments explicit in order to keep things simple.

Turning to the front end, there has been significant focus on mechanised meta-theory for language researchers (*e.g.*, POPLMark [2]), but it has not typically focussed on fitting into the overall context of a verified compiler. For type inference, Naraschewski and Nipkow [25] verify algorithm $\mathcal{W}$ for a basic typed λ-calculus plus let expressions. Our overall approach is similar to theirs, but they verify completeness and generalise nested lets, whereas we have a much richer language. Our approach to type soundness is similar to OCaml light [26], which also uses a concrete representation for ordinary variables and de Bruijn indices for type variables. The languages supported are also similar, except that they support type abbreviations whereas we support a module system. They also use a substitution-based semantics. Two other notable formalisations of ML metatheory by Lee at al. [13] and by Garrigue [6] focus on sophisticated type system features (functors and structural polymorphism, respectively).

Verified parsing has been a productive area recently. Our work distinguishes itself in its combination of soundness, completeness, guaranteed termination, and relative efficiency. Jourdan *et al.* [10] validate LR automata generated from a CFG, proving soundness and completeness. However, they have to provide a "fuel" parameter in order to ensure that the automata's execution terminates. Ridge [27] verifies a sound, complete and terminating parsing algorithm for arbitrary CFGs. As the input grammars may be ambiguous, Ridge's algorithm returns a list of parses. It is also rather inefficient, potentially $O(n^5)$. Earlier still, work by Barthwal and Norrish [3] achieves soundness and completeness, but is again missing termination. The big difference between these approaches and our own is that our approach is not *generic* as theirs are. Our PEG was hand-crafted, and its proofs of soundness and completeness with respect to the CFG were done manually.

14. Conclusion

The primary aim of CakeML is to provide an implementation of a practical programming language running on off-the-shelf hardware with the highest-levels of security and trustworthiness. We hope that it will be used as a platform for the development and deployment of programs where their correctness is the most important concern. Thus the trade-offs we have made in the design of the project differ from other efforts along the same lines. In particular, our focus has been on minimising the trusted computing base, not on optimisation or on breadth of source language features. In this sense, we believe that CakeML complements the verification efforts based around CompCert, which *are* focussed on optimisation and mainstream programming languages.

However, the design of CakeML does not rule out source- or IL-level optimisations, such as good support for multiple argument functions, inlining, constant propagation, and lambda lifting; the interesting question will be how to integrate them into the existing verification without requiring unreasonable effort. Furthermore, it does not appear difficult to add lower level optimisations with only modest changes to CakeML Bytecode, for example, the addition of registers and some form of register allocation.

Lastly, a verified compiler is most important in the context of verified applications to run on it. We are already working toward one example — a theorem prover using CakeML as its execution environment [24] — and hope that others will join in with applications drawn from a variety of domains.

References

[1] A. W. Appel. Verified software toolchain (invited talk). In *ESOP*, volume 6602 of *LNCS*, 2011.

[2] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLMark challenge. In *TPHOLs*, volume 3603 of *LNCS*, 2005.

[3] A. Barthwal and M. Norrish. Verified, executable parsing. In *ESOP*, volume 5502 of *LNCS*, 2009.

[4] A. Chlipala. A verified compiler for an impure functional language. In *POPL*, 2010.

[5] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[6] J. Garrigue. A certified implementation of ML with structural polymorphism. In *APLAS*, volume 6461 of *LNCS*, 2010.

[7] J. Guttman, J. Ramsdell, and M. Wand. VLISP: A verified implementation of Scheme. *Lisp and Symbolic Computation*, 8(1/2):5–32, 1995.

[8] HOL4. `http://hol.sourceforge.net`.

[9] J. Hurd. The OpenTheory standard theory library. In *NASA Formal Methods*, volume 6617 of *LNCS*, 2011.

[10] J.-H. Jourdan, F. Pottier, and X. Leroy. Validating LR(1) parsers. In *ESOP*, volume 7211 of *LNCS*, 2012.

[11] A. Koprowski and H. Binsztok. TRX: A formally verified parser interpreter. *Logical Methods in Computer Science*, 7(2), 2011.

[12] R. Kumar and M. Norrish. (Nominal) Unification by recursive descent with triangular substitutions. In *ITP*, volume 6172 of *LNCS*, 2010.

[13] D. K. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of Standard ML. In *POPL*, 2007.

[14] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7), 2009.

[15] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2), 2009.

[16] A. McCreight, T. Chevalier, and A. P. Tolmach. A certified framework for compiling and executing garbage-collected languages. In *ICFP*, 2010.

[17] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3), 1978.

[18] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[19] M. O. Myreen. Verified just-in-time compiler on x86. In *POPL*, 2010.

[20] M. O. Myreen and G. Curello. Proof pearl: A verified bignum implementation in x86-64 machine code. In *CPP*, volume 8307 of *LNCS*, 2013.

[21] M. O. Myreen and J. Davis. A verified runtime for a verified theorem prover. In *ITP*, volume 6898 of *LNCS*, 2011.

[22] M. O. Myreen and S. Owens. Proof-producing synthesis of ML from higher-order logic. In *ICFP*, 2012.

[23] M. O. Myreen, K. Slind, and M. J. C. Gordon. Extensible proof-producing compilation. In *CC*, volume 5501 of *LNCS*, 2009.

[24] M. O. Myreen, S. Owens, and R. Kumar. Steps towards verified implementations of HOL Light. In *ITP*, volume 7998 of *LNCS*, 2013.

[25] W. Naraschewski and T. Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. *J. Autom. Reasoning*, 23(3-4), 1999.

[26] S. Owens. A sound semantics for OCaml light. In *ESOP*, volume 4960 of *LNCS*, 2008.

[27] T. Ridge. Simple, functional, sound and complete parsing for all context-free grammars. In *CPP*, volume 7086 of *LNCS*, 2011.

[28] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *POPL*, 2009.

[29] J. Ševcík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. In *POPL*, 2011.

[30] D. Vytiniotis, S. L. Peyton Jones, T. Schrijvers, and M. Sulzmann. OutsideIn(X) Modular type inference with local assumptions. *J. Funct. Program.*, 21(4-5), 2011.

[31] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1), 1994.