Journal of Logical and Algebraic Methods in Programming 112 (2020) 100530

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

A verified proof checker for higher-order logic

Oskar Abrahamsson [1]

Chalmers University of Technology, Department of Computer Science and Engineering, SE-412 96 Göteborg, Sweden

ARTICLE INFO

Article history:
Received 1 March 2019
Received in revised form 28 October 2019
Accepted 10 January 2020
Available online 17 February 2020

Keywords:
Proof checker
Higher-order logic
Mechanized proof
Soundness

ABSTRACT

We present a computer program for checking proofs in higher-order logic (HOL) that is verified to accept only valid proofs. The proof checker is defined as functions in HOL and synthesized to CakeML code, and uses the Candle theorem prover kernel to check logical inferences. The checker reads proofs in the OpenTheory article format, which means proofs produced by various HOL proof assistants are supported. The proof checker is implemented and verified using the HOL4 theorem prover, and comes with a proof of soundness.

1. Introduction

This paper is about a verified proof checker for theorems in higher-order logic (HOL). A proof checker is a computer program which takes a logical conclusion together with a proof object representing the steps required to prove the conclusion, and returns a verdict whether or not the proof is valid.

Our checker is designed to read proof objects in the OpenTheory article format [1]. OpenTheory articles contain instructions on how to construct types, terms and theorems of HOL from previously known facts. The tool starts with the axioms of higher-order logic as its facts, and uses a previously verified implementation of the HOL Light kernel (called Candle) [2] to carry out all logical inferences. If all commands are successfully executed, the tool outputs a list of all proven theorems together with the logical context in which they are true.

The proof checker is implemented as a function (shallow embedding) in the logic of the HOL4 theorem prover [3]. We verify the correctness of the proof checker function, and prove a soundness theorem. This theorem in the HOL4 system guarantees that any theorem produced as a result of a successful run of the tool is a theorem in HOL.

Using a proof-producing synthesis mechanism [4] we synthesize a CakeML program from the shallow embedding. The resulting program is compiled to executable machine code using the CakeML compiler. Compilation is carried out completely within the logic of HOL4, enabling us to combine our soundness result with the end-to-end correctness theorem of the CakeML compiler [5]. This gives a theorem that guarantees that the proof checker is sound down to the machine code that executes it.

Contributions. In this work we present a verified proof checker for HOL. To the best of our knowledge, this is the first verified implementation of a proof checker for HOL. As a consequence of using the CakeML tools, we are able to obtain a correctness result about the executable machine code that is the proof checker program.

E-mail address: aboskar@chalmers.se.

[1] Partly supported by the Swedish Foundation for Strategic Research.

https://doi.org/10.1016/j.jlamp.2020.100530

Overview. To reach this goal we require:

(i) a file format for proof objects in HOL for which there exists sample proofs;
(ii) tool support for reasoning about the correctness of the actual implementation of our proof checker (as opposed to a *model*); and
(iii) a convincing way of connecting the correctness of the proof checker implementation with the *machine code* we obtain when compiling it.

We address (i) by using the OpenTheory framework [1]. Although originally designed with theory sharing between theorem provers in mind, the framework includes a convenient format for storing proofs, as well as a library of theorems.

The issue (ii) is tackled by implementing our proof checker in a computable subset of the HOL4 logic. In this way we are able to draw precise conclusions about the correctness of our program without the overhead of a program logic. Additionally, the implementation of the Candle theorem prover kernel [2] and its soundness proof lives in HOL4: we can use this result directly, as opposed to assuming it.

Finally, (iii) is addressed using the CakeML compiler toolchain. The CakeML toolchain can produce executable machine code from shallow embeddings of programs in HOL4. The compilation is proof-producing, and yields a theorem which states the correctness of the resulting machine code in terms of the logical functions from which it was synthesized. Consequently, any statement about the logical specification can be made into a statement about the machine code that executes it.

We start by introducing the OpenTheory framework, the CakeML compiler and the Candle theorem prover kernel (§2). We then explain, at a high level, the steps required to produce the proof checker implementation and verify its correctness (§3).

We show the details of the implementation (and specification) of the tool as a shallow embedding in the logic (§4), and how this shallow embedding is automatically refined into an equivalent CakeML program using a proof-producing synthesis procedure (§5).

We compile the synthesized program into machine code, and obtain a correctness theorem relating the machine code with the shallow embedding (§7). Following this, we state a theorem describing end-to-end correctness (soundness) of the proof checker, and describe how the proof is carried out using the existing soundness result of the Candle kernel (§8).

Finally, we comment on the results of running the checker on a collection of article files, and compare its execution time to that of an existing (unverified) tool implemented in Standard ML (§9).

Notation. Throughout this paper we use `typewriter font` for listings of ML program code, and sans-serif for constants and *italics* for variables in higher-order logic. The double implication $\iff$ stands for equality between boolean terms, and all other logical connectives (e.g. $\Rightarrow$, $\wedge$, $\vee$, $\neg$, $\ldots$) have their usual meanings.

2. Background

In this section we introduce the tools and concepts used in the remainder of this paper.

2.1. The OpenTheory framework

The purpose of the OpenTheory framework [1] is to facilitate sharing of logical theories between different interactive theorem provers (ITPs) that use HOL as their logic. Several such systems exist; e.g. HOL4 [3], HOL Light [6], ProofPower-HOL [7]. Although the logical cores of these tools coincide to some degree, the systems built around the logics (e.g. theory representation, and storage) are very different.

The aim of OpenTheory is to reduce the amount of duplicated effort when developing theories in these systems. It attempts to do so by defining:

- a version of HOL contained within the intersection of the logics of these tools, and
- a file format for storing instructions on how to construct definitions and theorems in this logic.

Collections of type- and constant definitions, terms and theorems are bundled up into *theories*, and instructions for reconstructing theories are recorded in OpenTheory *articles*. An OpenTheory article is a text file consisting of a sequence of commands corresponding to primitive inferences and term constructors/destructors of HOL.

Article files are usually produced by instructing a HOL theorem prover to record all primitive inferences used in the construction of theorems. In order to reconstruct the theory information, the OpenTheory framework defines an abstract machine that operates on article files. The machine interprets article commands into calls to a logical kernel, which in turn reconstructs the theory elements.

We have constructed our proof checker to read input represented in the OpenTheory article format. Our proof checker is a HOL function that is a variation on the OpenTheory abstract machine. In particular, we have left the machine without its built-in logical kernel, and let the Candle theorem prover kernel perform all logical reasoning.

2.2. The Candle theorem prover kernel

The Candle theorem prover kernel is a verified implementation of the HOL Light logical kernel by Kumar et al. [2]. The kernel is implemented as a collection of monadic functions [8] in a state-and-exception monad in the logic of the HOL4 theorem prover, and is proven sound with respect to a formal semantics which builds on Harrison's formalization of HOL Light [9].

As discussed in §2.1, we will use the Candle theorem prover kernel to execute all logical operations in our proof checker. Clearly, the main advantage of using the Candle kernel over implementing our own is its soundness result, which guarantees the validity of all HOL inferences executed by the kernel.

We return to Candle in §4, where we explain how our proof-checker is constructed on top of the Candle kernel; and in §8, where we show how to utilize its soundness result when verifying the end-to-end correctness of our checker.

2.3. The CakeML ecosystem

CakeML is a language in the style of Standard ML [10] and OCaml [11]. The language has a formal semantics, and supports most features familiar from Standard ML, such as references, I/O and exceptions.

The CakeML ecosystem consists of:

(i) the CakeML language and its formal semantics;
(ii) the end-to-end verified CakeML compiler, which can be run inside HOL;
(iii) tools for generating and reasoning about CakeML programs.

The CakeML compiler is an optimizing compiler for the CakeML language. The compiler backend supports code generation for multiple targets, including 32- and 64-bit flavors of Intel and ARM architectures, RISC-V and MIPS. The compiler is formally verified to produce machine code that is semantically compatible with the source program it compiles [5]. The compiler implementation, execution and verification is carried out completely within the logic of the HOL4 theorem prover.

Using the proof-producing synthesis mechanism of the CakeML ecosystem [4] together with the CakeML compiler's top-level correctness theorem, the system produces a theorem relating the resulting executable machine code with its logical specification. This enables us to extract useful, verified programs from logical functions in HOL4.

In §5 we show how we use the CakeML toolchain to synthesize a CakeML program from the logical specification of our proof checker; in §7 this program is compiled to machine code.

3. High-level approach

There are several parts involved in our proof checker development; a framework for storing logical theories (§2.1), a verified theorem prover kernel (§2.2), and a verified compiler (§2.3). In this section we explain, at a high level, how these parts come together into a verified program for checking HOL proofs.

Our program implementation consists chiefly of functions within the HOL4 logic, because this simplifies verification greatly. The CakeML compiler, on the other hand, operates on CakeML abstract syntax. Consequently, we must first move from logical functions to CakeML syntax; and finally, to executable machine code. Furthermore, the compilation is carried out *within* the logic of the theorem prover.

3.1. Terminology: levels of abstraction

There are clearly several layers of abstraction involved. Here is the terminology we will use:

- the *definition* of the OpenTheory abstract machine,
- a *shallow embedding* which implements the definition,
- a *deep embedding* that is a refinement of the shallow embedding, and
- the *machine code* which is obtained from compiling the deep embedding.

The shallow embedding is a function in the logic of HOL4. The deep embedding is CakeML abstract syntax synthesized from the shallow embedding. This abstract syntax is represented as a datatype in the logic. Finally, the machine code is a sequence of bytes which can be linked to produce an executable that runs the proof-checker.

3.2. Overview of steps

We now turn to an overview of the steps we take to produce the verified proof checker:

A.1 We begin by constructing a *shallow embedding* from the *definition* of the OpenTheory abstract machine. The *shallow embedding* is a monadic function in the logic of HOL4. As previously mentioned in §2.1, the logical kernel is left out;

what is left is a machine that performs bookkeeping of theory data (i.e. theorems, constants and types). The actual work of logical reasoning is left to the verified Candle kernel.

Concretely, we achieve this by implementing our *shallow embedding* in the same state-and-exception monad as the Candle logical kernel. In this way we are able to include the Candle kernel implementation as part of our program.

A.2 We synthesize *deeply-embedded* CakeML code from the *shallow embedding* of Step A.1 using a proof-producing mechanism. As a result of this synthesis we obtain a certificate theorem stating that the *deep embedding* is a refinement of the *shallow embedding*.

A.3 We prove a series of invariants for the *shallow embedding*. These invariants are needed in order to make use of the main soundness theorem of the Candle theorem prover. We will return to the details of these invariants in §8.

A.4 Using the existing Candle soundness theorem, we prove that any valid sequent produced by a successful run of the *shallowly embedded* proof checker is in fact true by the semantics of HOL. With the aid of the certificate theorems from A.2, we are able to conclude that the same holds for the *deeply-embedded* CakeML program.

A.5 Finally, the CakeML compiler is used to compile the *deep embedding* from A.2 into executable *machine code*. The compilation is carried out completely within the HOL4 logic, and produces a theorem that the *machine code* is compatible with the *deep embedding*. By combining this theorem with the results from A.2 and A.3, we obtain a theorem asserting that the *machine code* is a refinement of *shallow embedding* from A.1.

Finally, we connect the theorems from parts A.3 and A.5. The result is a theorem establishing soundness for the *machine code* that executes our proof checker.

Before we can describe the final end-to-end correctness theorem (§8), we will describe the OpenTheory abstract machine (§4), how we synthesize code from the shallow embeddings (§5), extend our program with verified I/O capabilities (§6), and finally, compile it to machine code (§7).

4. The OpenTheory abstract machine

The OpenTheory framework defines a file format (*articles*) for storing logical theories, and an abstract machine for extracting theories from such files. In this section we describe the operation of the abstract machine, and explain how we construct a shallow embedding in the HOL4 logic which implements it.

The OpenTheory machine is a stack-based abstract machine, which constructs types, terms and theorems of HOL by executing *commands* that update the machine state in various ways. Its operation is as follows. Commands are read from the input (a proof article), and interpreted into one of two types of actions:

(i) logical operations, such as inferences, constructor- or destructor applications on logical syntax; or
(ii) commands used to organize the machine state in various ways, such as stack and other data structures.

At any time during the run of the machine, theorems and definitions may be finalized by committing them to a special store. Once finalized, these theorems are never touched again.

4.1. Machine state

The state maintained by the machine during execution is the following:

- A stack of *objects*. We shall describe these objects shortly, but they include e.g. terms and types of HOL. The stack is the primary source of input (and destination for output) of commands.
- A dictionary, mapping natural numbers to objects. The dictionary enables persistent storage of objects that would otherwise be consumed by stack operations.
- A special stack dedicated to storing exported theorems. Once the production of a theorem is complete, it is pushed onto the theorem stack. Once there, it cannot be manipulated any further.
- A list of external assumptions on the logical context in which theorems are checked. Concretely, these assumptions are logical statements taken as axioms during the run of the machine, allowing for some modularity in theory reconstruction. For technical reasons, we leave this part out of our implementation; see §10 for further discussion.

We construct the record type state to represent the machine state. Here stack, dict and thms represent the aforementioned object stack, dictionary, and theorem stack, respectively. We also store a number linum for reporting the current position in the article file in case of error.

```
state = ⟨|
  stack : object list;
  dict : object num_map;
  thms : thm list;
  linum : int
|⟩
```

```
object =
  Num int
 | Name string
 | List (object list)
 | TypeOp string
 | Type type
 | Const string
 | Var (string × type)
 | Term term
 | Thm thm
```

Fig. 1. The type of OpenTheory objects. Those commands executed by the OpenTheory machine that take inputs and/or produce results use the type object.

4.2. Objects

All commands in the OpenTheory machine read input from the stack. Different commands accept different types of input, ranging from integer- and string literals, to terms of HOL. We unify these types under a datatype called object. See Fig. 1 for the definition of object.

In summary, the type object is made up of:

- syntactic elements of HOL (Type, Term, and Thm);
- references (by name) to variables and constants in HOL (Var and Const); and
- auxiliaries used in the construction of the above, such as lists and literals (List, Num, and Name).

4.3. Commands

Commands fetch input by popping object type elements from the stack. Those commands that produce results push these onto the stack.

As an example, consider the proof command called deductAntisym. The command deductAntisym pops two theorems (th_1 and th_2) from the stack, and calls on Candle to execute the inference rule DEDUCT_ANTISYM_RULE on these. Finally, the result is pushed back onto the stack.

Here is the definition of deductAntisym (using do-notation for monadic functions, which is familiar from Haskell):

```
deductAntisym s =
 do
  (obj,s) ← pop s;   th2 ← getThm obj;
  (obj,s) ← pop s;   th1 ← getThm obj;
  th ← DEDUCT_ANTISYM_RULE th1 th2;
  return (push (Thm th) s)
 od
```

Here, s (of type state) represents the state of the abstract machine. The internal commands pop and push are used for manipulating the object stack, and the function getThm extracts a value of type thm from an object with constructor Thm (or raises an exception otherwise). Finally, the machine executes the following primitive inference of HOL Light [6] on the theorems th_1 and th_2:

$$\frac{\Gamma \vdash p \quad \Delta \vdash q}{(\Gamma - \{q\}) \cup (\Delta - \{p\}) \vdash p = q} \; \texttt{DEDUCT\_ANTISYM\_RULE}$$

At the time of writing, there are 36 commands in the OpenTheory article format. For each proof command in the article format we implement the corresponding operation as a monadic HOL function. In addition, we implement some internal commands (such as push and pop above) to access and/or manipulate the machine state. For a complete listing of article commands and their semantics, see [12].

4.4. Wrapping up

Finally, we wrap our proof command specifications up into a function called readLine. The function readLine is the *shallow embedding* of the OpenTheory abstract machine. This function takes a machine state and a line of text (corresponding to a proof command) as input, and returns an updated state. If the execution of a command fails, an exception is raised and execution halts. The full definition of readLine is shown in Appendix A.

5. Proof-producing synthesis of CakeML

At this stage we have a shallow-embedded implementation of the OpenTheory abstract machine in HOL4 (see §4), together with the functions that make up the Candle theorem prover kernel. We apply a proof-producing synthesis tool [4] to the shallow embedding, and obtain the following:

- a deeply-embedded CakeML program, that can be compiled by the CakeML compiler; and
- a certificate theorem stating that the deep embedding (the program) is a refinement of the shallow embedding (the logical functions).

The certificate theorem produced by the synthesis mechanism is absolutely vital for the verification carried out in §8, as it eliminates the gap between the shallow- and deeply embedded views of the proof checker program (cf. §3). Using the certificate, we may turn any statement about the shallow embedding into a statement about the semantics of the deep embedding.

5.1. Refinement invariants

Before discussing the certificate theorem for our proof checker, we will take a step back and look at certificate theorems in general. This is the general shape of a certificate theorem produced by the proof-producing synthesis:

$$\vdash \mathsf{INV}\ x\ v$$

Here, INV is a relation stating that the deeply-embedded CakeML value v is a refinement of the shallow embedding x. We call INV a *refinement invariant*.

The CakeML tools define several refinement invariants for most basic types (integers, strings, etc.), as well as *higher-order* invariants; e.g. for expressing refinements of function types. Here is the invariant $\longrightarrow$, connecting the HOL function f and the CakeML function g:

$$\begin{array}{l} \vdash (\mathsf{A} \longrightarrow \mathsf{B})\ f\ g \\ \text{where the types are} \\ \quad f : \alpha\ \to\ \beta \\ \quad \mathsf{A} : \alpha\ \to\ \mathsf{v}\ \to\ \mathsf{bool} \quad (\text{specifies refinement of values of type } \alpha) \\ \quad \mathsf{B} : \beta\ \to\ \mathsf{v}\ \to\ \mathsf{bool} \quad (\text{specifies refinement of values of type } \beta) \end{array} \tag{1}$$

Certificate theorems in the style of the Theorem (1) are generally obtained when synthesizing *pure* CakeML programs from logical functions. The CakeML tools define two alternative refinement invariants for dealing with (potentially effectful) monadic functions: ArrowP, and ArrowM. The invariant ArrowM is used in place of $\longrightarrow$ to express refinement of monadic functions. The invariant ArrowP extends ArrowM to permit side-effects; e.g. state updates.

5.2. Certificate theorem

Here is the certificate theorem for our shallow embedding readLine:

$$\begin{array}{l} \vdash \mathsf{ArrowP\ F\ (hol\_store,}p\mathsf{)\ (Pure\ (Eq\ string\_type}\ \mathit{line\_v}\mathsf{))} \\ \quad \mathsf{(ArrowM\ F\ (hol\_store,}p\mathsf{)\ (EqSt\ (Pure\ (Eq\ reader\_state\_type}\ \mathit{state\_v}\mathsf{))}\ \mathit{state}\mathsf{)} \\ \quad\ \ \mathsf{(Monad\ reader\_state\_type\ hol\_exn\_type))\ readLine\ readline\_v} \end{array} \tag{2}$$

The specifics of the symbols involved in this theorem are outside the scope of this paper; see e.g. [4]. In short, the Theorem (2) states that readline_v is a refinement of readLine. Here, readline_v is the deep embedding that was synthesized from readLine. The invariants ArrowP and ArrowM tell us that readline_v was synthesized from a (curried) monadic function.

6. Proof checker program with I/O

Our proof-checker implementation is just about ready to be compiled; all that remains is to provide the synthesized deep embedding from §5 with input from the file system. We achieve this by wrapping the deep embedding in a ML program which takes care of I/O. The verification of the wrapper is explained in §6.2. Here is the listing for the wrapper program.

```
fun reader_main () =
  let
    val _ = init_reader ()
  in
    case CommandLine.arguments () of
```

```
        [fname] => read_file fname
      | [] => read_stdin ()
      | _ => TextIO.output TextIO.stdErr msg_usage
    end;
```

The program `reader_main` is parsed into a deeply embedded CakeML program. Here is an overview of the functionality performed by `reader_main`:

(i) The program starts by initializing the logical kernel, in particular it installs the axioms of higher-order logic (`init_reader`).
(ii) An article is read from a file (`read_file`), or standard input (`read_stdin`), and split into commands. These commands are then passed one by one to readLine (see §4) until the input is exhausted, or an exception is raised.
(iii) In case of success, the program prints out the proved theorems, together with the logical context in which they are theorems. In case of failure, the wrapper reports the line number of the failing command and exits.

We intentionally leave out listings of `read_file` and `read_stdin` for brevity. See Appendix B for the full listings.

6.1. Specification

Unlike previous stages of development (§5), the program `reader_main` must be manually verified to implement its specification. We define a logical function reader_main as the specification of `reader_main`. It is defined in terms of two functions read_file and read_stdin, corresponding to `read_file` and `read_stdin`, respectively. See Appendix C for the definitions of read_file and read_stdin.

We define reader_main as follows:

$$
\begin{array}{l}
\mathsf{reader\_main}\ \mathit{fs}\ \mathit{refs}\ \mathit{cl} = \\
\quad \mathsf{let}\ \mathit{refs}\ =\ \mathsf{snd}\ (\mathsf{init\_reader}\ ()\ \mathit{refs})\ \mathsf{in} \\
\quad\quad \mathsf{case}\ \mathit{cl}\ \mathsf{of} \\
\quad\quad\quad [\mathit{fname}]\ \Rightarrow\ \mathsf{read\_file}\ \mathit{fs}\ \mathit{refs}\ \mathit{fname} \\
\quad\quad \mid []\ \Rightarrow\ \mathsf{read\_stdin}\ \mathit{fs}\ \mathit{refs} \\
\quad\quad \mid \_\ \Rightarrow\ (\mathsf{add\_stderr}\ \mathit{fs}\ \mathsf{msg\_usage}, \mathit{refs}, \mathsf{None})
\end{array}
$$

The arguments to the function reader_main is a model of the file system, *fs*; a list of command line arguments, *cl*; and a model of the Candle kernel state (i.e. the contents of references at runtime), *refs*.

Both read_file and read_stdin are defined in terms of our shallow embedding readLine. Consequently, reader_main becomes the top-level specification for the entire proof checker program.

6.2. Verification using characteristic formulae

To show that `reader_main` adheres to its specification reader_main (see A.3 in §3) we prove a theorem using the characteristic formulæ (CF) framework for CakeML [13]. The CF framework provides a program logic for ML programs. Program specifications in CF are stated using Hoare-style triples

$$\{\!|\, P \,|\!\}\ f\ \cdot\ a\ \{\!|\, Q \,|\!\}$$

where P and Q are pre- and post-conditions on the program heap, expressed in separation logic; and $f \cdot a$ denotes the application of f to the argument list a.

Correctness of main program. This is the theorem we prove to assert that reader_main_v (the deeply-embedded syntax of `reader_main`) implements its specification reader_main:

$$
\begin{array}{l}
\vdash (\exists s.\ \mathsf{init\_reader}\ ()\ \mathit{refs} = (\mathsf{Success}\ (), s)) \wedge \mathsf{input\_exists}\ \mathit{fs}\ \mathit{cl}\ \wedge \\
\quad \mathsf{unit\_type}\ ()\ \mathit{unit\_v} \Rightarrow \\
\quad \{\!|\mathsf{commandline}\ \mathit{cl} * \mathsf{stdio}\ \mathit{fs} * \mathsf{hol\_store}\ \mathit{refs}|\!\} \\
\quad\quad \mathsf{reader\_main\_v} \cdot [\mathit{unit\_v}] \\
\quad \{\!|\mathsf{POSTv}\ \mathit{res}. \\
\quad\quad \langle \mathsf{unit\_type}\ ()\ \mathit{res} \rangle * \mathsf{stdio}\ (\mathsf{fst}\ (\mathsf{reader\_main}\ \mathit{fs}\ \mathit{refs}\ (\mathsf{tl}\ \mathit{cl})))|\!\}
\end{array}
\tag{3}
$$

Here, $*$ is the separating conjunction; commandline, stdio, and hol_store are heap assertions for the program command line, file system, and the state of the Candle logical kernel, respectively; and POSTv binds the function return value, for use in the post-condition. The exact details of the Theorem (3) are not important here; for an in-depth treatment, see [13].

Theorem (3) is the main specification of our deeply-embedded proof checker program reader_main_v. It should be read as: “if the program reader_main_v is executed from any initial state in which kernel initialization succeeds, and if any input exists on the file system, then the program terminates with a result of type unit, and produces exactly the output that reader_main does.”

The proof of Theorem (3) makes use of the certificate theorem from §5.2 which gives the semantics of the synthesized code readline_v in terms of the logical function readLine.

Summary. We conclude this section by summarizing our efforts so far.

(i) We have constructed a *shallow embedding* of the OpenTheory abstract machine, on top of the Candle theorem prover kernel (§4).
(ii) We have synthesized *deeply-embedded* CakeML from the shallow embedding, and obtained a certificate theorem (§5).
(iii) Finally, in this section, we have extended our deep embedding in code which handles I/O operations, and verified that the sum of the parts implements the semantics of the *shallow embedding*.

Below, we show how the CakeML compiler is used to compile reader_main_v to executable machine code, while at the same time producing a proof of refinement.

7. In-logic compilation

In this section we explain how the proof checker program from §6 is compiled in a way which allows us to obtain a strong correctness guarantee on the machine code produced by the compilation.

The CakeML compiler supports two modes of compilation:

(i) compilation of deep embeddings *inside* the HOL4 logic, by evaluating the shallow-embedded compiler under a call-by-value semantics;
(ii) compilation of source files (read from the file system) using a verified compiler executable.

In mode (i), the compiler produces a theorem which states that the resulting machine code is a refinement of the input program. This theorem is the CakeML compiler top-level correctness theorem specialized on the program it compiles, its specification, and the target architecture.

The CakeML compiler comes with backends for multiple architectures: x86-64, ARMv6, ARMv8, RISC-V, and MIPS [14]. The models used for reasoning about the machine code of these targets were specified using the L3 specification language [15], and were not designed specifically for use in the CakeML compiler.

We apply the in-logic compilation mode (i) to the deeply-embedded CakeML program from §6. In what follows, reader_main_v is the deep embedding of the proof checker program, and reader_main is its top-level specification (semantics).

Here is the theorem we obtain when compiling reader_main_v:

$$
\begin{array}{l}
\vdash \mathsf{input\_exists}\ fs\ cl \wedge \mathsf{wfcl}\ cl \wedge \mathsf{wfFS}\ fs \wedge \mathsf{STD\_streams}\ fs \Rightarrow \\
\quad (\mathsf{installed\_x64}\ \mathsf{reader\_code}\ (\mathsf{basis\_ffi}\ cl\ fs)\ mc\ ms \Rightarrow \\
\quad\ \mathsf{machine\_sem}\ mc\ (\mathsf{basis\_ffi}\ cl\ fs)\ ms \subseteq \\
\quad\ \mathsf{extend\_with\_resource\_limit}\ \{\ \mathsf{Terminate\ Success}\ (\mathsf{reader\_io\_events}\ cl\ fs)\ \}\ ) \wedge \\
\quad \mathsf{let}\ (fs\_out, hol\_refs, final\_state) = \mathsf{reader\_main}\ fs\ \mathsf{init\_refs}\ (\mathsf{tl}\ cl) \\
\quad \mathsf{in} \\
\quad\ \ \mathsf{extract\_fs}\ fs\ (\mathsf{reader\_io\_events}\ cl\ fs) = \mathsf{Some}\ fs\_out
\end{array} \tag{4}
$$

In brief, this theorem states that the semantics of the machine code of the compiled program reader_code only includes behaviors allowed by the shallow embedding reader_main. We will explain Theorem (4) in the following paragraphs.

Assumptions on the environment. Theorem (4) contains the following assertion, which ensures that reader_code is executed in a machine state *ms* where the necessary code and data are correctly installed in memory:

$$\mathsf{installed\_x64}\ \mathsf{reader\_code}\ (\mathsf{basis\_ffi}\ cl\ fs)\ mc\ ms$$

The arguments to installed_x64 are the concrete machine code reader_code, a machine state *ms*, and an architecture-specific configuration, *mc*. In addition, it takes an oracle basis_ffi *cl fs*, which represents our assumptions about the file system and command line.

Out-of-memory errors. The top-level correctness result of the CakeML compiler guarantees that any machine code obtained from compilation is semantically *compatible* with the observable semantics of the source program that was compiled. Concretely, compatible means “equivalent, up to failure from running out of memory.” This is expressed in Theorem (4) by the following lines:

$$\begin{array}{l} \mathsf{machine\_sem}\ mc\ (\mathsf{basis\_ffi}\ cl\ fs)\ ms \subseteq \\ \quad \mathsf{extend\_with\_resource\_limit}\ \{\ \mathsf{Terminate\ Success}\ (\mathsf{reader\_io\_events}\ cl\ fs)\ \} \end{array}$$

Here, machine_sem denotes the semantics of the machine code produced during compilation, and extend_with_resource_limit $\{\cdots\}$ is the set of all prefixes of the observable semantics of the source program, as well as all those prefixes concatenated with a final event that denotes failure.

Observable semantics. The CakeML compiler's correctness is stated in terms of *observable* events. This semantics consists of a (possibly infinite) sequence of I/O events that modify our model of the world in some way. The following line states that the result of running these computations amounts to the same modifications of the file system model *fs*, as the program specification reader_main does:

$$\mathsf{extract\_fs}\ fs\ (\mathsf{reader\_io\_events}\ cl\ fs) = \mathsf{Some}\ fs\_out$$

With the help of Theorem (5) we have established a convincing connection between the logical specification of our proof checker (§4), and the machine code which executes it. Consequently, any claims made about the shallow-embedded proof checker can be transported to the level of machine code. In the next section, we bring all of these results together to form a single top-level correctness theorem.

8. End-to-end correctness

In this section we present the main correctness theorem for the OpenTheory proof checker. This theorem is a soundness result which ensures that the executable machine code that is the compiled proof checker (§7) only accepts valid proofs of theorems. In particular, we show that any theorem constructed from a successful run of the OpenTheory proof checker is in fact true by the semantics of HOL. This result is made possible by the soundness theorem of the Candle theorem prover kernel [2].

Here is the soundness result for the OpenTheory proof checker.

$$\begin{array}{l}
\vdash \mathsf{input\_exists}\ fs\ cl \wedge \mathsf{wfcl}\ cl \wedge \mathsf{wfFS}\ fs \wedge \mathsf{STD\_streams}\ fs \Rightarrow \\
\quad (\mathsf{installed\_x64}\ \mathsf{reader\_code}\ (\mathsf{basis\_ffi}\ cl\ fs)\ mc\ ms \Rightarrow \\
\quad\ \mathsf{machine\_sem}\ mc\ (\mathsf{basis\_ffi}\ cl\ fs)\ ms \subseteq \\
\quad\ \mathsf{extend\_with\_resource\_limit} \\
\qquad\quad \{\ \mathsf{Terminate\ Success}\ (\mathsf{reader\_io\_events}\ cl\ fs)\ \}\ ) \wedge \\
\quad \exists fs\_out\ hol\_refs\ s. \\
\qquad \mathsf{extract\_fs}\ fs\ (\mathsf{reader\_io\_events}\ cl\ fs) = \mathsf{Some}\ fs\_out \wedge \\
\qquad (\mathsf{no\_errors}\ fs\ fs\_out \Rightarrow \\
\qquad\ \mathsf{reader\_main}\ fs\ \mathsf{init\_refs}\ (\mathsf{tl}\ cl) = (fs\_out, hol\_refs, \mathsf{Some}\ s) \wedge \\
\qquad\ hol\_refs.\mathsf{the\_context}\ \mathsf{extends}\ \mathsf{init\_ctxt} \wedge \\
\qquad\ fs\_out = \mathsf{add\_stdout}\ (\mathsf{flush\_stdin}\ (\mathsf{tl}\ cl)\ fs) \\
\qquad\qquad (\mathsf{print\_theorems}\ s\ hol\_refs.\mathsf{the\_context}) \wedge \\
\qquad\ \forall asl\ c. \\
\qquad\qquad \mathsf{mem}\ (\mathsf{Sequent}\ asl\ c)\ s.\mathsf{thms} \wedge \\
\qquad\qquad \mathsf{is\_set\_theory}\ \mu \Rightarrow \\
\qquad\qquad (\mathsf{thyof}\ hol\_refs.\mathsf{the\_context}, asl) \models c)
\end{array} \tag{5}$$

where $\mathsf{no\_errors}\ fs\ fs\_out = (fs.\mathsf{stderr} = fs\_out.\mathsf{stderr})$

The first part of Theorem (5) is identical to the machine code correctness theorem (4) in §7. In short, it states that the machine code reader_code faithfully implements the shallow embedding reader_main; see §7 for details.

The interesting parts of Theorem (5) are the last few lines, starting at the existential quantification $\exists fs\_out$. The lines

$$\begin{array}{l} \mathsf{no\_errors}\ fs\ fs\_out\ \Rightarrow \\ \mathsf{reader\_main}\ fs\ \mathsf{init\_refs}\ (\mathsf{tl}\ cl) = (fs\_out, hol\_refs, \mathsf{Some}\ s) \wedge \ldots \end{array}$$

state: if no errors were displayed on screen, then the OpenTheory proof checker successfully processed all commands in the article, and returned a final state s of type state.

The next few lines contain information about this final state; in particular, that:

- all constructed theorems (those in s.thms; see §4) are true under the semantics of HOL;
- the logical context (*hol_refs*.the_context) in which these theorems are true is the result of a sequence of valid updates to the initial context of the Candle kernel; and

- the result displayed on screen (add_stdout $\cdots$) by the program is a textual representation of the logical context and the constructed theorems.

Before moving on, we note a somewhat particular feature of Theorem (5); namely the requirement is_set_theory μ. In brief, is_set_theory assumes the existence of a set theory expressive enough to contain the semantics of HOL; it is used in the Candle soundness result to lift syntactic entailment to semantic entailment. We will touch on the subject briefly in §8.1, but refer readers to Kumar, et al. [2] for an in-depth discussion.

We will use the remainder of this section to explain how we obtain a soundness result for the shallow embedding from §4. We then compose this result with the machine code theorem from §7 in order to obtain the Theorem (5).

8.1. The Candle soundness result

In this section we explain what is required to make use of the Candle soundness result when proving our top-level correctness theorem (5). The formalization of the Candle logical kernel is divided in two parts: a calculus of proof rules for constructing sequents, and a formal semantics. Both systems are defined in the logic of HOL4.

We will not attempt to explain the formalization at any greater depth as this is well outside the scope of this work. However, a basic understanding of some of the techniques used to obtain the Candle soundness result will be necessary to arrive at Theorem (5) in §8.

Syntactic predicates. The Candle proof development defines a number of predicates on syntactic elements of HOL. The most important of these is the relation THM, which states that a sequent is the result of a valid inference in HOL, in a specific context. It is defined in terms of a proof rule for HOL, $\vdash$:

$$\text{THM}\ \mathit{ctxt}\ (\text{Sequent}\ \mathit{asl}\ c) = (\text{thyof}\ \mathit{ctxt}, \mathit{asl}) \vdash c$$

Here, $\vdash$ is an inductively defined relation that makes up the proof calculus (i.e. syntactic inference rules) of the higher-order logic implemented by the Candle logical kernel. We leave out the definition of $\vdash$ here; see e.g. [2,9] for a description of the calculus.

For the proof rule $\vdash$ to establish validity of inferences, it imposes some restrictions on terms and types used in inferences; e.g. terms must be well-typed, constants and types must be defined prior to use, and type operators must be used with their correct arity. These restrictions are established by the relations TYPE and TERM.

Soundness. Finally, any statement about $\vdash$ (and consequently, THM) can be turned into a statement about semantic entailment, thanks to the main soundness result of the Candle kernel [2]:

$$\text{is\_set\_theory}\ \mu \Rightarrow \forall \mathit{hyps}\ c.\ \mathit{hyps} \vdash c \Rightarrow \mathit{hyps} \models c$$

We make use of this in §8.3 to lift a syntactic result about our proof checker into the semantic domain.

8.2. Preserving invariants

In order to establish soundness for our proof checker, we need to show a result which states that all theorems constructed by the proof-checker are in fact true theorems of HOL. In this section we explain how this is achieved by proving a preservation result for the shallow embedding from §4.

We will obtain this result in three steps, by:

(i) defining a property for the type object, which will establish the relevant invariants (THM, etc.) on the HOL syntax carried by object (§4.2);
(ii) defining a property for the OpenTheory machine state type state (§4.1), imposing the object property from (i) on all its objects; and
(iii) proving that the property from (ii) is preserved under the shallow embedding readLine (§4.4).

Object predicate. We start by addressing Step (i), and define a property on objects:

```
OBJ ctxt obj =
 case obj of
  List xs ⇒ every (OBJ ctxt) xs
 | Type ty ⇒ TYPE ctxt ty
 | Term tm ⇒ TERM ctxt tm
 | Thm thm ⇒ THM ctxt thm
 | Var (n,ty) ⇒ TERM ctxt (Var n ty) ∧ TYPE ctxt ty
 | _ ⇒ T
```

The function OBJ asserts that all types are valid, e.g. type operators exist in the context *ctxt*, and have the correct arity (TYPE); and that all terms are well-typed in *ctxt*, and contain only defined constants (TERM).

State predicate. Next, we carry out Step (ii) by lifting the properties OBJ and THM to the state type. We do this with a function called READER_STATE:

$$\begin{array}{l}
\mathsf{READER\_STATE}\ ctxt\ state = \\
\quad \mathsf{every}\ (\mathsf{THM}\ ctxt)\ state.\mathsf{thms} \wedge \\
\quad \mathsf{every}\ (\mathsf{OBJ}\ ctxt)\ state.\mathsf{stack} \wedge \\
\quad \forall n\ obj. \\
\qquad \mathsf{lookup}\ (\mathsf{Num}\ n)\ state.\mathsf{dict} = \mathsf{Some}\ obj \Rightarrow \\
\qquad \mathsf{OBJ}\ ctxt\ obj
\end{array}$$

The important part about READER_STATE is that THM holds for all HOL sequents in the theorem stack *state*.thms; enforcing OBJ on the stack and dictionary is simply a means to achieving this.

Preservation theorem. Finally, we take care of Step (iii). We prove the following preservation theorem, which guarantees that THM holds for all sequents in the program state, at all times during execution:

$$\begin{array}{l}
\vdash \mathsf{STATE}\ ctxt\ refs \wedge \mathsf{READER\_STATE}\ ctxt\ st \wedge \\
\quad \mathsf{readLine}\ line\ st\ refs = (res, refs') \Rightarrow \\
\quad \exists upd. \\
\qquad \mathsf{STATE}\ (upd +\!\!+\ ctxt)\ refs' \wedge \\
\qquad \forall st'.\ res = \mathsf{Success}\ st' \Rightarrow \mathsf{READER\_STATE}\ (upd +\!\!+\ ctxt)\ st'
\end{array} \tag{6}$$

The relation STATE connects the logical context *ctxt* with the concrete state of the Candle kernel at runtime. The context *ctxt* is modeled as a sequence of *updates* (e.g. constant- and type definitions, new axioms, etc.). With this in mind, Theorem (6) can be read as: “the relations STATE and READER_STATE are preserved under readLine, up to a finite sequence of valid context updates to the initial context *ctxt*.”

Using Theorem (6), we are able to prove that THM holds for all theorems kept in the state at all times, as long as the function readLine starts from an initial state where this is true (e.g. the empty state). In §8.3 we compose this result with the Candle soundness result (§8.1), and show that soundness holds for our shallow embedded proof checker.

8.3. Soundness of the shallow embedding

With Theorem (6) in §8.2, we showed that any sequent constructed by the proof checker at runtime is the result of a valid inference in HOL. In this section we lift this result into a theorem about soundness, by using the Candle soundness result shown in §8.1.

Our soundness theorem is stated in terms of the proof checker specification reader_main from §6.1:

$$\begin{array}{l}
\vdash \mathsf{is\_set\_theory}\ \mu \wedge \\
\quad \mathsf{reader\_main}\ fs\ \mathsf{init\_refs}\ cl = (fs\_out, hol\_refs, \mathsf{Some}\ s) \Rightarrow \\
\quad (\forall asl\ c. \\
\qquad \mathsf{mem}\ (\mathsf{Sequent}\ asl\ c)\ s.\mathsf{thms} \Rightarrow \\
\qquad (\mathsf{thyof}\ hol\_refs.\mathsf{the\_context}, asl) \models c) \wedge \\
\quad hol\_refs.\mathsf{the\_context}\ \mathsf{extends}\ \mathsf{init\_ctxt} \wedge \\
\quad fs\_out = \mathsf{add\_stdout}\ (\mathsf{flush\_stdin}\ cl\ fs)\ (\mathsf{print\_theorems}\ s\ hol\_refs.\mathsf{the\_context})
\end{array} \tag{7}$$

With this theorem, we have all ingredients required to obtain the main correctness Theorem (5) from §8:

- Theorem (7) is stated in terms reader_main, and guarantees that the main proof checker program from §6 is sound.
- Theorem (4) shows that the machine code reader_code is a refinement of the program in §6.

Because both these theorems are stated in terms of reader_main, the results can be trivially composed in the HOL4 system to produce the desired theorem (5).

9. Results

Our proof checker was used to check a few articles from the OpenTheory standard library. These articles were selected based on the number of proof commands contained in the article (i.e. their size); larger article files exist in the standard library, but require significantly more time to process. All articles were successfully processed without errors.

Table 1
Comparison of average running times when running each tool 10 times on each input. Times are formatted as (mean ± σ).

	`bool.art`	`base.art`	`real.art`	`word.art`
# commands	62k	1718k	1285k	2121k
OPC	0.353 ± 0.002 s	9.730 ± 0.156 s	7.260 ± 0.018 s	12.05 ± 0.133 s
MLT	0.076 ± 0.002 s	1.967 ± 0.016 s	1.526 ± 0.008 s	2.629 ± 0.015 s
PML	0.160 ± 0.002 s	6.597 ± 0.192 s	4.410 ± 0.060 s	7.623 ± 0.165 s
MML	0.934 ± 0.008 s	85.01 ± 0.655 s	46.45 ± 0.137 s	121.9 ± 0.395 s
OPC/MLT	4.63	4.95	4.76	4.58
OPC/PML	2.21	1.48	1.65	1.58
OPC/MML	0.38	0.11	0.16	0.10

Here OPC is our verified proof-checker binary; MLT is the OpenTheory tool compiled with MLton; PML is the OpenTheory tool compiled with Poly/ML; MML is the OpenTheory tool compiled with Moscow ML.

We have evaluated the performance of our proof checker program, and compared it to an existing (unverified) tool [16], built using three Standard ML compilers: MLton [17], Poly/ML [18] and Moscow ML [19]. Tests were carried out on a Intel i7-7820HQ running at 2.90 GHz with 16 GB RAM, by recording time elapsed when running each tool 10 times on the same input. The results of the performance measurements are shown in Table 1.

When compared against the OpenTheory tool [16], our proof checker runs a factor of 4.7 times slower than the MLton compiled binary on average, and 1.7 times slower than the Poly/ML binary on average. A significant portion of this slowdown is caused by poor I/O performance, as our proof checker spends about half of its time performing system calls for I/O. It is difficult to determine the exact cause of the remainder of the slowdown; our HOL implementation is different from that of the OpenTheory tool, and the performance of the executable code generated by the compilers used in this test varies greatly (cf. Table 1). We expect that improvements to CakeML I/O facilities will improve the performance of our proof checker.

10. Discussion and related work

In this work we have implemented and verified a proof checker for HOL that checks proofs in the OpenTheory article format. The proof checker builds on the verified Candle theorem prover kernel by Kumar, et al. [2], and uses the CakeML toolchain [4,5,20] to produce a verified executable binary. To the best of our knowledge, this is the first fully verified proof checker for HOL.

We have left out some features present in the OpenTheory article format when implementing our checker. In particular, theories in the OpenTheory framework support external assumptions, such as constant definitions, type operators, and axioms. Our proof checker implementation (§4) does not currently support external assumptions, because of the way in which constants and type operators are treated in the readLine function. However, we believe it could be extended to do so without compromising soundness.

The main motivation behind the OpenTheory article format is mainly *theorem export*. Our tool checks the validity of proofs by carrying out all inferences required to reconstruct theorems, and if the reconstruction succeeds, we know by the correctness result in §8 that the theorem must be valid. However, this approach is not without its drawbacks, as there is no way to tell the checker what theorem we *expect* it to prove. Hence, if proof recording has gone awry (for whatever reason), it is possible that we prove a different (albeit still true) theorem.

HOL proof checkers. It appears that proof checkers for higher-order logic are few and far between.

The OpenTheory framework [1] includes a tool called the OpenTheory tool [16], written in Standard ML. Among other things, the tool is capable of checking OpenTheory articles in the same way our verified proof checker is. When compared to the OpenTheory tool (§9), our tool runs slower, and supports fewer of the features available in the OpenTheory framework. However, the correctness of the OpenTheory tool has not been verified in any way.

The HOL Zero system by Adams [21] is a theorem prover for higher-order logic with a particular focus on trustworthiness. Unlike ours, the system is not formally verified; instead, its claims of high reliability are grounded in a simple and understandable design of the logical kernel on which the tool builds. Unlike other HOL provers, the tool is not interactive, but rather, it acts as a proof-checker of sorts.

Verified proof checkers. The IVY system (McCune and Shumsky [22]) is a verified prover for first-order logic with equality. IVY relies on fast, trusted C code for finding proofs, and verifies the resulting proofs using a checker algorithm which has been verified sound using the ACL2 system [23].

Ridge and Margetson [22] implements a theorem prover for first-order logic, and verifies it complete and sound with respect to a standard semantics. The development and verification is carried out in Isabelle/HOL [24], and includes an "algorithm which tests a sequent *s* for first-order validity." The algorithm can be executed within the Isabelle/HOL logic, by using the rewrite engine.

The Milawa theorem prover (Davis and Myreen [25]) is perhaps the most impressive work to date in the space of verified theorem provers. Milawa is an extensible theorem prover for a first-order logic, in the style of ACL2 [23]. The system starts out as a simple proof checker, and is able to bootstrap itself into a fully-fledged theorem prover by replacing parts of its logical kernel at runtime. In [25], the authors verify that Milawa is sound down to the machine code which executes it, when run on top of their verified LISP implementation Jitawa.

11. Summary

We have presented a verified computer program for checking proofs of theorems in higher-order logic. The proof checker program is implemented in CakeML, and is compiled to machine code using the CakeML compiler. The program reads proof articles in the OpenTheory article format, and has been formally verified to only accept valid proofs. To the best of our knowledge, this is the first formally verified proof checker for HOL.

The proof checker implementation and its proof is available at GitHub: github.com/CakeML/cakeml/tree/master/candle/standard/opentheory

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The original implementation of the OpenTheory stack machine in monadic HOL was done by Ramana Kumar, who also provided helpful support during the course of this work. The author would also like to thank Magnus Myreen for feedback on this text. Finally, the author thanks the anonymous reviewers for their helpful comments.

Appendix A. OpenTheory abstract machine

The definition of the shallow-embedded OpenTheory machine (§4.4).

```
readLine line s =
 if line = "version" then
  do
   (obj,s) ← pop s; getNum obj;
   return s
  od
  else if line = "absTerm" then
  do
   (obj,s) ← pop s; b ← getTerm obj;
   (obj,s) ← pop s; v ← getVar obj;
   tm ← mk_abs (mk_var v,b);
   return (push (Term tm) s)
  od
  else if line = "absThm" then
  do
   (obj,s) ← pop s; th ← getThm obj;
   (obj,s) ← pop s; v ← getVar obj;
   th ← ABS (mk_var v) th;
   return (push (Thm th) s)
  od
  else if line = "appTerm" then
  do
   (obj,s) ← pop s; x ← getTerm obj;
   (obj,s) ← pop s; f ← getTerm obj;
   fx ← mk_comb (f,x);
   return (push (Term fx) s)
  od
  else if line = "appThm" then
  do
   (obj,s) ← pop s; xy ← getThm obj;
   (obj,s) ← pop s; fg ← getThm obj;
   th ← MK_COMB (fg,xy);
   return (push (Thm th) s)
  od
...
```

```
. . .
    else if $line$ = "assume" then
    do
      $(obj,s)$ $\leftarrow$ pop $s$; $tm$ $\leftarrow$ getTerm $obj$;
      $th$ $\leftarrow$ ASSUME $tm$;
      return (push (Thm $th$) $s$)
    od
  else if $line$ = "axiom" then
   do
    $(obj,s)$ $\leftarrow$ pop $s$; $tm$ $\leftarrow$ getTerm $obj$;
    $(obj,s)$ $\leftarrow$ pop $s$; $ls$ $\leftarrow$ getList $obj$;
    $ls$ $\leftarrow$ map getTerm $ls$;
    $th$ $\leftarrow$ find_axiom $(ls,tm)$;
    return (push (Thm $th$) $s$)
   od
  else if $line$ = "betaConv" then
   do
    $(obj,s)$ $\leftarrow$ pop $s$; $tm$ $\leftarrow$ getTerm $obj$;
    $th$ $\leftarrow$ BETA_CONV $tm$;
    return (push (Thm $th$) $s$)
   od
  else if $line$ = "cons" then
   do
    $(obj,s)$ $\leftarrow$ pop $s$; $ls$ $\leftarrow$ getList $obj$;
    $(obj,s)$ $\leftarrow$ pop $s$;
    return (push (List $(obj::ls)$) $s$)
   od
  else if $line$ = "const" then
   do
    $(obj,s)$ $\leftarrow$ pop $s$; $n$ $\leftarrow$ getName $obj$;
    return (push (Const $n$) $s$)
   od
  else if $line$ = "constTerm" then
   do
    $(obj,s)$ $\leftarrow$ pop $s$; $ty$ $\leftarrow$ getType $obj$;
    $(obj,s)$ $\leftarrow$ pop $s$; $nm$ $\leftarrow$ getConst $obj$;
    $ty_0$ $\leftarrow$ get_const_type $nm$;
    $tm$ $\leftarrow$
      case match_type $ty_0$ $ty$ of
        None $\Rightarrow$ failwith "constTerm"
      | Some $theta$ $\Rightarrow$ mk_const $(nm,theta)$;
    return (push (Term $tm$) $s$)
   od
  else if $line$ = "deductAntisym" then
   do
    $(obj,s)$ $\leftarrow$ pop $s$; $th_2$ $\leftarrow$ getThm $obj$;
    $(obj,s)$ $\leftarrow$ pop $s$; $th_1$ $\leftarrow$ getThm $obj$;
    $th$ $\leftarrow$ DEDUCT_ANTISYM_RULE $th_1$ $th_2$;
    return (push (Thm $th$) $s$)
   od
  else if $line$ = "def" then
   do
    $(obj,s)$ $\leftarrow$ pop $s$; $n$ $\leftarrow$ getNum $obj$;
    $obj$ $\leftarrow$ peek $s$;
    if $n < 0$ then failwith "def" else
      return (insert_dict (Num $n$) $obj$ $s$)
   od
. . .
```

```
...
else if line = "defineConst" then
 do
  (obj,s) ← pop s;  tm ← getTerm obj;
  (obj,s) ← pop s;  n ← getName obj;
  ty ← type_of tm;
  eq ← mk_eq (mk_var (n,ty),tm);
  th ← new_basic_definition eq;
  return (push (Thm th) (push (Const n) s))
 od
else if line = "defineConstList" then
 do
  (obj,s) ← pop s;  th ← getThm obj;
  (obj,s) ← pop s;  ls ← getList obj;
  ls ← map getNvs ls;
  th ← INST ls th;
  th ← new_specification th;
  ls ← map getCns ls;
  return (push (Thm th) (push (List ls) s))
 od
else if line = "defineTypeOp" then
 do
  (obj,s) ← pop s;  th ← getThm obj;
  (obj,s) ← pop s;  getList obj;
  (obj,s) ← pop s;  rep ← getName obj;
  (obj,s) ← pop s;  abs ← getName obj;
  (obj,s) ← pop s;  nm ← getName obj;
  ($th_1$,$th_2$) ← new_basic_type_definition nm abs rep th;
  (_,a) ← dest_eq (concl $th_1$);
  $th_1$ ← ABS a $th_1$;
  $th_2$ ← SYM $th_2$;
  (_,Pr) ← dest_eq (concl $th_2$);
  (_,r) ← dest_comb Pr;
  $th_2$ ← ABS r $th_2$;
  return (push (Thm $th_2$) (push (Thm $th_1$) (push (Const rep)
      (push (Const abs) (push (TypeOp nm) s)))))
 od
else if line = "eqMp" then
 do
  (obj,s) ← pop s;  $th_2$ ← getThm obj;
  (obj,s) ← pop s;  $th_1$ ← getThm obj;
  th ← EQ_MP $th_1$ $th_2$;
  return (push (Thm th) s)
 od
else if line = "hdTl" then
 do
  (obj,s) ← pop s;  ls ← getList obj;
  case ls of
   [] ⇒ failwith "hdTl"
  | h::t ⇒ return (push (List t) (push h s))
 od
else if line = "nil" then return (push (List []) s)
else if line = "opType" then
 do
  (obj,s) ← pop s;  ls ← getList obj;
  args ← map getType ls;
  (obj,s) ← pop s;  tyop ← getTypeOp obj;
  t ← mk_type (tyop,args);
  return (push (Type t) s)
 od
else if line = "pop" then do (_,s) ← pop s;  return s od
...
```

```
...
else if line = "pragma" then
 do
  (obj,s) ← pop s;
  nm ← handle (getName obj) (λ e. return "bogus");
  if nm = "debug" then failwith (state_to_string s) else return s
 od
else if line = "proveHyp" then
 do
  (obj,s) ← pop s; $th_2$ ← getThm obj;
  (obj,s) ← pop s; $th_1$ ← getThm obj;
  th ← PROVE_HYP $th_2$ $th_1$;
  return (push (Thm th) s)
 od
else if line = "ref" then
 do
  (obj,s) ← pop s; n ← getNum obj;
  if $n < 0$ then failwith "ref" else
   case lookup (Num n) s.dict of
    None ⇒ failwith "ref"
   | Some obj ⇒ return (push obj s)
 od
else if line = "refl" then
 do
  (obj,s) ← pop s; tm ← getTerm obj;
  th ← REFL tm;
  return (push (Thm th) s)
 od
else if line = "remove" then
 do
  (obj,s) ← pop s; n ← getNum obj;
  if $n < 0$ then failwith "ref" else
   case lookup (Num n) s.dict of
    None ⇒ failwith "remove"
   | Some obj ⇒ return (push obj (delete_dict (Num n) s))
 od
else if line = "subst" then
 do
  (obj,s) ← pop s; th ← getThm obj;
  (obj,s) ← pop s; (tys,tms) ← getPair obj;
  tys ← getList tys;
  tys ← map getTys tys;
  th ← handle_clash (INST_TYPE tys th) (λ e. failwith "the impossible");
  tms ← getList tms;
  tms ← map getTms tms;
  th ← INST tms th;
  return (push (Thm th) s)
 od
else if line = "sym" then
 do
  (obj,s) ← pop s; th ← getThm obj;
  th ← SYM th;
  return (push (Thm th) s)
 od
else if line = "thm" then
 do
  (obj,s) ← pop s; c ← getTerm obj;
  (obj,s) ← pop s; h ← getList obj;
  h ← map getTerm h;
  (obj,s) ← pop s; th ← getThm obj;
  th ← ALPHA_THM th (h,c);
  return (s with thms := th::s.thms)
 od
...
```

```
...
 else if line = "trans" then
  do
   (obj,s) ← pop s; $th_2$ ← getThm obj;
   (obj,s) ← pop s; $th_1$ ← getThm obj;
   th ← TRANS $th_1$ $th_2$;
   return (push (Thm th) s)
  od
 else if line = "typeOp" then
  do
   (obj,s) ← pop s; n ← getName obj;
   return (push (TypeOp n) s)
  od
 else if line = "var" then
  do
   (obj,s) ← pop s; ty ← getType obj;
   (obj,s) ← pop s; n ← getName obj;
   return (push (Var (n,ty)) s)
  od
 else if line = "varTerm" then
  do
   (obj,s) ← pop s; v ← getVar obj;
   return (push (Term (mk_var v)) s)
  od
 else if line = "varType" then
  do
   (obj,s) ← pop s; n ← getName obj;
   return (push (Type (mk_vartype n)) s)
  od
 else
  case s2i line of
   Some n ⇒ return (push (Num n) s)
  | None ⇒
     case explode line of
       "" ⇒ failwith ("unrecognised input: " ˆ line)
     | "\"" ⇒ failwith ("unrecognised input: " ˆ line)
     | #"""::c::cs ⇒
       return
        (push (Name (implode (front (c::cs)))) s)
     | _ ⇒ failwith ("unrecognised input: " ˆ line)
```

Appendix B. Listings of CakeML code

The listing for `read_stdin` (§6).

```
fun read_stdin () =
  let
    val ls = TextIO.inputLines TextIO.stdin
  in
    process_list ls init_state
  end;
```

The listing for `read_file` (§6).

```
fun read_file file =
  let
    val ins = TextIO.openIn file
  in
    process_lines ins init_state;
    TextIO.closeIn ins
  end
  handle TextIO.BadFileName =>
    TextIO.output TextIO.stdErr
      (msg_filename_err file);
```

The listing for `process_list`, which calls `process_line` on a list of commands.

```
fun process_list ls s =
  case ls of
    [] => TextIO.print
      (print_theorems s (Kernel.context ()))
  | l::ls =>
    case process_line s l of
      Inl s =>
        process_list ls (next_line s)
    | Inr e =>
        TextIO.output TextIO.stdErr (line_fail s e);
```

The listing for `process_lines`, which reads a proof command (string) from an input stream, and calls `process_line` on the result, until the input is exhausted.

```
fun process_lines ins st0 =
  case TextIO.inputLine ins of
    None =>
      TextIO.print (print_theorems st0 (Kernel.context ()))
  | Some ln =>
    case process_line st0 ln of
      Inl st1 =>
        process_lines ins (next_line st1)
    | Inr e =>
        TextIO.output TextIO.stdErr (line_fail st0 e))`;
```

The listing for `process_line`, which calls a synthesized version of readLine (§5) on a proof command (§4.3).

```
fun process_line st ln =
  if invalid_line ln then
    Inl st
  else
    Inl (readline (preprocess ln) st)
    handle Fail e => Inr e;
```

Appendix C. Specifications for CakeML code

The definition of readLines, which calls on readLine (§4.4, and Appendix A) to process a list of proof commands (§4.3).

```
readLines lines st =
 case lines of
  [] ⇒ return (st,lines_read st)
 | l::ls ⇒
    if invalid_line l then readLines ls (next_line st) else
    do
     st′ ← handle (readLine (preprocess l) st)
                  (λ e. failwith (line_num_err st e));
     readLines ls (next_line st′)
    od
```

The definition of read_file (§6.1).

```
read_file fs refs fname =
 if inFS_fname fs (File fname) then
  case readLines (all_lines fs (File fname)) init_state refs of
   (Success (s,_),refs) ⇒
     (add_stdout fs (print_theorems s refs.the_context),refs,Some s)
  | (Failure (Fail e),refs) ⇒ (add_stderr fs e,refs,None)
 else (add_stderr fs (msg_filename_err fname),refs,None)
```

The definition of read_stdin (§6.1).

```
read_stdin fs refs =
 let fs′ = fastForwardFD fs 0 in
  case readLines (all_lines fs (IOStream "stdin")) init_state refs of
   (Success (s,_),refs) ⇒
      (add_stdout fs′ (print_theorems s refs.the_context),refs,Some s)
  | (Failure (Fail e),refs) ⇒
      (add_stderr fs′ e,refs,None)
```

References

[1] J. Hurd, The OpenTheory standard theory library, in: NFM, 2011, pp. 177–191.
[2] R. Kumar, R. Arthan, M.O. Myreen, S. Owens, Self-formalisation of higher-order logic - semantics, soundness, and a verified implementation, J. Autom. Reason. 56 (3) (2016) 221–259.
[3] K. Slind, M. Norrish, A brief overview of HOL4, in: TPHOLs, 2008, pp. 28–32.
[4] S. Ho, O. Abrahamsson, R. Kumar, M.O. Myreen, Y.K. Tan, M. Norrish, Proof-producing synthesis of CakeML with I/O and local state from monadic HOL functions, in: IJCAR, 2018, pp. 646–662.
[5] Y.K. Tan, M.O. Myreen, R. Kumar, A. Fox, S. Owens, M. Norrish, The verified CakeML compiler backend, J. Funct. Program. 29 (2019) e2.
[6] J. Harrison, HOL Light: an overview, in: TPHOLs, 2009, pp. 60–66.
[7] R. Arthan, The ProofPower web pages [online], 2017. (Accessed 22 February 2019).
[8] P. Wadler, Monads for functional programming, in: Advanced Functional Programming, Tutorial Text, in: Lecture Notes in Computer Science, Springer, 1995.
[9] J. Harrison, Towards self-verification of HOL Light, in: IJCAR, 2006, pp. 177–191.
[10] R. Milner, M. Tofte, R. Harper, Definition of Standard ML, MIT Press, 1997.
[11] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, J. Vouillon, The OCaml system documentation and user's manual [online], 2018. (Accessed 25 February 2019).
[12] J. Hurd, The OpenTheory article file format [online], 2014. (Accessed 22 February 2019).
[13] A. Guéneau, M.O. Myreen, R. Kumar, M. Norrish, Verified characteristic formulae for CakeML, in: ESOP, 2017, pp. 584–610.
[14] A.C.J. Fox, M.O. Myreen, Y.K. Tan, R. Kumar, Verified compilation of CakeML to multiple machine-code targets, in: CPP, ACM, 2017, pp. 125–137.
[15] A.C.J. Fox, Directions in ISA specification, in: ITP, Springer, 2012, pp. 338–344.
[16] J. Hurd, The OpenTheory tool [online], 2018. (Accessed 26 February 2019).
[17] The MLton compiler [online]. (Accessed 26 October 2019).
[18] The Poly/ML compiler [online]. (Accessed 26 October 2019).
[19] The Moscow ML compiler [online]. (Accessed 26 October 2019).
[20] R. Kumar, M.O. Myreen, M. Norrish, S. Owens, CakeML: a verified implementation of ML, in: POPL, 2014, pp. 179–192.
[21] M. Adams, Introducing HOL Zero - (extended abstract), in: ICMS, 2010, pp. 142–143.
[22] T. Ridge, J. Margetson, A mechanically verified, sound and complete theorem prover for first order logic, in: TPHOLs, 2005, pp. 294–309.
[23] M. Kaufmann, J.S. Moore, An industrial strength theorem prover for a logic based on common lisp, IEEE Trans. Softw. Eng. 23 (4) (1997) 203–213.
[24] T. Nipkow, L.C. Paulson, M. Wenzel, Isabelle/HOL - a Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283, Springer, 2002.
[25] J. Davis, M.O. Myreen, The reflective Milawa theorem prover is sound (down to the machine code that runs it), J. Autom. Reason. 55 (2) (2015) 117–183.