CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Thesis for the Degree of Licentiate of Engineering

Verified proof checking for higher-order logic

Oskar Abrahamsson

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Gothenburg, Sweden
2020

Verified proof checking for higher-order logic

Department of Computer Science and Engineering

Chalmers University of Technology and
University of Gothenburg

SE-412 96 Gothenburg, Sweden
Telephone +46 (0)31-772 1000

Printed at Reproservice, Chalmers University of Technology
Gothenburg, Sweden, 2020

Abstract

This thesis is about verified computer-aided checking of mathematical proofs. We build on tools for proof-producing program synthesis, and verified compilation, and a verified theorem proving kernel. Using these tools, we have produced a mechanized proof checker for higher-order logic that is verified to only accept valid proofs. To the best of our knowledge, this is the only proof checker for HOL that has been verified to this degree of rigor.

Mathematical proofs exist to provide a high degree of confidence in the truth of statements. The level of confidence we place in a proof depends on its correctness. This correctness is usually established through proof checking, performed either by human or machine. One benefit of using a machine for this task is that the correctness of the machine itself can be proven.

The main contribution of this work is a verified mechanized proof checker for theorems in higher-order logic (HOL). The checker is implemented as functions in the logic of the HOL4 theorem prover, and it comes with a soundness result, which states that it will only accept proofs of true theorems of HOL. Using a technique for proof-producing code generation (which is extended as part of this thesis), we synthesize a CakeML program that is compiled using the CakeML compiler. The CakeML compiler is verified to preserve program semantics. As a consequence, we are able to obtain a soundness result about the machine code which implements the proof checker.

Contents

Chapter 1

Introduction

This Licentiate thesis is about rigorous mechanized checking of mathematical proofs. Its main contribution is a mechanized proof checker which is verified to be correct using state-of-the-art tools and techniques.

1.1 Motivation

Mathematical proof is used to establish strong guarantees about the truth of statements in a general way. Empirical methods (e.g. experiments or tests) can only be used to validate the truth of general statements for a finite number of instances. In contrast, the strength of mathematical proof is that it makes it possible to show the truth of statements *for all* instances.

Mathematical proofs are produced and checked. Their production requires intuition and creativity, at least as far as their statement is concerned. Checking an existing proof is, on the other hand, a mechanical process that can be carried out by both humans and machines. Automating this process is valuable, because a human can then be convinced of the correctness of an argument without performing the laborious proof checking herself, as long as she is willing to trust the correctness of the automatic proof checker.

A mechanized proof checker is only useful if it is performing its task correctly and, therefore, we need to establish this correctness in a rigorous way. Of course, one way to produce such evidence is to use mathematical proof. In this work we utilize computer-aided tools called *interactive theorem provers* to produce and verify the correctness of a mechanized proof checker.

1.2 Concepts in mechanized proof checking

Before we discuss the main contribution of this work, we introduce the key concepts involved in the topic of this thesis here. In what follows, we will explain each concept, and its relevance to this work.

Formal logic. Formal logics are mathematical languages that enable us to make precise mathematical statements, and construct proofs in a mechanical way. A formal logic consists of a syntax, and a well-defined meaning of the

syntax, called a semantics. A logic also comes with a calculus of syntactic proof rules for how to construct new syntactic objects from existing ones. These rules are proven sound with respect to the semantics, meaning that they can only be used to construct syntax that is true according to the semantics. The advantage of using a formal logic is that any reasoning using the rules of the language is guaranteed to result in valid proofs.

Higher-order logic (HOL). Higher-order logic is an expressive formal logic. Its expressivity allows it to both describe the syntax and semantics of a computer program implementation of a mechanized proof checker, and to act as the programming language for such an implementation. The latter is not only convenient, but also allows us to draw very strong conclusions about the correctness of our programs.

Interactive theorem provers (ITPs). Interactive theorem provers are programs designed to aid reasoning in a formal logic. They are called *interactive* because human interaction is required to guide the system when carrying out proof (even though ITPs allow for a significant degree of automation). These proofs are checked by the system, meaning that the user can trust any theorem produced by the system, as long as she trusts the system itself.

The LCF-approach. The LCF-approach is a method of designing ITP systems in a way that enables extensibility without compromising soundness. To this end, theorems are modeled as an abstract data type in a functional programming language (called ML, for Meta Language), accessible only by means of functions corresponding to the primitive inferences (i.e. the basic rules) of the logic. The LCF-approach was developed as part of the Edinburgh LCF system [13], but the LCF-style design is still integral to most modern ITPs.

The HOL4 theorem prover. The HOL4 theorem prover [35] is an ITP for HOL. Like most other HOL provers, it follows the LCF-approach. HOL4 includes state-of-the-art code generation techniques that we develop and make use of in this work. In addition, the system hosts the CakeML programming language and its compiler, as well as a verified implementation of a HOL logical kernel, called Candle. Both CakeML and Candle are discussed below.

The LCF-style design of HOL4 ensures that all proofs carried out in the system are reduced to a fixed set of primitive inferences. As a consequence, it is possible to record proofs, by logging which inferences were used. These proofs can then be checked by external programs, e.g. a checker for OpenTheory articles; see below.

The OpenTheory framework. A mechanized proof checker requires a data representation for the proofs it checks. One such representation is the OpenTheory article format [20], which is part of the OpenTheory framework [19]. Arti-

cles in the OpenTheory format provide a means to record and store proofs of HOL theorems in a way that is supported by several HOL ITPs. In addition to this, the OpenTheory framework includes its own proof checking tool [21].

The CakeML language and tools. CakeML is a functional programming language that comes with a verified compiler [36], and a proof-producing code generation mechanism for the HOL4 system [30]. Using the CakeML tools, it is possible to synthesize executable programs from functions in the HOL4 logic, i.e. HOL. The correctness result of the CakeML compiler guarantees that the resulting executables behave as their logical counterparts. These techniques have been used to produce a verified implementation of HOL called Candle, discussed below.

The Candle theorem prover kernel. The Candle theorem prover kernel [23] is a verified implementation of an LCF-style kernel for HOL. The Candle kernel is verified to be sound with respect to the semantics of HOL, meaning that the kernel is guaranteed to accept only valid proof steps. Its verification was carried out using the HOL4 system by Kumar, et al. [23], and the CakeML tools can be used to produce an executable version of the kernel.

A verified OpenTheory proof checker. The OpenTheory proof checker is a mechanized proof checker that reads OpenTheory articles, and uses the Candle kernel to check the validity of inferences. Incorporating the Candle kernel into our proof checker enables us to build on its soundness result. The proof checker is compiled to executable machine code using the CakeML compiler, which is semantics preserving. As a result, we obtain a soundness result about the resulting machine code.

1.3 Contributions

This Licentiate thesis makes the following contributions:

(i) We extend existing techniques for proof-producing code generation to support a larger class of programs. We show how these techniques can be used to develop software with very strong end-to-end correctness guarantees that reach down to the machine code that actually runs the software.

(ii) The main product of the work described in this thesis is a new proof checker for higher-order logic that is verified to be sound. As a consequence of using the CakeML tools, we are able to obtain the same soundness result for the machine code that executes the proof checker. To the best of our knowledge, this is the only proof checker for HOL that has been verified to this degree of rigor.

1.4 Summary of included papers

This Licentiate thesis consists of the following two papers.

I Oskar Abrahamsson, Son Ho, Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Yong Kiam Tan. **Proof-Producing Synthesis of CakeML from Monadic HOL Functions.** Published in Springer's Journal of Automated Reasoning, 2020.

II Oskar Abrahamsson. **A verified proof checker for higher-order logic.** Published in Elsevier's Journal of Logic and Algebraic Methods in Programming, 2020.

Both papers appear in this document unedited, with the exception of adjustments in typesetting.

1.4.1 Proof-Producing Synthesis of CakeML from Monadic HOL Functions

Paper I, "Proof Producing Synthesis of CakeML from Monadic HOL Functions," introduces a tool which makes it possible to perform programming in HOL, using state, and effects such as input and output (I/O), and exceptions. For the uninitiated, one can understand this as: programming using the HOL4 logic, and automatically translating those programs to equivalent CakeML code. The technical contribution is based on is an extension of previous work on synthesis of non-effectful CakeML programs [30]. See Chapter 2 in this thesis for Paper I.

We say that the tool is *proof-producing* because each run of the tool derives a proof of correspondence, called a certificate, that relates the input logical functions with the synthesized program output. The certificate guarantees that execution of the resulting CakeML program will compute the same values, and modify the state in the same way, as the input logical functions. As a consequence, any verification result about the logical input functions can be made into a result about the synthesized CakeML code.

All useful programs (i.e. those programs that produce something observable) perform *side effects*. By side effects, we mean operations such as externally visible modifications to memory, and performing I/O. The work in this paper utilizes monads [37] to allow us to write programs that produce side-effects inside the logic, thereby granting us greater expressivity when using HOL as a programming language.

These contributions were crucial to the development of the work in Paper II, which is described below.

Statement of contribution. I contributed to the writing of this paper, particularly Section 2.7. I implemented some of the examples discussed in this paper, including the OpenTheory proof checker, and some other examples included in the source code repository for the tool.

1.4.2 A verified proof checker for higher-order logic

Paper II, "A verified proof checker for higher-order logic," introduces a mechanized proof checker for proofs of theorems in HOL that is verified to be sound down to the level of machine code that executes it. To the best of our knowledge, it is the only proof checker for HOL that has been verified to this degree of rigor. See Chapter 3 in this thesis for Paper II.

The checker itself is a computer program, implemented using HOL as a programming language. It reads proofs of HOL theorems represented in the OpenTheory article format [20] as input and uses the Candle kernel [23] to check proof steps, and outputs a verdict stating whether the proof was valid.

The proof checker is verified to be *sound* with respect to the semantics of HOL, meaning that it is guaranteed to accept only proofs of true theorems. We are able to obtain this soundness result because the checker uses the Candle theorem prover kernel [23], which is verified to be sound, as its logical kernel.

This paper improves on the state-of-the-art by: (i) establishing a particularly strong soundness result for the proof checker; and (ii) showing how such a result can be transported to the level of the compiled machine code. The techniques presented in Paper I are used to synthesize stateful CakeML from the proof checker function in the logic, and to transport its soundness theorem to the level of CakeML code. This CakeML program is compiled to executable machine code in a proof-producing way, using the CakeML compiler [36] inside HOL4. Our approach allows us to obtain the soundness result of the checker also for the machine code that executes it.

Statement of contribution. I am the sole author of this article. All work is my own, aside from the initial implementation of the OpenTheory abstract machine, which was done by Ramana Kumar before my work started.

Chapter 2

Proof-Producing Synthesis of CakeML from Monadic HOL Functions

Oskar Abrahamsson, Son Ho, Hrutvik Kanabar, Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Yong Kiam Tan

Abstract. We introduce an automatic method for producing stateful ML programs together with proofs of correctness from monadic functions in HOL. Our mechanism supports references, exceptions, and I/O operations, and can generate functions manipulating local state, which can then be encapsulated for use in a pure context. We apply this approach to several non-trivial examples, including the instruction encoder and register allocator of the otherwise pure CakeML compiler, which now benefits from better runtime performance. This development has been carried out in the HOL4 theorem prover.

Published in *Journal of Automated Reasoning*, 2020.

2.1 Introduction

This paper is about bridging the gap between programs verified in logic and verified implementations of those programs in a programming language (and ultimately machine code). As a toy example, consider computing the nth Fibonacci number. The following is a recursion equation for a function, fib, in higher-order logic (HOL) that does the job:

$$\mathsf{fib}\ n = \mathsf{if}\ n < 2\ \mathsf{then}\ n\ \mathsf{else}\ \mathsf{fib}\ (n-1) + \mathsf{fib}\ (n-2)$$

A hand-written implementation (shown here in CakeML [24], which has similar syntax and semantics to Standard ML) would look something like this:

```
fun fiba i j n = if n = 0 then i else fiba j (i+j) (n-1);
(print (n2s (fiba 0 1 (s2n (hd (CommandLine.arguments())))));
 print "\n")
handle _ => print_err ("usage: " ^ CommandLine.name() ^ " <n>\n");
```

In moving from mathematics to a real implementation, some issues are apparent:

(i) We use a tail-recursive linear-time algorithm, rather than the exponential-time recursion equation.

(ii) The whole program is not a pure function: it does I/O, reading its argument from the command line and printing the answer to standard output.

(iii) We use exception handling to deal with malformed inputs (if the arguments do not start with a string representing a natural number, `hd` or `s2n` may raise an exception).

The first of these issues (i) can easily be handled in the realm of logical functions. We define a tail-recursive version in logic:

$$\mathsf{fiba}\ i\ j\ n = \mathsf{if}\ n = 0\ \mathsf{then}\ i\ \mathsf{else}\ \mathsf{fiba}\ j\ (i+j)\ (n-1)$$

then produce a correctness theorem, $\vdash \forall n.\ \mathsf{fiba}\ 0\ 1\ n = \mathsf{fib}\ n$, with a simple inductive proof (a 5-line tactic proof in HOL4, not shown).

Now, because fiba is a logical function with an obvious computational counterpart, we can use proof-producing synthesis techniques [30] to automatically synthesise code verified to compute it. We thereby produce something like the first line of the CakeML code above, along with a theorem relating the semantics of the synthesised code back to the function in logic.

But when it comes to handling the other two issues, (ii) and (iii), and producing and verifying the remaining three lines of CakeML code, our options are less straightforward. The first issue was easy because we were working with a *shallow embedding*, where one writes the program as a function in logic and proves properties about that function directly. Shallow embeddings rely on an

```
fibm () =
 do
  args ← commandline (arguments ());
  a ← hd args;
  n ← s2n a;
  stdio (print (n2s (fiba 0 1 n)));
  stdio (print "\n")
 od otherwise
 do
  name ← commandline (name ());
  stdio (print_err ("usage: " ^ name ^ " <n>\n"))
 od
```

Figure 2.1. The Fibonacci program written using do-notation in logic.

analogy between mathematical functions and procedures in a pure functional programming language. However, effects like state, I/O, and exceptions, can stretch this analogy too far. The alternative is a *deep embedding*: one writes the program as an input to a formal semantics, which can accurately model computational effects, and proves properties about its execution under those semantics.

Proofs about shallow embeddings are relatively easy since they are in the native language of the theorem prover, whereas proofs about deep embeddings are filled with tedious details because of the indirection through an explicit semantics. Still, the explicit semantics make deep embeddings more realistic. An intermediate option that is suitable for the effects we are interested in — state/references, exceptions, and I/O — is to use *monadic functions*: one writes (shallow) functions that represent computations, aided by a composition operator (monadic bind) for stitching together effects. The monadic approach to writing effectful code in a pure language may be familiar from the Haskell language which made it popular.

For our nth Fibonacci example, we can model the effects of the whole program with a monadic function, fibm, that calls the pure function fiba to do the calculation. Figure 2.1 shows how fibm can be written using do-notation familiar from Haskell. This is as close as we can get to capturing the effectful behaviour of the desired CakeML program while remaining in a shallow embedding. Now how can we produce real code along with a proof that it has the correct semantics? If we use the proof-producing synthesis techniques mentioned above [30], we produce *pure* CakeML code that exposes the monadic plumbing in an explicit state-passing style. But we would prefer verified *effectful* code that uses native features of the target language (CakeML) to implement the monadic effects.

In this paper, we present an automated technique for producing verified

effectful code that handles I/O, exceptions, and other issues arising in the move from mathematics to real implementations. Our technique systematically establishes a connection between shallowly embedded functions in HOL with monadic effects and deeply embedded programs in the impure functional language CakeML. The synthesised code is efficient insofar as it uses the native effects of the target language and is close to what a real implementer would write. For example, given the monadic fibm function above, our technique produces essentially the same CakeML program as on the first page (but with a `let` for every monad bind), together with a proof that the synthesised program is a refinement.

Contributions Our technique for producing verified effectful code from monadic functions builds on a previous limited approach [30]. The new generalised method adds support for the following features:

- global references and exceptions (as before, but generalised),
- mutable arrays (both fixed and variable size),
- input/output (I/O) effects,
- local mutable arrays and references, which can be integrated seamlessly with code synthesis for otherwise pure functions,
- composable effects, whereby different state and exception monads can be combined using a lifting operator, and,
- support for recursive programs where termination depends on monadic state.

As a result, we can now write *whole programs* as shallow embeddings and obtain real verified code via synthesis. Prior to this work, whole program verification in CakeML involved manual deep embedding proofs for (at the very least) the I/O wrapper. To exercise our toolchain, we apply it to several examples:

- the nth Fibonacci example already seen (exceptions, I/O)
- the Floyd Warshall algorithm for finding shortest paths (arrays)
- an in-place quicksort algorithm (polymorphic local arrays, exceptions)
- the instruction encoder in the CakeML compiler's assembler (local arrays)
- the CakeML compiler's register allocator (local refs, arrays)
- the Candle theorem prover's kernel [23] (global refs, exceptions)
- an OpenTheory [19] article checker (global refs, exceptions, I/O)

In §2.6, we compare runtimes with the previous non-stateful versions of CakeML's register allocator and instruction encoder; and for the OpenTheory reader we compare the amount of code/proof required before and after using our technique.

The HOL4 development is at `https://code.cakeml.org`; our new synthesis tool is at `https://code.cakeml.org/tree/master/translator/monadic`.

Additions. This paper is an extended version of our earlier conference paper [17]. The following contributions are new to this work: a brief discussion of how *polymorphic* functions that use type variables in their local state can be synthesized (§2.4), a section on synthesis of recursive programs where termination depends on the monadic state (§2.5), and new case studies using our tool, e.g., quicksort with polymorphic local arrays (§2.4), and the CakeML compiler's instruction encoder (§2.6).

2.2 High-level ideas

This paper combines the following three concepts in order to deliver the contributions listed above. The main ideas will be described briefly in this section, while subsequent sections will provide details. The three concepts are:

(i) synthesis of stateful ML code as described in our previous work [30],

(ii) separation logic [33] as used by characteristic formulae for CakeML [14],

(iii) a new abstract synthesis mode for the CakeML synthesis tools [30].

Our previous work on proof-producing synthesis of stateful ML (i) was severely limited by the requirement to have a hard-coded invariant on the program's state. There was no support for I/O and all references had to be declared globally. At the time of its development, we did not have a satisfactory way of generalising the hard-coded state invariant.

In this paper we show (in §2.3) that the separation logic of CF (ii) can be used to neatly generalise the hard-coded state invariant of our prior work (i). CF-style separation logic easily supports references and arrays, including resizable arrays, and, supports I/O too because it allows us to treat I/O components as if they are heap components. Furthermore, by carefully designing the integration of (i) and (ii), we retain the frame rule from the separation logic. In the context of code synthesis, this frame rule allows us to implement a lifting feature for changing the type of the state-and-exception monads. Being able to change types in the monads allows us to develop *reusable* libraries — e.g. verified file I/O functions — that users can lift into the monad that is appropriate for their application.

The combination of (i) and (ii) does not by itself support synthesis of code with local state due to inherited limitations of (i), wherein the generated code must be produced as a concrete list of global declarations. For example, if monadic functions, say `foo` and `bar`, refer to a common reference, say `r`, then `r` must be defined globally:

```
val r = ref 0;
fun foo n = ...; (* code that uses r *)
fun bar n = ...; (* code that uses r and calls foo *)
```

In this paper (in §2.4), we introduce a new *abstract* synthesis mode (iii) which removes the requirement of generating code that only consists of a list of global declarations, and, as a result, we are now able to synthesise code such as the following, where the reference `r` is a local variable:

```
fun pure_bar k n =
  let
    val r = ref k
    fun foo n = ... (* code that uses r *)
    fun bar n = ... (* code that uses r and calls foo *)
  in Success (bar n) end
  handle e => Failure e;
```

In the input to the synthesis tool, this declaration and initialisation of local state corresponds to applying the state-and-exception monad. Expressions that fully apply the state-and-exception monad can subsequently be used in the synthesis of *pure* CakeML code: the monadic synthesis tool can prove a pure specification for such expressions, thereby encapsulating the monadic features.

2.3 Generalised approach to synthesis of stateful ML code

This section describes how our previous approach to proof-producing synthesis of stateful ML code [30] has been generalised. In particular, we explain how the separation logic from our previous work on characteristic formulae [14] has been used for the generalisation (§2.3.3); and how this new approach adds support for user-defined references, fixed- and variable-length arrays, I/O functions (§2.3.4), and a handy feature for reusing state-and-exception monads (§2.3.5).

In order to make this paper as self-contained as possible, we start with a brief look at how the semantics of CakeML is defined (§2.3.1) and how our previous work on synthesis of pure CakeML code works (§2.3.2), since the new synthesis method for stateful code is an evolution of the original approach for pure code.

2.3.1 Preliminaries: CakeML semantics

The semantics of the CakeML language is defined in the *functional big-step* style [32], which means that the semantics is an interpreter defined as a functional program in the logic of a theorem prover.

The definition of the semantics is layered. At the top-level the semantics function defines what the observable I/O events are for a given whole program. However, more relevant to the presentation in this paper is the next layer down: a function called evaluate that describes exactly how expressions evaluate. The type of the evaluate function is shown below. This function takes as arguments

a state (with a type variable for the I/O environment), a value environment, and a list of expressions to evaluate. It returns a new state and a value result.

$$\mathsf{evaluate} : \delta\ \mathsf{state} \rightarrow \mathsf{v\ sem\_env} \rightarrow \mathsf{exp\ list} \rightarrow \delta\ \mathsf{state} \times (\mathsf{v\ list},\ \mathsf{v})\ \mathsf{result}$$

The semantics state is defined as the record type below. The fields relevant for this presentation are: refs, clock and ffi. The refs field is a list of store values that acts as a mapping from reference names (list index) to reference and array values (list element). The clock is a logical clock for the functional big-step style. The clock allows us to prove termination of evaluate and is, at the same time, used for reasoning about divergence. Lastly, ffi is the parametrised oracle model of the foreign function interface, i.e. I/O environment.

$$\delta\ \mathsf{state} = \langle\!|\ \mathsf{clock} : \mathsf{num}\ ;\ \mathsf{refs} : \mathsf{store\_v\ list}\ ;\ \mathsf{ffi} : \delta\ \mathsf{ffi\_state}\ ;\ \ldots\ |\!\rangle$$
$$\text{where}\ \ \mathsf{store\_v}\ =\ \mathsf{Refv\ v}\ |\ \mathsf{W8array}\ (\mathsf{word8\ list})\ |\ \mathsf{Varray}\ (\mathsf{v\ list})$$

A call to the function evaluate returns one of two results: Rval res for successfully terminating computations, and Rerr err for stuck computations.

Successful computations, Rval res, return a list res of CakeML values. CakeML values are modelled in the semantics using a datatype called v. This datatype includes (among other things) constructors for (mutually recursive) closures (Closure and Recclosure), datatype constructor values (Conv), and literal values (Litv) such as integers, strings, characters etc. These will be explained when needed in the rest of the paper.

Stuck computations, Rerr err, carry an error value err that is one of the following. For this paper, Rraise exc is the most relevant case.

- Rraise exc indicates that evaluation results in an uncaught exception exc. These exceptions can be caught with a `handle` in CakeML.
- Rabort Rtimeout_error indicates that evaluation of the expression consumes all of the logical clock. Programs that hit this error for all initial values of the clock are considered diverging.
- Rabort Rtype_error, for other kinds of errors, e.g. when evaluating ill-typed expressions, or attempting to access unbound variables.

2.3.2 Preliminaries: Synthesis of pure ML code

Our previous work [30] describes a *proof-producing* algorithm for synthesising CakeML functions from functions in higher-order logic. Here proof-producing means that each execution proves a theorem (called a certificate theorem) guaranteeing correctness of that execution of the algorithm. In our setting, these theorems relate the CakeML semantics of the synthesised code with the given HOL function.

The whole approach is centred around a systematic way of proving theorems relating HOL functions (i.e. HOL terms) with CakeML expressions. In order

for us to state relations between HOL terms and CakeML expressions, we need a way to state relations between HOL terms and CakeML values. For this we use relations (int, list $\cdot$, $\cdot \longrightarrow \cdot$, etc.) which we call refinement invariants. The definition of the simple int refinement invariant is shown below: int i v is true if CakeML value v of type v represents the HOL integer i of type int.

$$\mathsf{int}\ i = (\lambda\, v.\ v = \mathsf{Litv}\ (\mathsf{IntLit}\ i))$$

Most refinement invariants are more complicated, e.g. list (list int) xs v states that CakeML value v represents lists of int lists xs of HOL type int list list.

We now turn to CakeML expressions: we define a predicate called Eval in order to conveniently state relationships between HOL terms and CakeML expressions. The intuition is that Eval env exp P is true if exp evaluates (in environment env) to some result res (of HOL type v) such that P holds for res, i.e. P res. The formal definition below is cluttered by details regarding the clock and references: there must be a large enough clock and exp may allocate new references, $refs'$, but must not modify any existing references, $refs$. We express this restriction on the references using list append $+\!\!+$. Note that any list index that can be looked up in $refs$ has the same look up in $refs +\!\!+ refs'$.

$$\begin{array}{l}\mathsf{Eval}\ env\ exp\ P = \\ \quad \forall\, refs. \\ \quad\quad \exists\, res\ refs'. \\ \quad\quad\quad \mathsf{eval\_rel}\ (\mathsf{empty}\ with\ \mathsf{refs}\ :=\ refs)\ env\ exp \\ \quad\quad\quad\quad (\mathsf{empty}\ with\ \mathsf{refs}\ :=\ refs +\!\!+ refs')\ res \wedge P\ res\end{array}$$

The use of Eval and the main idea behind the synthesis algorithm is most conveniently described using an example. The example we consider here is the following HOL function:

$$\mathsf{add1} = (\lambda\, x.\ x + 1)$$

The main part of the synthesis algorithm proceeds as a syntactic bottom-up pass over the given HOL term. In this case, the bottom-up pass traverses HOL term $\lambda\, x.\ x + 1$. The result of each stage of the pass is a theorem stated in terms of Eval in the format shown below. Such theorems state a connection between a HOL term t and some generated $code$ w.r.t. a refinement invariant ref_inv that is appropriate for the type of t.

$$\text{general format:}\quad assumptions \Rightarrow \mathsf{Eval}\ env\ code\ (ref\_inv\ t)$$

For our little example, the algorithm derives the following theorems for the subterms x and 1, which are the leaves of the HOL term. Here and elsewhere in this paper, we display CakeML abstract syntax as concrete syntax inside $\lfloor \cdots \rfloor$, i.e. $\lfloor 1 \rfloor$ is actually the CakeML expression Lit (IntLit 1) in the theorem prover HOL4; similarly $\lfloor \mathsf{x} \rfloor$ is actually displayed as Var (Short "x") in HOL4. Note that

both theorems below are of the required general format.

$$\begin{array}{l} \vdash \mathsf{T} \Rightarrow \mathsf{Eval}\ env\ \lfloor \mathsf{1} \rfloor\ (\mathsf{int}\ 1) \\ \vdash \mathsf{Eval}\ env\ \lfloor \mathsf{x} \rfloor\ (\mathsf{int}\ x) \Rightarrow \mathsf{Eval}\ env\ \lfloor \mathsf{x} \rfloor\ (\mathsf{int}\ x) \end{array} \tag{2.1}$$

The algorithm uses theorems (2.1) when proving a theorem for the compound expression $x + 1$. The process is aided by an auxiliary lemma for integer addition, shown below. The synthesis algorithm is supported by several such pre-proved lemmas for various common operations.

$$\begin{array}{l} \vdash \mathsf{Eval}\ env\ x_1\ (\mathsf{int}\ n_1) \Rightarrow \\ \quad \mathsf{Eval}\ env\ x_2\ (\mathsf{int}\ n_2) \Rightarrow \\ \quad \mathsf{Eval}\ env\ \lfloor x_1\ \mathsf{+}\ x_2 \rfloor\ (\mathsf{int}\ (n_1\ \mathsf{+}\ n_2)) \end{array}$$

By choosing the right specialisations for the variables, x_1, x_2, n_1, n_2, the algorithm derives the following theorem for the body of the running example. Here the assumption on evaluation of $\lfloor \mathsf{x} \rfloor$ was inherited from (2.1).

$$\vdash \mathsf{Eval}\ env\ \lfloor \mathsf{x} \rfloor\ (\mathsf{int}\ x) \Rightarrow \mathsf{Eval}\ env\ \lfloor \mathsf{x\ +\ 1} \rfloor\ (\mathsf{int}\ (x + 1)) \tag{2.2}$$

Next, the algorithm needs to introduce the λ-binder in $\lambda\, x.\ x + 1$. This can be done by instantiation of the following pre-proved lemma. Note that the lemma below introduces a refinement invariant for function types, $\longrightarrow$, which combines refinement invariants for the input and output types of the function [30].

$$\begin{array}{l} \vdash (\forall\, v\ x.\ a\ x\ v \Rightarrow \mathsf{Eval}\ (env\,[n \mapsto v])\ body\ (b\ (f\ x))) \Rightarrow \\ \quad \mathsf{Eval}\ env\ \lfloor \mathsf{fn\ n\ =>}\ body \rfloor\ ((a \longrightarrow b)\ f) \end{array}$$

An appropriate instantiation and combination with (2.2) produces the following:

$$\vdash \mathsf{T} \Rightarrow \mathsf{Eval}\ env\ \lfloor \mathsf{fn\ x\ =>\ x\ +\ 1} \rfloor\ ((\mathsf{int} \longrightarrow \mathsf{int})\ (\lambda\, x.\ x + 1))$$

which, after only minor reformulation, becomes a certificate theorem for the given HOL function `add1`:

$$\vdash \mathsf{Eval}\ env\ \lfloor \mathsf{fn\ x\ =>\ x\ +\ 1} \rfloor\ ((\mathsf{int} \longrightarrow \mathsf{int})\ \mathsf{add1})$$

Additional notes. The main part of the synthesis algorithm is always a bottom-up traversal as described above. However, synthesis of recursive functions requires an additional post-processing phase which involves an automatic induction proof. We omit a detailed description of such induction proofs since we have described our solution previously [30]. However, we discuss our solution at a high level in §2.5.3 where we explain how the previously published approach has been modified to tackle monadic programs in which termination depends on the monadic state.

2.3.3 Synthesis of stateful ML code

Our algorithm for synthesis of stateful ML is very similar to the algorithm described above for synthesis of pure CakeML code. The main differences are:

- the input HOL terms must be written in a state-and-exception monad, and
- instead of Eval and $\cdot \longrightarrow \cdot$, the derived theorems use EvalM and $\cdot \longrightarrow^{M} \cdot$,

where EvalM and $\cdot \longrightarrow^{M} \cdot$ relate the monad's state to the references and foreign function interface of the underlying CakeML state (fields refs and ffi). These concepts will be described below.

Generic state-and-exception monad. The new generalised synthesis workflow uses the following state-and-exception monad $(\alpha,\ \beta,\ \gamma)$ M, where α is the state type, β is the return type, and γ is the exception type.

$$(\alpha,\ \beta,\ \gamma)\ \mathsf{M}\ =\ \alpha\ \rightarrow (\beta,\ \gamma)\ \mathsf{exc}\ \times\ \alpha$$
$$\text{where}\ \ (\beta,\ \gamma)\ \mathsf{exc}\ =\ \mathsf{Success}\ \beta\ |\ \mathsf{Failure}\ \gamma$$

We define the following interface for this monad type. Note that syntactic sugar is often used: in our case, we write do $n\ \leftarrow\ foo$; return $(bar\ n)$ od (as was done in §2.1) when we mean bind $foo\ (\lambda\, n.$ return $(bar\ n))$.

return x = $\lambda\, s.\ (\mathsf{Success}\ x,s)$

bind $x\ f$ =
  $\lambda\, s.$ case $x\ s$ of $(\mathsf{Success}\ y,s)\ \Rightarrow\ f\ y\ s\ |\ (\mathsf{Failure}\ x,s)\ \Rightarrow\ (\mathsf{Failure}\ x,s)$

x otherwise y =
  $\lambda\, s.$ case $x\ s$ of $(\mathsf{Success}\ v,s)\ \Rightarrow\ (\mathsf{Success}\ v,s)\ |\ (\mathsf{Failure}\ e,s)\ \Rightarrow\ y\ s$

Functions that update the content of state can only be defined once the state type is instantiated. A function for changing a monad M to have a different state type is introduced in §2.3.5.

Definitions and lemmas for synthesis. We define EvalM as follows. A CakeML source expression exp is considered to satisfy an execution relation P if for any CakeML state s, which is related by state_rel to the state monad state st and state assertion H, the CakeML expression exp evaluates to a result res such that the relation P accepts the transition and state_rel_frame holds for state assertion H. The auxiliary functions state_rel and state_rel_frame will be described below. The first argument ro can be used to restrict effects to

references only, as described a few paragraphs further down.

```
EvalM ro env st exp P H =
 ∀ s.
    state_rel H st s ⇒
    ∃ s2 res st2 ck.
       (evaluate (s with clock := ck) env [exp] = (s2,res)) ∧
       P st (st2,res) ∧ state_rel_frame ro H (st,s) (st2,s2)
```

In the definition above, state_rel and state_rel_frame are used to check that the user-specified state assertion H relates the CakeML states and the monad states. Furthermore, state_rel_frame ensures that the separation logic frame rule is true. Both use the separation logic set-up from our previous work on characteristic formulae for CakeML [14], where we define a function st2heap which, given a projection p and CakeML state s, turns the CakeML state into a set representation of the reference store and foreign-function interface (used for I/O).

The H in the definition above is a pair (h,p) containing a heap assertion h and the projection p. We define state_rel (h,p) st s to state that the heap assertion produced by applying h to the current monad state st must be true for some subset produced by st2heap when applied to the CakeML state s. Here $*$ is the separating conjunction and T is true for any heap.

```
state_rel (h,p) st s = (h st * T) (st2heap p s)
```

The relation state_rel_frame states: any frame F that is true separately from h st_1 for the initial state is also true for the final state; and if the references-only ro configuration is set, then the only difference in the states must be in the references and clock, i.e. no I/O operations are permitted. The ro flag is instantiated to true when a pure specification (Eval) is proved for local state (§2.4).

```
state_rel_frame ro (h,p) (st1,s1) (st2,s2) =
  (ro ⇒ ∃ refs. s2 = s1 with refs := refs) ∧
  ∀ F.
     (h st1 * F) (st2heap p s1) ⇒
     (h st2 * F * T) (st2heap p s2)
```

We prove lemmas to aid the synthesis algorithm in construction of proofs. The lemmas shown in this paper use the following definition of monad.

```
monad a b x st1 (st2,res) =
  case (x st1,res) of
    ((Success y,st),Rval [v]) ⇒ (st = st2) ∧ a y v
  | ((Failure e,st),Rerr (Rraise v)) ⇒ (st = st2) ∧ b e v
  | _ ⇒ F
```

Synthesis makes use of the following two lemmas in proofs involving monadic return and bind. For return x, synthesis proves an Eval-theorem for x. For bind,

it proves a theorem that fits the shape of the first four lines of the lemma and returns a theorem consisting of the last two lines, appropriately instantiated.

$$\vdash \mathsf{Eval}\ env\ exp\ (a\ x) \Rightarrow$$
$$\quad \mathsf{EvalM}\ ro\ env\ st\ exp\ (\mathsf{monad}\ a\ b\ (\mathsf{return}\ x))\ H$$

$$\vdash ((assums_1 \Rightarrow \mathsf{EvalM}\ ro\ env\ st\ e_1\ (\mathsf{monad}\ b\ c\ x)\ H) \wedge$$
$$\quad \forall z\ v.$$
$$\qquad b\ z\ v \wedge assums_2\ z \Rightarrow$$
$$\qquad \mathsf{EvalM}\ ro\ (env\ [n \mapsto v])\ (\mathsf{snd}\ (x\ st))\ e_2\ (\mathsf{monad}\ a\ c\ (f\ z))\ H) \Rightarrow$$
$$\quad assums_1 \wedge (\forall z.\ (\mathsf{fst}\ (x\ st) = \mathsf{Success}\ z) \Rightarrow assums_2\ z) \Rightarrow$$
$$\quad \mathsf{EvalM}\ ro\ env\ st\ \lfloor \texttt{let n = } e_1 \texttt{ in } e_2 \rfloor\ (\mathsf{monad}\ a\ c\ (\mathsf{bind}\ x\ f))\ H$$

2.3.4 References, arrays and I/O

The synthesis algorithm uses specialised lemmas when the generic state-and-exception monad has been instantiated. Consider the following instantiation of the monad's state type to a record type. The programmer's intention is that the lists are to be synthesised to arrays in CakeML and the I/O component IO_fs is a model of a file system (taken from a library).

$$\mathsf{example\_state} =$$
$$\quad \langle\!|\ \mathsf{ref1} : \mathsf{int};\ \mathsf{farray1} : \mathsf{int\ list};\ \mathsf{rarray1} : \mathsf{int\ list};\ \mathsf{stdio} : \mathsf{IO\_fs}\ |\!\rangle$$

With the help of getter- and setter-functions and library functions for file I/O, users can conveniently write monadic functions that operate over this state type.

When it comes to synthesis, the automation instantiates H with an appropriate heap assertion, in this instance: ASSERT. The user has informed the synthesis tool that farray1 is to be a fixed-size array and rarray1 is to be a resizable-size array. A resizable-array is implemented as a reference that contains an array, since CakeML (like SML) does not directly support resizing arrays. Below, REF_REL int ref1_loc st.ref1 asserts that int relates the value held in a reference at a fixed store location ref1_loc to the integer in st.ref1. Similarly, ARRAY_REL and RARRAY_REL specify a connection for the array fields. Lastly, STDIO is a heap assertion for the file I/O taken from a library.

$$\mathsf{ASSERT}\ st =$$
$$\quad \mathsf{REF\_REL\ int\ ref1\_loc}\ st.\mathsf{ref1} * \mathsf{RARRAY\_REL\ int}\ rarray1\_loc\ st.\mathsf{rarray1} *$$
$$\quad \mathsf{ARRAY\_REL\ int\ farray1\_loc}\ st.\mathsf{farray1} * \mathsf{STDIO}\ st.\mathsf{stdio}$$

Automation specialises pre-proved EvalM lemmas for each term that might be encountered in the monadic functions. As an example, a monadic function might contain an automatically defined function update_farray1 for updating array farray1. Anticipating this, synthesis automation can, at set-up time, automatically derive the following lemma which it can use when it encounters

update_farray1.

$$\vdash \mathsf{Eval}\ env\ e_1\ (\mathsf{num}\ n) \wedge \mathsf{Eval}\ env\ e_2\ (\mathsf{int}\ x) \wedge (\mathsf{lookup\_var}\ \lfloor \texttt{farray1} \rfloor\ env = \mathsf{Some\ farray1\_loc}) \Rightarrow \mathsf{EvalM}\ ro\ env\ st\ \lfloor \texttt{Array.update (farray1,}e_1\texttt{,}e_2\texttt{)} \rfloor\ (\mathsf{monad\ unit\ exc}\ (\mathsf{update\_farray1}\ n\ x))\ (\mathsf{ASSERT},p)$$

2.3.5 Combining monad state types

Previously developed monadic functions (e.g. from an existing library) can be used as part of a larger context, by combining state-and-exception monads with different state types. Consider the case of the file I/O in the example from above. The following EvalM theorem has been proved in the CakeML basis library.

$$\vdash \mathsf{Eval}\ env\ e\ (\mathsf{string}\ x) \wedge (\mathsf{lookup\_var}\ \lfloor \texttt{print} \rfloor\ env = \mathsf{Some\ print\_v}) \Rightarrow \mathsf{EvalM\ F}\ env\ st\ \lfloor \texttt{print}\ e \rfloor\ (\mathsf{monad\ unit}\ b\ (\mathsf{print}\ x))\ (\mathsf{STDIO},p)$$

This can be used directly if the state type of the monad is the IO_fs type. However, our example above uses example_state as the state type.

To overcome such type mismatches, we define a function liftM which can bring a monadic operation defined in libraries into the required context. The type of liftM $r\ w$ is $(\alpha,\ \beta,\ \gamma)\ \mathsf{M} \rightarrow (\epsilon,\ \beta,\ \gamma)\ \mathsf{M}$, for appropriate r and w.

$$\mathsf{liftM}\ r\ w\ op = \lambda s.\ \mathsf{let}\ (ret,new) = op\ (r\ s)\ \mathsf{in}\ (ret,w\ (\mathsf{K}\ new)\ s)$$

Our liftM function changes the state type. A simpler lifting operation can be used to change the exception type.

For our example, we define stdio f as a function that performs f on the IO_fs-part of a example_state. (The fib example in §2.1 used a similar stdio.)

$$\mathsf{stdio} = \mathsf{liftM}\ (\lambda s.\ s.\mathsf{stdio})\ (\lambda f\ s.\ s\ with\ \mathsf{stdio}\ updated\_by\ f)$$

Our synthesis mechanism automatically derives a lemma that can transfer any EvalM result for the file I/O model to a similar EvalM result wrapped in the stdio function. Such lemmas are possible because of the separation logic frame rule that is part of EvalM. The generic lemma is the following:

$$\vdash (\forall st.\ \mathsf{EvalM}\ ro\ env\ st\ exp\ (\mathsf{monad}\ a\ b\ op)\ (\mathsf{STDIO},p)) \Rightarrow \forall st.\ \mathsf{EvalM}\ ro\ env\ st\ exp\ (\mathsf{monad}\ a\ b\ (\mathsf{stdio}\ op))\ (\mathsf{ASSERT},p)$$

And the following is the transferred lemma, which enables synthesis of HOL terms of the form stdio (print x) for Eval-synthesisable x.

$$\vdash \mathsf{Eval}\ env\ e\ (\mathsf{string}\ x) \wedge (\mathsf{lookup\_var}\ \lfloor \texttt{print} \rfloor\ env = \mathsf{Some\ print\_v}) \Rightarrow \mathsf{EvalM\ F}\ env\ st\ \lfloor \texttt{print}\ e \rfloor\ (\mathsf{monad\ unit\ exc}\ (\mathsf{stdio}\ (\mathsf{print}\ x)))\ (\mathsf{ASSERT},p)$$

Changing the monad state type comes at no additional cost to the user; our tool is able to derive both the generic and transferred EvalM lemmas, when provided with the original EvalM result.

2.4 Local state and the abstract synthesis mode

This section explains how we have adapted the method described above to also support generation of code that uses local state and local exceptions. These features enable use of stateful code (EvalM) in a pure context (Eval). We used these features to significantly speed up parts of the CakeML compiler (see §2.6).

In the monadic functions, users indicate that they want local state to be generated by using the following run function. In the logic, the run function essentially just applies a monadic function m to an explicitly provided state st.

$$\begin{array}{l}\mathsf{run} : (\alpha,\ \beta,\ \gamma)\ \mathsf{M} \rightarrow \alpha \rightarrow (\beta,\ \gamma)\ \mathsf{exc} \\ \mathsf{run}\ m\ st = \mathsf{fst}\ (m\ st)\end{array}$$

In the generated code, an application of run to a concrete monadic function, say bar, results in code of the following form:

```
fun run_bar k n =
  let
   val r = ref ... (* allocate, initialise, let-bind all local state *)
    fun foo n = ... (* all auxiliary funs that depend on local state *)
     fun bar n = ... (* define the main monadic function *)
 in Success (bar n) end (* wrap normal result in Success constructor *)
 handle e => Failure e; (* wrap any exception in Failure constructor *)
```

Synthesis of locally effectful code is made complicated in our setting for two reasons: (i) there are no fixed locations where the references and arrays are stored, e.g. we cannot define ref1_loc as used in the definition of ASSERT in §2.3.4; and (ii) the local names of state components must be in scope for all of the function definitions that depend on local state.

Our solution to challenge (i) is to leave the location values as variables (loc_1, loc_2, loc_3) in the heap assertion when synthesising local state. To illustrate, we will adapt the example_state from §2.3.4: we omit IO_fs in the state because I/O cannot be made local. The local-state enabled heap assertion is:

$$\begin{array}{l}\mathsf{LOCAL\_ASSERT}\ loc_1\ loc_2\ loc_3\ st = \\ \quad \mathsf{REF\_REL\ int}\ loc_1\ st.\mathsf{ref1} * \mathsf{RARRAY\_REL\ int}\ loc_2\ st.\mathsf{rarray1} * \\ \quad \mathsf{ARRAY\_REL\ int}\ loc_3\ st.\mathsf{farray1}\end{array}$$

The lemmas referring to local state now assume they can find the right variable locations with variable look-ups.

$$\begin{array}{l}\vdash \mathsf{Eval}\ env\ e_1\ (\mathsf{num}\ n) \wedge \mathsf{Eval}\ env\ e_2\ (\mathsf{int}\ x) \wedge \\ \quad (\mathsf{lookup\_var}\ \lfloor \texttt{farray1} \rfloor)\ env = \mathsf{Some}\ loc_3) \Rightarrow \\ \quad \mathsf{EvalM}\ ro\ env\ st\ \lfloor \texttt{Array.update (farray1,}e_1\texttt{,}e_2\texttt{)} \rfloor \\ \quad\quad (\mathsf{monad\ unit\ exc}\ (\mathsf{update\_farray1}\ n\ x))\ (\mathsf{LOCAL\_ASSERT}\ loc_1\ loc_2\ loc_3,p)\end{array}$$

Challenge (ii) was caused by technical details of our previous synthesis methods. The previous version was set up to only produce top-level declarations,

which is incompatible with the requirement to have local (not globally fixed) state declarations shared between several functions. The requirement to only have top-level declarations arose from our desire to keep things simple: each synthesised function is attached to the end of a concrete linear program that is being built. It is beneficial to be concrete because then each assumption on the lexical environment where the function is defined can be proved immediately on definition. We will call this old approach the *concrete mode* of synthesis, since it eagerly builds a concrete program.

In order to support having functions access local state, we implement a new *abstract mode* of synthesis. In the abstract mode, each assumption on the lexical environment is left as an unproved side condition as long as possible. This allows us to define functions in a dynamic environment.

To prove a pure specification (Eval) from the EvalM theorems, the automation first proves that the generated state-allocation and -initialisation code establishes the relevant heap assertion (e.g. LOCAL_ASSERT); it then composes the abstractly synthesised code while proving the environment-related side conditions (e.g. presence of loc_3). The final proof of an Eval theorem requires instantiating the references-only ro flag to true, in order to know that no I/O occurs (§2.3.3).

Type variables in local monadic state

Our previous approach [30] allowed synthesis of (pure) polymorphic functions. Our new mechanism is able to support the same level of generality by permitting type variables in the type of monadic state that is used locally. As an example, consider a monadic implementation of an in-place quicksort algorithm, quicksort, with the following type signature:

$$\begin{array}{l}\mathsf{quicksort}\ :\ \alpha\ \mathsf{list}\ \rightarrow\ (\alpha\ \rightarrow\ \alpha\ \rightarrow\ \mathsf{bool})\ \rightarrow\ (\alpha\ \mathsf{state},\ \alpha\ \mathsf{list},\ \mathsf{exn})\ \mathsf{M}\\ \text{where}\ \ \alpha\ \mathsf{state}\ =\ \langle\!|\ \mathsf{arr}\ :\ \alpha\ \mathsf{list}\ |\!\rangle\end{array}$$

The function quicksort takes a list of values of type α and an ordering on α as input, producing a sorted list as output. However, internally it copies the input list into a mutable array in order to perform fast in-place random accesses.

The heap assertion for α state is called POLY_ASSERT, and is defined below:

$$\mathsf{POLY\_ASSERT}\ A\ loc\ st\ =\ \mathsf{RARRAY\_REL}\ A\ loc\ st.\mathsf{arr}$$

Here, A is a refinement invariant for logical values of type α. This parametrisation over state type variables is similar to the way in which location values were parametrised to solve challenge (i) above.

Applying run to quicksort, and synthesising CakeML from the result gives the following certificate theorem which makes the stateful quicksort callable from pure translations.

$$\begin{array}{l}\vdash\ (\mathsf{list}\ a\ \longrightarrow\ (a\ \longrightarrow\ a\ \longrightarrow\ \mathsf{bool})\ \longrightarrow\ \mathsf{exc\_type}\ (\mathsf{list}\ a)\ \mathsf{exn})\\ \quad \mathsf{run\_quicksort}\ \lfloor\mathtt{run\_quicksort}\rfloor\end{array}$$

Here exc_type (list a) exn is the refinement invariant for type (α list, exn) exc.

For the quicksort example, we have manually proved that quicksort will always return a Success value, provided the comparison function orders values of type α. The result of this effort is CakeML code for quicksort that uses state internally, but can be used as if it is a completely pure function without any use of state or exceptions.

2.5 Termination that depends on monadic state

In this section, we describe how the proof-producing synthesis method in §2.3 has been extended to deal with a class of recursive monadic functions whose termination depends on the state hidden in the monad. This class of functions creates new difficulties, as (i) the HOL4 function definition system is unable to prove termination of these functions; and, (ii) our synthesis method relies on induction theorems produced by the definition system to discharge preconditions during synthesis.

We address issue (i) by extending the HOL4 definition system with a set of congruence rewrites for the monadic bind operation, bind (§2.5.2). We then explain, at a high level, how the proof-producing synthesis in §2.3 is extended to deal with the preconditions that arise when synthesising code from recursive monadic functions (§2.5.3).

We begin with a brief overview of how recursive function definitions are handled by the HOL4 function definition system (§2.5.1).

2.5.1 Preliminaries: function definitions in HOL4

In order to accept recursive function definitions, the HOL4 system requires a well-founded relation to be found between the arguments of the function, and those of recursive applications. The system automatically extracts conditions that this relation must satisfy, attempts to guess a well-founded relation based on these conditions, and then uses this relation to solve the termination goal.

Function definitions involving higher-order functions (e.g. bind) sometimes causes the system to derive unprovable termination conditions, if it cannot extract enough information about recursive applications. When this occurs, the user must provide a congruence theorem that specifies the context of the higher-order function. The system uses this theorem to derive correct termination conditions, by rewriting recursive applications.

2.5.2 Termination of recursive monadic functions

By default, the HOL4 system is unable to automatically prove termination of recursive monadic functions involving bind. To aid the system in extracting provable termination conditions, we introduce the following congruence

theorem for bind:

$$\begin{array}{l}\vdash (x = x') \wedge (s = s') \wedge \\ \quad (\forall\, y\ s''.\ (x'\ s' = (\mathsf{Success}\ y, s'')) \Rightarrow (f\ y\ s'' = f'\ y\ s'')) \Rightarrow \\ \quad (\mathsf{bind}\ x\ f\ s = \mathsf{bind}\ x'\ f'\ s')\end{array} \tag{2.3}$$

Theorem (2.3) expresses a rewrite of the term bind $x\ f\ s$ in terms of rewrites involving its component subterms (x, f, and s), but allows for the assumption that $x'\ s'$ (the rewritten effect) must execute successfully.

However, rewriting definitions with (2.3) is not always sufficient: in addition to ensuring that the effect x in bind $x\ f$ executed successfully, the HOL4 system must also know the value and state resulting from its execution. This problem arises because the monadic state argument to bind is left implicit in user definitions. We address this issue by rewriting the defining equations of monadic functions using η-expansion before passing them to the definition system, making all partial bind applications syntactically fully applied. The whole process is automated so that it is opaque to the user, allowing definition of recursive monadic functions with no additional effort.

2.5.3 Synthesising ML from recursive monadic functions

The proof-producing synthesis method described in §2.3.2 is syntax-directed and proceeds in a bottom-up manner. For recursive functions, a tweak to this strategy is required, as bottom-up traversal would require any recursive calls to be treated before the calling function (this is clearly cyclic).

We begin with a brief explanation of how our previous (pure) synthesis tool [30] tackles recursive functions, before outlining how our new approach builds on this.

Pure recursive functions. As an example, consider the function gcd that computes the greatest common divisor of two positive integers:

$$\mathsf{gcd}\ m\ n = \mathsf{if}\ n > 0\ \mathsf{then}\ \mathsf{gcd}\ n\ (m\ \mathsf{mod}\ n)\ \mathsf{else}\ m$$

Before traversing the function body of gcd in a bottom-up manner, we simply assume the desired Eval result to hold for all recursive applications in the function definition, and record their arguments during synthesis. This results in the following Eval theorem for gcd (where Eq is defined as $\mathsf{Eq}\ a\ x = (\lambda\, y\ v.\ (x = y) \wedge a\ y\ v)$, and is used to record arguments for recursive applications):

$$\begin{array}{l}\vdash (n > 0 \Rightarrow \\ \quad \mathsf{Eval}\ env\ \lfloor\mathsf{gcd}\rfloor\ ((\mathsf{Eq\ int}\ n \longrightarrow \mathsf{Eq\ int}\ (m\ \mathsf{mod}\ n) \longrightarrow \mathsf{int})\ \mathsf{gcd})) \Rightarrow \\ \quad \mathsf{Eval}\ env\ \lfloor\mathsf{gcd}\rfloor\ ((\mathsf{Eq\ int}\ m \longrightarrow \mathsf{Eq\ int}\ n \longrightarrow \mathsf{int})\ \mathsf{gcd})\end{array} \tag{2.4}$$

and below is the desired Eval result for gcd:

$$\vdash \mathsf{Eval}\ env\ \lfloor\mathsf{gcd}\rfloor\ ((\mathsf{Eq\ int}\ m \longrightarrow \mathsf{Eq\ int}\ n \longrightarrow \mathsf{int})\ \mathsf{gcd}) \tag{2.5}$$

Theorems (2.4) and (2.5) match the shape of the hypothesis and conclusion (respectively) of the induction theorem for gcd:

$$\vdash (\forall\, m\ n.\ (n > 0 \Rightarrow P\ n\ (m \bmod n)) \Rightarrow P\ m\ n) \Rightarrow \forall\, m\ n.\ P\ m\ n$$

By instantiating this induction theorem appropriately, the preconditions in (2.4) can be discharged (and if automatic proof fails, the goal is left for the user to prove).

Monadic recursive functions. Function definitions whose termination depends on the monad give rise to induction theorems which also depend on the monad. This creates issues, as the monad argument is left implicit in the definition. As an example, here is a function linear_search that searches through an array for a value:

```
linear_search val idx =
  do
    len ← arr_length;
    if idx ≥ len then return None else
      do
        elem ← arr_sub idx;
        if elem = val then return (Some idx) else linear_search val (idx + 1)
      od
  od
```

When given the above definition, the HOL4 system automatically derives the following induction theorem:

$$\begin{array}{l}\vdash (\forall\, val\ idx\ s.\\ \quad (\forall\, len\ s'\ elem\ s''.\\ \quad\quad (\text{arr\_length}\ s = (\text{Success}\ len,s')) \wedge \neg(idx \geq len) \wedge\\ \quad\quad (\text{arr\_sub}\ idx\ s' = (\text{Success}\ elem,s'')) \wedge elem \neq val \Rightarrow\\ \quad\quad P\ val\ (idx + 1)\ s'') \Rightarrow\\ \quad P\ val\ idx\ s) \Rightarrow\\ \forall\, val\ idx\ s.\ P\ val\ idx\ s\end{array} \tag{2.6}$$

The context of recursive applications (arr_length and arr_sub) has been extracted correctly by HOL4, using the congruence theorem (2.3) and automated η-expansion for bind (see §2.5.2).

However, there is now a mismatch between the desired form of the EvalM result and the conclusion of the induction theorem: the latter depends explictly on the state, but the function depends on it only implicitly. We have modified our synthesis tool to account for this, in order to correctly discharge the necessary preconditions as above. When preconditions cannot be automatically discharged, they are left as proof obligations to the user, and the partial results derived are saved in the HOL4 theorem database.

2.6 Case studies and experiments

In this section, we present the runtime and proof size results of applying our method to some case studies.

Register allocation. The CakeML compiler's register allocator is written with a state (and exception) monad but it was previously synthesized to pure CakeML code. We updated it to use the new synthesis tool, resulting in the automatic generation of stateful CakeML code. The allocator benefits significantly from this change because it can now make use of CakeML arrays via the synthesis tool. It was previously confined to using tree-like functional arrays for its internal state, leading to logarithmic access overheads. This is not a specific issue for the CakeML compiler; a verified register allocator for CompCert [8] also reported log-factor overheads due to (functional) array accesses.

Tests were carried out using versions of the bootstrapped CakeML compiler. We ran each test 50 times on the same input program, recording time elapsed in each compiler phase. For each test, we also compared the resulting executables 10 times, to confirm that both compilers generated code of comparable quality (i.e. runtime performance). Performance experiments were carried out on an Intel i7-2600 running at 3.4GHz with 16 GB of RAM. The results are summarized in Table 2.1. Full data is available at `https://cakeml.org/ijcar18.zip`. [1]

Table 2.1. Compilation and run times (in seconds) for various CakeML benchmarks. These compare a version of the CakeML compiler where the register allocator is purely functional (old) against a version which uses local state and arrays (new).

Timing	**Benchmark**					
	knuth-bendix	smith-normal-form	tail-fib	pidig-its	life	logic
Compile (old)	18.15	16.34	8.86	9.16	9.51	12.31
Run (old)	19.58	23.53	16.60	15.47	25.59	23.33
Compile (new)	1.21	1.46	0.99	1.02	1.05	1.62
Run (new)	19.90	22.91	16.70	15.64	24.17	22.33

In the largest program (`knuth-bendix`), the new register allocator ran 15 times faster (with a wide 95% CI of 11.76–20.93 due in turn to a high standard deviation on the runtimes for the old code). In the smaller `pidigits` benchmark, the new register allocator ran 9.01 times faster (95% CI of 9.01–9.02).

[1] These tests were performed for the earlier conference version of this paper [17] comparing two earlier versions of the CakeML compiler. The compiler has changed significantly since then but we have we kept these experiments because they provide a fairer comparison of register allocation performance with/without using the synthesis tool to generate stateful code.

Across 6 example input programs, we saw ratios of runtimes between 7.58 and 15.06. Register allocation was previously such a significant part of the compiler runtime that this improvement results in runtime improvements for the whole compiler (on these benchmark programs) of factors between 2 and 9 times.

Speeding up the CakeML compiler. The register allocator exemplifies one way the synthesis tool can be used to improve existing, verified CakeML programs and in particular, the CakeML compiler itself. Briefly, the steps are: (i) re-implement slow parts of the compiler with, e.g., an appropriate state monad, (ii) verify that this new implementation produces the same result as the existing (verified) implementation, (iii) swap in the new implementation, which synthesizes to stateful code, during the bootstrap of the CakeML compiler. (iv) The preceeding steps can be repeated as desired, relying on the automated synthesis tool for quick iteration.

As another example, we used the synthesis tool to improve the assembly phase of the compiler. A major part of time spent in this phase is running the instruction encoder, which performs several word arithmetic operations when it computes the byte-level representation of each instruction. However, duplicate instructions appear very frequently, so we implemented a cache of the byte-level representations backed by a hash table represented as a state monad (i). This caching implementation is then verified (ii), before a verified implementation is synthesized where the hash table is implemented as an array (iii). We also iterated through several candidate hash functions (iv). Overall, this change took about 1-person week to implement, verify, and integrate in the CakeML compiler. We benchmarked the cross-compile bootstrap times of the CakeML compiler after this change to measure its impact across different CakeML compilation targets. Results are summarized in Table 2.2. Across compilation targets, the assembly phase is between 1.25 to 1.64 times faster.

Table 2.2. CakeML compiler cross-compile bootstrap time (in seconds) spent in the assembly phase for its various compilation targets. † For the ARMv8 target, the cross-compile bootstrap does not run to completion at the point of writing. This is for reasons unrelated to the changes in this paper.

Timing	**Cross-Compilation Target**				
	ARMv6	ARMv8 (†)	MIPS	RISC-V	x64
Assembly (old)	8.86	-	8.69	9.21	8.27
Assembly (new)	6.43	-	6.94	6.7	5.04

OpenTheory article checker. The type changing feature from §2.3.5 enabled us to produce an OpenTheory [19] article checker with our new synthesis approach, and reduce the amount of manual proof required in a previous version. The checker reads articles from the file system, and performs each logical

inference in the OpenTheory framework using the verified Candle kernel [23]. Previously, the I/O code for the checker was implemented in stateful CakeML, and verified manually using characteristic formulae. By replacing the manually verified I/O wrapper by monadic code we removed 400 lines of tedious manual proof.

2.7 Related work

Effectful code using monads. Our work on encapsulating stateful computations (§2.4) in pure programs is similar in purpose to that of the ST monad [26]. The main difference is how this encapsulation is performed: the ST monad relies on parametric polymorphism to prevent references from escaping their scope, whereas we utilise lexical scoping in synthesised code to achieve a similar effect.

Imperative HOL by Bulwahn et al. [9] is a framework for implementing and reasoning about effectful programs in Isabelle/HOL. Monadic functions are used to describe stateful computations which act on the heap, in a similar way as §2.3 but with some important differences. Instead of using a state monad, the authors introduce a polymorphic *heap monad* – similar in spirit to the ST monad, but without encapsulation – where polymorphism is achieved by mapping HOL types to the natural numbers. Contrary to our approach, this allows for heap elements (e.g. references) to be declared on-the-fly and used as first-class values. The drawback, however, is that only countable types can be stored on the heap; in particular, the heap monad does not admit function-typed values, which our work supports.

More recently, Lammich [25] has built a framework for the refinement of pure data structures into imperative counterparts, in Imperative HOL. The refinement process is automated, and refinements are verified using a program logic based on separation logic, which comes with proof-tools to aid the user in verification.

Both developments [9, 25] differ from ours in that they lack a verified mechanism for extracting executable code from shallow embeddings. Although stateful computations are implemented and verified within the confines of higher-order logic, Imperative HOL relies on the unverified code-generation mechanisms of Isabelle/HOL. Moreover, neither work presents a way to deal with I/O effects.

Verified compilation. Mechanisms for synthesising programs from shallow embeddings defined in the logics of interactive theorem provers exist as components of several verified compiler projects [5, 18, 29, 30]. Although the main contribution of our work is proof-producing synthesis, comparisons are relevant as our synthesis tool plays an important part in the CakeML compiler [24]. To the best of our knowledge, ours is the first work combining effectful computations with proof-producing synthesis and fully verified compilation.

CertiCoq by Anand et al. [5] strives to be a fully verified optimising compiler for functional programs implemented in Coq. The compiler front-end supports the full syntax of the dependently typed logic Gallina, which is reified into a deep embedding and compiled to Cminor through a series of verified compilation steps [5]. Contrary to the approach we have taken [30] (see §2.3.2), this reification is neither verified nor proof-producing, and the resulting embedding has no formal semantics (although there are attempts to resolve this issue [6]). Moreover, as of yet, no support exists for expressing effectful computations (such as in §2.3.4) in the logic. Instead, effects are deferred to wrapper code from which the compiled functions can be called, and this wrapper code must be manually verified.

The Œuf compiler by Mullen et al. [29] is similar in spirit to CertiCoq in that it compiles pure Coq functions to Cminor through a verified process. Similarly, compiled functions are pure, and effects must be performed by wrapper code. Unlike CertiCoq, Œuf supports only a limited subset of Gallina, from which it synthesises deeply embedded functions in the Œuf-language. The Œuf language has both denotational and operational semantics, and the resulting syntax is automatically proven equivalent with the corresponding logical functions through a process of computational denotation [29].

Hupel and Nipkow [18] have developed a compiler from Isabelle/HOL to CakeML AST. The compiler satisfies a partial correctness guarantee: if the generated CakeML code terminates, then the result of execution is guaranteed to relate to an equality in HOL. Our approach proves termination of the code.

2.8 Conclusion

This paper describes a technique that makes it possible to synthesise whole programs from monadic functions in HOL, with automatic proofs relating the generated effectful code to the original functions. Using the separation logic from characteristic formulae for CakeML, the synthesis mechanism supports references, exceptions, I/O, reusable library developments, encapsulation of locally stateful computations inside pure functions, and code generation for functions where termination depends on state. To our knowledge, this is the first proof-producing synthesis technique with the aforementioned features.

We hope that the techniques developed in this paper will allow users of the CakeML tools to develop verified code using only shallow embeddings. We hope that only expert users, who develop libraries, will need to delve into manual reasoning in CF or direct reasoning about deeply embedded CakeML programs.

Acknowledgements The first and fifth authors were partly supported by the Swedish Foundation for Strategic Research. The seventh author was supported by an A*STAR National Science Scholarship (PhD), Singapore. The third author

was supported by the UK Research Institute in Verified Trustworthy Software Systems (VeTSS).

Chapter 3

A verified proof checker for higher-order logic

Oskar Abrahamsson

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.

Published in *Journal of Logic and Algebraic Methods in Programming*, 2020.

3.1 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 [19]. 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) [23] 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 [35]. 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 [17] 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 [36]. 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.

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 [19]. 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 [23] 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 (§3.2). We then explain, at a high level, the steps required to produce the proof checker implementation and verify its correctness (§3.3).

We show the details of the implementation (and specification) of the tool as a shallow embedding in the logic (§3.4), and how this shallow embedding is automatically refined into an equivalent CakeML program using a proof-producing synthesis procedure (§3.5).

We compile the synthesised program into machine code, and obtain a correctness theorem relating the machine code with the shallow embedding (§3.7). 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 (§3.8).

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 (§3.9).

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.

3.2 Background

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

3.2.1 The OpenTheory framework

The purpose of the OpenTheory framework [19] 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 [35], HOL Light [16], 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.

3.2.2 The Candle theorem prover kernel

The Candle theorem prover kernel is a verified implementation of the HOL Light logical kernel by Kumar et al. [23]. The kernel is implemented as a collection of monadic functions [37] 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 [15].

As discussed in §3.2.1, 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 §3.4, where we explain how our proof-checker is constructed on top of the the Candle kernel; and in §3.8, where we show how to utilize its soundness result when verifying the end-to-end correctness of our checker.

3.2.3 The CakeML ecosystem

CakeML is a language in the style of Standard ML [28] and OCaml [27]. 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 [36]. 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 [17] 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 §3.5 we show how we use the CakeML toolchain to synthesize a CakeML program from the logical specification of our proof checker; in §3.7 this program is compiled to machine code.

3.3 High-level approach

There are several parts involved in our proof checker development; a framework for storing logical theories (§3.2.1), a verified theorem prover kernel (§3.2.2), and a verified compiler (§3.2.3). 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.

3.3.1 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.

3.3.2 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 §3.2.1, 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 §3.8.

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 (§3.8), we will describe the OpenTheory abstract machine (§3.4), how we synthesize code from the shallow embeddings (§3.5), extend our program with verified I/O capabilities (§3.6), and finally, compile it to machine code (§3.7).

3.4 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.

3.4.1 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 §3.10 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
|⟩
```

3.4.2 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 Figure 3.1 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).

3.4.3 Commands

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

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

Figure 3.1. The type of OpenTheory objects. Those commands executed by the OpenTheory machine that take inputs and/or produce results use the type object.

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 [16] on the theorems th_1 and th_2:

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

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 [20].

3.4.4 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 3.A.

3.5 Proof-producing synthesis of CakeML

At this stage we have a shallow-embedded implementation of the OpenTheory abstract machine in HOL4 (see §3.4), together with the functions that make up the Candle theorem prover kernel. We apply a proof-producing synthesis tool [17] 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 §3.8, as it eliminates the gap between the shallow- and deeply embedded views of the proof checker program (cf. §3.3). Using the certificate, we may turn any statement about the shallow embedding into a statement about the semantics of the deep embedding.

3.5.1 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:

$$\begin{array}{l} \vdash (\mathsf{A} \longrightarrow \mathsf{B})\ f\ g \\ \text{where the types are} \\ \quad f : \alpha \to \beta \\ \quad \mathsf{A} : \alpha \to \mathsf{v} \to \mathsf{bool} \quad \text{(specifies refinement of values of type } \alpha) \\ \quad \mathsf{B} : \beta \to \mathsf{v} \to \mathsf{bool} \quad \text{(specifies refinement of values of type } \beta) \end{array} \tag{3.1}$$

Certificate theorems in the style of the Theorem (3.1) 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.

3.5.2 Certificate theorem

Here is the certificate theorem for our shallow embedding readLine:

$$\begin{aligned}\vdash\ &\mathsf{ArrowP\ F\ (hol\_store}, p)\ (\mathsf{Pure\ (Eq\ string\_type}\ line\_v))\\ &(\mathsf{ArrowM\ F\ (hol\_store}, p)\ (\mathsf{EqSt\ (Pure\ (Eq\ reader\_state\_type}\ state\_v))\ state)\\ &(\mathsf{Monad\ reader\_state\_type\ hol\_exn\_type}))\ \mathsf{readLine\ readline\_v}\end{aligned} \tag{3.2}$$

The specifics of the symbols involved in this theorem are outside the scope of this paper; see e.g. [17]. In short, the Theorem (3.2) 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.

3.6 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 §3.5 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 §3.6.2. 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 §3.4) 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 3.B for the full listings.

3.6.1 Specification

Unlike previous stages of development (§3.5), 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 3.C for the definitions of read_file and read_stdin.

We define reader_main as follows:

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

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.

3.6.2 Verification using characteristic formulae

To show that `reader_main` adheres to its specification reader_main (see A.3 in §3.3) we prove a theorem using the characteristic formulæ (CF) framework for CakeML [14]. 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:

$$\begin{array}{l}\vdash (\exists\, s.\ \mathsf{init\_reader}\ ()\ \mathit{refs} = (\mathsf{Success}\ (), s)) \wedge \mathsf{input\_exists}\ \mathit{fs}\ \mathit{cl} \wedge \\ \quad \mathsf{unit\_type}\ ()\ \mathit{unit\_v} \Rightarrow \\ \quad \{|\mathsf{commandline}\ \mathit{cl} * \mathsf{stdio}\ \mathit{fs} * \mathsf{hol\_store}\ \mathit{refs}|\} \\ \quad\ \mathsf{reader\_main\_v} \cdot [\mathit{unit\_v}] \\ \quad \{|\mathsf{POSTv}\ \mathit{res}. \\ \quad\quad \langle \mathsf{unit\_type}\ ()\ \mathit{res}\rangle * \mathsf{stdio}\ (\mathsf{fst}\ (\mathsf{reader\_main}\ \mathit{fs}\ \mathit{refs}\ (\mathsf{tl}\ \mathit{cl})))|\}\end{array} \tag{3.3}$$

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 (3.3) are not important here; for an in-depth treatment, see [14].

Theorem (3.3) 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 (3.3) makes use of the certificate theorem from §3.5.2 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 (§3.4).

(ii) We have synthesized *deeply-embedded* CakeML from the shallow embedding, and obtained a certificate theorem (§3.5).

(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.

3.7 In-logic compilation

In this section we explain how the proof checker program from §3.6 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 [12]. The models used for reasoning about the machine code of these targets were specified using the L3 specification language [11], 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 §3.6. 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:

$$
\begin{array}{l}
\vdash \mathsf{input\_exists}\ fs\ cl \wedge \mathsf{wfcl}\ cl \wedge \mathsf{wfFS}\ fs \wedge \mathsf{STD\_streams}\ fs \Rightarrow \\
\quad (\mathsf{installed\_x64}\ \mathsf{reader\_code}\ (\mathsf{basis\_ffi}\ cl\ fs)\ mc\ ms \Rightarrow \\
\quad\ \mathsf{machine\_sem}\ mc\ (\mathsf{basis\_ffi}\ cl\ fs)\ ms \subseteq \\
\quad\ \mathsf{extend\_with\_resource\_limit}\ \{\ \mathsf{Terminate\ Success}\ (\mathsf{reader\_io\_events}\ cl\ fs)\ \}\ ) \wedge \\
\quad \mathsf{let}\ (fs\_out, hol\_refs, final\_state) = \mathsf{reader\_main}\ fs\ \mathsf{init\_refs}\ (\mathsf{tl}\ cl) \\
\quad \mathsf{in} \\
\quad\ \mathsf{extract\_fs}\ fs\ (\mathsf{reader\_io\_events}\ cl\ fs) = \mathsf{Some}\ fs\_out
\end{array}
\tag{3.4}
$$

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 (3.4) in the following paragraphs.

Assumptions on the environment Theorem (3.4) 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 (3.4) 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 (3.5) we have established a convincing connection between the logical specification of our proof checker (§3.4), 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.

3.8 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 (§3.7) 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 [23].

Here is the soundness result for the OpenTheory proof checker.

$$
\begin{array}{l}
\vdash \mathsf{input\_exists}\ fs\ cl \wedge \mathsf{wfcl}\ cl \wedge \mathsf{wfFS}\ fs \wedge \mathsf{STD\_streams}\ fs \Rightarrow \\
\quad (\mathsf{installed\_x64}\ \mathsf{reader\_code}\ (\mathsf{basis\_ffi}\ cl\ fs)\ mc\ ms \Rightarrow \\
\quad\ \mathsf{machine\_sem}\ mc\ (\mathsf{basis\_ffi}\ cl\ fs)\ ms \subseteq \\
\quad\ \mathsf{extend\_with\_resource\_limit} \\
\qquad\quad \{\ \mathsf{Terminate\ Success}\ (\mathsf{reader\_io\_events}\ cl\ fs)\ \}\ ) \wedge \\
\quad \exists fs\_out\ hol\_refs\ s. \\
\qquad \mathsf{extract\_fs}\ fs\ (\mathsf{reader\_io\_events}\ cl\ fs) = \mathsf{Some}\ fs\_out \wedge \\
\qquad (\mathsf{no\_errors}\ fs\ fs\_out \Rightarrow \\
\qquad\ \mathsf{reader\_main}\ fs\ \mathsf{init\_refs}\ (\mathsf{tl}\ cl) = (fs\_out, hol\_refs, \mathsf{Some}\ s) \wedge \\
\qquad\ hol\_refs.\mathsf{the\_context\ extends\ init\_ctxt} \wedge \\
\qquad\ fs\_out = \mathsf{add\_stdout}\ (\mathsf{flush\_stdin}\ (\mathsf{tl}\ cl)\ fs) \\
\qquad\quad (\mathsf{print\_theorems}\ s\ hol\_refs.\mathsf{the\_context}) \wedge \\
\qquad\ \forall asl\ c. \\
\qquad\quad \mathsf{mem}\ (\mathsf{Sequent}\ asl\ c)\ s.\mathsf{thms} \wedge \\
\qquad\quad \mathsf{is\_set\_theory}\ \mu \Rightarrow \\
\qquad\quad (\mathsf{thyof}\ hol\_refs.\mathsf{the\_context}, asl) \models c) \\
\\
\text{where}\ \ \mathsf{no\_errors}\ fs\ fs\_out = (fs.\mathsf{stderr} = fs\_out.\mathsf{stderr})
\end{array}
\tag{3.5}
$$

The first part of Theorem (3.5) is identical to the machine code correctness theorem (3.4) in §3.7. In short, it states that the machine code reader_code faithfully implements the shallow embedding reader_main; see §3.7 for details.

The interesting parts of Theorem (3.5) 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 §3.4) 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 (3.5); 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 §3.8.1, but refer readers to Kumar, et al. [23] 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 §3.4. We then compose this result with the machine code theorem from §3.7 in order to obtain the Theorem (3.5).

3.8.1 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 (3.5). 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 (3.5) in §3.8.

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, $\vdash$:

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

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. [23, 15] 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 [23]:

$$\mathsf{is\_set\_theory}\ \mu \Rightarrow \forall\, hyps\ c.\ hyps \vdash c \Rightarrow hyps \models c$$

We make use of this in §3.8.3 to lift a syntactic result about our proof checker into the semantic domain.

3.8.2 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 §3.4.

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 (§3.4.2);

(ii) defining a property for the OpenTheory machine state type state (§3.4.1), 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 (§3.4.4).

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:

```
READER_STATE ctxt state =
  every (THM ctxt) state.thms ∧
  every (OBJ ctxt) state.stack ∧
  ∀ n obj.
     lookup (Num n) state.dict = Some obj ⇒
     OBJ ctxt obj
```

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:

$$\begin{array}{l}\vdash \mathsf{STATE}\ ctxt\ refs \wedge \mathsf{READER\_STATE}\ ctxt\ st \wedge \\ \quad \mathsf{readLine}\ line\ st\ refs = (res,refs') \Rightarrow \\ \quad \exists\, upd. \\ \qquad \mathsf{STATE}\ (upd +\!\!+ ctxt)\ refs' \wedge \\ \qquad \forall\, st'.\ res = \mathsf{Success}\ st' \Rightarrow \mathsf{READER\_STATE}\ (upd +\!\!+ ctxt)\ st'\end{array} \tag{3.6}$$

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 (3.6) 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 (3.6), 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 §3.8.3 we compose this result with the Candle soundness result (§3.8.1), and show that soundness holds for our shallow embedded proof checker.

3.8.3 Soundness of the shallow embedding

With Theorem (3.6) in §3.8.2, 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 §3.8.1.

Our soundness theorem is stated in terms of the proof checker specification reader_main from §3.6.1:

$$\begin{array}{l}\vdash \mathsf{is\_set\_theory}\ \mu \wedge \\ \quad \mathsf{reader\_main}\ fs\ \mathsf{init\_refs}\ cl = (fs\_out,hol\_refs,\mathsf{Some}\ s) \Rightarrow \\ \quad (\forall\, asl\ c. \\ \qquad \mathsf{mem}\ (\mathsf{Sequent}\ asl\ c)\ s.\mathsf{thms} \Rightarrow \\ \qquad (\mathsf{thyof}\ hol\_refs.\mathsf{the\_context},asl) \models c) \wedge \\ \quad hol\_refs.\mathsf{the\_context}\ \mathsf{extends}\ \mathsf{init\_ctxt} \wedge \\ \quad fs\_out = \mathsf{add\_stdout}\ (\mathsf{flush\_stdin}\ cl\ fs)\ (\mathsf{print\_theorems}\ s\ hol\_refs.\mathsf{the\_context})\end{array} \tag{3.7}$$

With this theorem, we have all ingredients required to obtain the main correctness Theorem (3.5) from §3.8:

- Theorem (3.7) is stated in terms reader_main, and guarantees that the main proof checker program from §3.6 is sound.
- Theorem (3.4) shows that the machine code reader_code is a refinement of the program in §3.6.

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 (3.5).

3.9 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.

We have evaluated the performance of our proof checker program, and compared it to an existing (unverified) tool [21], built using three Standard ML compilers: MLton [1], Poly/ML [3] and Moscow ML [2]. 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 3.1.

Table 3.1. Comparison of average running times when running each tool 10 times on each input. Times are formatted as $(\mathrm{mean} \pm \sigma)$.

	`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

where OPC is our verified proof-checker binary
MLT is the OpenTheory tool compiled with MLton
PML ———— " ———— Poly/ML
MML ———— " ———— Moscow ML

When compared against the OpenTheory tool [21], 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 3.1). We expect that improvements to CakeML I/O facilities will improve the performance of our proof checker.

3.10 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. [23], and uses the CakeML toolchain [17, 36, 24] 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 (§3.4) 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 §3.8 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 [19] includes a tool called the OpenTheory tool [21], 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 (§3.9), 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 [4] 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 [34]) 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 [22].

Ridge and Margetson [34] 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 [31], 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 [10]) 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 [22]. 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 [10], 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.

3.11 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: `code.cakeml.org/tree/master/candle/standard/opentheory`

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.

3.A OpenTheory abstract machine

The definition of the shallow-embedded OpenTheory machine (§3.4.4).

```
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) ← pop s; tm ← getTerm obj;
   th ← ASSUME tm;
   return (push (Thm th) s)
  od
else if line = "axiom" then
 do
  (obj,s) ← pop s; tm ← getTerm obj;
  (obj,s) ← pop s; ls ← getList obj;
  ls ← map getTerm ls;
  th ← find_axiom (ls,tm);
  return (push (Thm th) s)
 od
else if line = "betaConv" then
 do
  (obj,s) ← pop s; tm ← getTerm obj;
  th ← BETA_CONV tm;
  return (push (Thm th) s)
 od
else if line = "cons" then
 do
  (obj,s) ← pop s; ls ← getList obj;
  (obj,s) ← pop s;
  return (push (List (obj::ls)) s)
 od
else if line = "const" then
 do
  (obj,s) ← pop s; n ← getName obj;
  return (push (Const n) s)
 od
else if line = "constTerm" then
 do
  (obj,s) ← pop s; ty ← getType obj;
  (obj,s) ← pop s; nm ← getConst obj;
  ty0 ← get_const_type nm;
                              ...
```

```
...
    tm ←
     case match_type ty0 ty of
       None ⇒ failwith "constTerm"
     | Some theta ⇒ mk_const (nm,theta);
    return (push (Term tm) s)
  od
else if line = "deductAntisym" then
  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
else if line = "def" then
  do
    (obj,s) ← pop s; n ← getNum obj;
    obj ← 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;
  (th1,th2) ← new_basic_type_definition nm abs rep th;
  (_,a) ← dest_eq (concl th1);
  th1 ← ABS a th1;
  th2 ← SYM th2;
  (_,Pr) ← dest_eq (concl th2);
  (_,r) ← dest_comb Pr;
  th2 ← ABS r th2;
  return (push (Thm th2) (push (Thm th1) (push (Const rep)
       (push (Const abs) (push (TypeOp nm) s)))))
 od
else if line = "eqMp" then
 do
  (obj,s) ← pop s; th2 ← getThm obj;
  (obj,s) ← pop s; th1 ← getThm obj;
  th ← EQ_MP th1 th2;
  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; th2 ← getThm obj;
  (obj,s) ← pop s; th1 ← getThm obj;
  th ← PROVE_HYP th2 th1;
  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)
 odelse if line = "trans" then
 do
  (obj,s) ← pop s; th2 ← getThm obj;
  (obj,s) ← pop s; th1 ← getThm obj;
  th ← TRANS th1 th2;
  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: " ^ line)
    | "\"" ⇒ failwith ("unrecognised input: " ^ line)
    | #"""::c::cs ⇒
      return
        (push (Name (implode (front (c::cs)))) s)
    | _ ⇒ failwith ("unrecognised input: " ^ line)
```

3.B Listings of CakeML code

The listing for `read_stdin` (§3.6).

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

The listing for `read_file` (§3.6).

```
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 (§3.5) on a proof command (§3.4.3).

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

3.C Specifications for CakeML code

The definition of readLines, which calls on readLine (§3.4.4, and Appendix A) to process a list of proof commands (§3.4.3).

```
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 (§3.6.1).

```
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 (§3.6.1).

```
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)
```

Bibliography

[1] The MLton compiler. Accessed: 26-Oct-2019.

[2] The Moscow ML compiler. Accessed: 26-Oct-2019.

[3] The Poly/ML compiler. Accessed: 26-Oct-2019.

[4] Mark Adams. Introducing HOL Zero - (extended abstract). In *ICMS*, pages 142–143, 2010.

[5] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. CertiCoq: A verified compiler for Coq. In *CoqPL*, 2017.

[6] Abhishek Anand, Simon Boulier, Nicolas Tabareau, and Matthieu Sozeau. Typed Template Coq – Certified Meta-Programming in Coq. In *CoqPL*, 2018.

[7] Rob Arthan. The ProofPower web pages, 2017. Accessed: 22-Feb-2019.

[8] Sandrine Blazy, Benoît Robillard, and Andrew W. Appel. Formal verification of coalescing graph-coloring register allocation. In *ESOP*, 2010.

[9] Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. Imperative functional programming with Isabelle/HOL. In *TPHOLs*, pages 134–149, 2008.

[10] Jared Davis and Magnus O. Myreen. The reflective Milawa theorem prover is sound (down to the machine code that runs it). *JAR*, 55(2):117–183, 2015.

[11] Anthony C. J. Fox. Directions in ISA specification. In *ITP*, pages 338–344. Springer, 2012.

[12] Anthony C. J. Fox, Magnus O. Myreen, Yong Kiam Tan, and Ramana Kumar. Verified compilation of CakeML to multiple machine-code targets. In *CPP*, pages 125–137. ACM, 2017.

[13] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Springer, 1979.

[14] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. Verified characteristic formulae for CakeML. In *ESOP*, pages 584–610, 2017.

[15] John Harrison. Towards self-verification of HOL Light. In *IJCAR*, pages 177–191, 2006.

[16] John Harrison. HOL Light: An overview. In *TPHOLs*, pages 60–66, 2009.

[17] Son Ho, Oskar Abrahamsson, Ramana Kumar, Magnus O. Myreen, Yong Kiam Tan, and Michael Norrish. Proof-producing synthesis of CakeML with I/O and local state from monadic HOL functions. In *IJCAR*, pages 646–662, 2018.

[18] Lars Hupel and Tobias Nipkow. A verified compiler from Isabelle/HOL to CakeML. In *ESOP*. Springer, 2018.

[19] Joe Hurd. The OpenTheory standard theory library. In *NFM*, pages 177–191, 2011.

[20] Joe Hurd. The OpenTheory article file format, 2014. Accessed: 22-Feb-2019.

[21] Joe Hurd. The OpenTheory tool, 2018. Accessed: 26-Feb-2019.

[22] Matt Kaufmann and J. Strother Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Trans. Software Eng.*, 23(4):203–213, 1997.

[23] Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. Self-formalisation of higher-order logic - semantics, soundness, and a verified implementation. *JAR*, 56(3):221–259, 2016.

[24] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In *POPL*, pages 179–192, 2014.

[25] Peter Lammich. Refinement to Imperative/HOL. In *ITP*, 2015.

[26] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *PLDI*, pages 24–35, 1994.

[27] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system documentation and user's manual, 2018. Accessed: 25-Feb-2019.

[28] Robin Milner, Mads Tofte, and Robert Harper. *Definition of Standard ML*. MIT Press, 1997.

[29] Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. Œuf: minimizing the Coq extraction TCB. In *CPP*, 2018.

[30] Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *JFP*, 24(2-3):284–315, 2014.

[31] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[32] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional big-step semantics. In *ESOP*, pages 589–615, 2016.

[33] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[34] Tom Ridge and James Margetson. A mechanically verified, sound and complete theorem prover for first order logic. In *TPHOLs*, pages 294–309, 2005.

[35] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *TPHOLs*, pages 28–32, 2008.

[36] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *JFP*, 29, 2019.

[37] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, Tutorial Text*, LNCS. Springer, 1995.