CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Thesis for the Degree of Doctor of Philosophy

A Verified Theorem Prover for Higher-Order Logic

Oskar Abrahamsson

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

Gothenburg, Sweden
2022

A Verified Theorem Prover for Higher-Order Logic
Oskar Abrahamsson
ISBN 978-91-7905-735-0

Doktorsavhandlingar vid Chalmers Tekniska Högskola
Ny serie nr. 5201
ISSN 0346-718X

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, 2022

Abstract

This thesis is about mechanically establishing the correctness of computer programs. In particular, we are interested in establishing the correctness of tools used in computer-aided mathematics. We build on tools for proof-producing program synthesis, and verified compilation, and a verified theorem proving kernel. With these, we have produced an interactive theorem prover for higher-order logic, called Candle, that is verified to accept only true theorems. To the best of our knowledge, Candle is the only interactive theorem prover for higher-order logic that has been verified to this degree.

Candle and all technology that underpins it is developed using the HOL4 theorem prover. We use proof-producing synthesis and the verified CakeML compiler to obtain a machine code executable for the Candle theorem prover. Because the CakeML compiler is verified to preserve program semantics, we are able to obtain a soundness result about the machine code which implements the Candle theorem prover.

Contents

Chapter 1

Introduction

This thesis is concerned with establishing the correctness of computer programs using rigorous mathematical proof. In particular, we concern ourselves with a class of programs called *interactive theorem provers*, which are tools used for computer-assisted mathematics. Our main contribution is a system for interactive theorem proving which has been mechanically verified to be correct using another interactive theorem prover. By correctness, we mean that the theorem prover is unable to prove any falsehood.

1.1 Motivation

The primary focus of this thesis is the correctness of computer programs. We use mathematical methods when reasoning about program correctness, because we wish to have the highest confidence possible in our conclusions. When we use mathematics to establish the correctness of a computer program, we say that we subject it to *formal verification*.

Formal verification of software at a high level of detail requires complex models about programming languages and computers, and gives rise to large and detailed mathematical proofs. Maintaining these models, and producing and checking such detailed proofs, is generally beyond what a human can manage with only pen and paper. To this end, we use tools called interactive theorem provers, which are designed to assist humans in precisely this task.

Interactive theorem provers help us manage the complexity involved in formal verification by managing our mathematical definitions, theorems and proofs; by checking all proof steps for correctness; and by automating the tedious parts of our work, so that we may focus on its high-level aspects.

With mechanized proof checking, we no longer have to concern ourselves with the correctness of our proofs: instead, what matters is the correctness of the tool that checks the proof. It is therefore imperative that the interactive theorem prover we use works as advertised: it must be impossible to use the system to prove falsehoods.

Of course, one way to establish the correctness of an interactive theorem prover is to subject it to formal verification! This is the main contribution of this thesis: a new interactive theorem prover for higher-order logic, called Candle, which has been formally verified to be correct. Candle's correctness result states that anything that Candle claims to be a theorem is actually true.

1.2 Concepts

In this section, we introduce the necessary background for this thesis. We explain the most important concepts that underpin this work, and what their relevance to the work is.

Formal logic. Formal logics are mathematical languages that enable us to model and reason about concepts very precisely. They also enable us to construct and check proofs in a mechanical way. A formal logic consists of a syntax, i.e., symbols that make up logical statements, and proof rules that govern how symbols and statements may be combined to form new statements, and a well-defined meaning of the syntax, called a semantics.

For a logic to be useful, it should only be possible to use its proof rules to derive syntax that is true according to its semantics. A calculus with this property is said to be *sound* with respect to the semantics. Reasoning using a sound calculus is guaranteed to result in valid proofs.

Higher-order logic (HOL). Higher-order logic is an expressive formal logic, based on Church's type theory. Its expressivity allows it to both describe the syntax and semantics of programming languages, and to be used as a programming language itself. Writing programs in the same language as we state facts about those programs makes the task of verification simpler, because it minimizes the number of layers of abstraction present between specification and implementation.

Interactive theorem provers (ITPs). Interactive theorem provers are computer programs designed to assist humans with reasoning in a formal logic. They are called *interactive* because human interaction is required to guide the system when carrying out proof. Even so, modern ITPs employ a significant degree of automation to reduce the amount of tedious work a human has to carry out when constructing a formal proof.

All terms, definitions, and proof commands that a user enters into the ITP system are checked for wellformedness, meaning that the user can trust any theorem produced by the system, as long as they trust the correctness of the system itself.

The LCF approach. The LCF approach is a method of designing theorem provers in a way that enables extensibility without compromising soundness. In the LCF approach, 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 rules of the proof calculus) of the logic. The LCF approach was pioneered by Milner and his collaborators as part of the Edinburgh LCF system [27] over forty years ago, but most modern ITPs are still designed in the LCF style.

The HOL4 theorem prover. The HOL4 theorem prover [57] is an interactive theorem prover for higher-order logic. Like most other HOL theorem provers, it follows the LCF tradition; its ancestry traces back to Mike Gordon's original HOL system [26], which in turn evolved from Edinburgh LCF [27].

HOL4 is host to several state-of-the-art code generation techniques and tools that we develop and make use of in this work [50, 61]. This includes, e.g., the CakeML programming language and its compiler, as well as a verified implementation of a HOL logical kernel, called the Candle kernel. Both CakeML and the Candle kernel are important parts that underpin the work presented in this thesis, and are discussed below.

HOL Light. HOL Light [34] is a minimalistic HOL theorem prover. It shares a common ancestry with HOL4, but its implementation is more lightweight, and its trusted kernel is simpler. However, HOL Light comes with powerful automation, and an extensive library of formalized mathematics. The system played a key role in the formalization of Hales' proof of the Kepler conjecture [32]. Our new theorem prover, Candle, is a verified implementation of HOL Light.

CakeML and proof-producing code generation. CakeML is a functional programming language that comes with a verified compiler [61], and a proof-producing code generation mechanism for the HOL4 system [50]. Using the CakeML tools, it is possible to synthesize executable programs from functions in the HOL4 logic (i.e., HOL). The CakeML compiler comes with an interactive read-eval-print loop, and has been proven to only produce executables that behave as the input programs they were created from.

The compiler and its tools have been used to produce various verified programs, for example: a SAT proof checker [60]; a checker for floating-point error bounds [11]; a checker for vote counting [25]; a monitor for cyber-physical systems [13]; and a verified implementation of the HOL Light kernel, called the Candle kernel, discussed below.

The kernel of the Candle theorem prover. The Candle theorem prover kernel [41] is a verified implementation of the HOL Light kernel. 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. [41], and the CakeML tools can be used to produce an executable version of the kernel. The Candle kernel is the logical kernel used by the Candle theorem prover.

The OpenTheory framework. The main contribution of the OpenTheory framework [37] is the OpenTheory article format [38]: a format for transferring logical theories between HOL theorem provers. OpenTheory articles provide a means to record and store proofs of HOL theorems; several HOL theorem provers can store logical definitions and proofs in this format. The OpenTheory

framework includes an unverified proof checking tool [39]. We use OpenTheory articles as the proof format for a new verified HOL proof checker, which is described below.

A verified OpenTheory proof checker. One of the contributions of this thesis is a new, verified proof checker for OpenTheory articles. 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.

The Candle interactive theorem prover. The main contribution of this thesis is a new, fully verified interactive theorem prover for HOL, called Candle, which aims to be a faithful clone of HOL Light. Candle's development touches on most of the concepts explained so far: it builds on the CakeML compiler's interactive read-eval-print loop and the previously verified Candle kernel. Candle's top-level correctness result states that all its theorems are valid statements w.r.t. a formal semantics of higher-order logic.

1.3 Contributions

The main contribution of this thesis is the construction and soundness proof of a new interactive theorem prover, called Candle:

- Candle has been verified sound with respect to a formal semantics of HOL. Its main soundness result is a strong end-to-end theorem that guarantees the soundness of the machine code artifact that executes the Candle theorem prover. To the best of our knowledge, Candle is the only interactive theorem prover for HOL that has been subjected to this degree of verification.

The journey towards this goal has brought along with it several contributions:

- We extended existing techniques for proof-producing program synthesis from functional programs written in higher-order logic (Chapter 2). The new techniques can synthesize programs that manipulate mutable state and perform I/O from HOL functions modeling these effects using monads. These techniques are used in all other contributions in this thesis.

- We developed a new proof-checker for higher-order logic, and verify its soundness (Chapter 3). The proof checker was developed in the HOL4 logic, and can be compiled to machine code by the verified CakeML

compiler. Using the CakeML compiler allows us to obtain a soundness result for the machine code which executes the proof checker.

- In our work on Candle, we showed that the soundness of an entire LCF-style theorem prover system can be proved from the soundness of its kernel (Chapter 4). Traditional soundness arguments for LCF-style ITPs rest on the type system of the ML compiler which hosts the system. We chose a different approach for Candle, and distilled what is sufficient to ensure prover soundness into a few syntactic and semantic requirements, and augmented the actual Candle implementation to satisfy these requirements by construction.
- We added a new, verified rule for computation to the Candle theorem prover kernel (Chapter 5). Contemporary ITPs either implement slow and safe rules of computation, or achieve speed by relying on insecure methods for computing normal forms outside their trusted parts. We show that one can have both speed and safety with a verified interactive theorem prover.

1.4 Summary of included papers

This thesis consists of four papers, listed below. Each paper is given a short summary in the following subsections.

I Oskar Abrahamsson, Son Ho, Hrutvik Kanabar, Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Yong Kiam Tan. Proof-Producing Synthesis of CakeML from Monadic HOL Functions. *J. Autom. Reason.*, 64(7):1287–1306, Springer, 2020.

II Oskar Abrahamsson. A Verified Proof Checker for Higher-Order Logic. *J. Log. Algebraic Methods Program.*, 112:100530, Elsevier, 2020.

III Oskar Abrahamsson, Magnus O. Myreen, Ramana Kumar, and Thomas Sewell. Candle: A Verified Implementation of HOL Light. In *Interactive Theorem Proving (ITP)*, volume 237 of *LIPIcs*, 3:1–3:17, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

IV Oskar Abrahamsson and Magnus O. Myreen. Fast, Verified Computation for Candle. *Technical report*, 2022.

All four papers appear in this document unedited, with the exception of adjustments to typesetting.

1.4.1 Proof-Producing Synthesis of CakeML from Monadic HOL Functions

Paper I introduces a tool which makes it possible to perform programming in HOL, using state and effects, such as I/O, and exceptions. For the uninitiated,

one can understand this as: using the HOL4 logic as a programming language, and automatically translating those programs to equivalent CakeML code. The technical contributions in this paper extends previous work on synthesis of non-effectful CakeML programs [50]. See Chapter 2 in this thesis for Paper I.

We say that the tool is *proof-producing* because each at execution, the tool derives a proof that relates the input logical functions with the synthesized program output. The proof certifies that the synthesized program will compute the same values during execution, and modify the state in the same way, as the input logical functions. As a consequence, any verification result about the logical functions can be made into a result about the synthesized program 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 allows us to model side-effects and I/O inside the logic using monads [64], thereby granting us greater expressivity when using HOL as a programming language. These features were crucial to the development of the work in Papers II, III, and IV, which are described below.

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

1.4.2 A Verified Proof Checker for Higher-Order Logic

Paper II introduces a mechanized proof checker for proofs of theorems in HOL. The checker 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 [38] as input and uses the Candle kernel [41] to check proof steps, and outputs a verdict stating whether the proof is 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, as its logical kernel, the Candle theorem prover kernel [41], which is verified to be sound.

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. The resulting CakeML code is compiled to executable machine code in a proof-producing way, by executing the CakeML compiler [61] inside the HOL4 logic. This approach transports the soundness result of the checker further, to level of the machine code that executes the proof checker.

Statement of contribution. I am the sole author of Paper II. 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.

1.4.3 Candle: A Verified Implementation of Higher-Order Logic

Paper III contributes a new, verified interactive theorem prover for HOL, called Candle. The theorem prover is built on the previously verified Candle theorem prover kernel [41], from which it also takes its name. Candle makes use of the CakeML compiler's new dynamic compilation primitive, called Eval[1], to provide interactivity and extensibility. This is achieved by allowing the user to enter arbitrary program text at runtime, which is compiled and executed dynamically.

We have proved a strong soundness result for Candle: anything that the system claims to be a theorem is a valid statement according to a formal semantics of HOL, *regardless of what program text the user enters into the system at runtime.* Candle's soundness theorem is stated in terms of the machine code which executes the theorem prover; and it is, to our knowledge, the only such result for a HOL theorem prover.

An important part of the work in Paper III was to make Candle as faithful to HOL Light as possible. To this end, the CakeML compiler was extended with a new parser frontend for the subset of OCaml that is used by the HOL Light base libraries and theories. As a consequence, Candle can execute a substantial part of HOL Light's code and theories.

Statement of contribution. I set up the safety invariants for program values that guarantees the veracity of theorem values at runtime, and proved the main simulation theorem which states that these invariants are preserved by the source language semantics. I proved most of the top-level soundness theorem for the Candle ITP, and contributed to the writing of this paper.

1.4.4 Fast, Verified Computation for Candle

Paper IV is about extending the Candle kernel with a verified rule for computation. The rule has been shown to be sound with respect to Candle's HOL semantics, and has been fully integrated into the Candle theorem prover and its soundness proofs.

Most modern HOL ITPs come with functionality for computing normal forms of terms, but the implementations of these functions are either excessively slow, or rely on unverified computation outside of the theorem provers logic.

Safe computation functions in modern HOL ITPs are slow because the LCF design of these systems require that any computations decompose to

[1]This work is not yet published.

applications of primitive inference rules. Unfortunately, these rules usually perform tedious type checking, and checks for α-conversion, and expensive substitutions.

Some ITPs forego their trusted kernels altogether, and implement fast computation rules using interpreters, and unverified translations between logical terms, and the internal language representation of the interpreter.

In Paper IV, we extend Candle's verified kernel with a computation function, and update its verification proofs. In doing so, we show that it is possible to extend the trusted kernel of a HOL theorem prover with a fast interpreter for computation without compromising soundness. The end-to-end soundness proof of Candle has been updated to cover this addition to the kernel.

Paper IV is a technical report where the development is supported by completed formal proofs.

Statement of contribution. I implemented the first three versions of the interpreter that is used in computing normal forms of terms, and proved the correctness of the interpreters, and proved the soundness of the computation rules built around the interpreters. I integrated the computation rules and their correctness results with the existing Candle ITP development, and updated Candle's soundness result. I contributed to the writing of this paper.

The rest of this thesis consists of four chapters, each corresponding to one of the papers listed above.

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 [42], 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 [50] 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 [50], 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 [50]. 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 [41] (global refs, exceptions)
- an OpenTheory [37] 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 [35]. 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 [50],

(ii) separation logic [55] as used by characteristic formulae for CakeML [29],

(iii) a new abstract synthesis mode for the CakeML synthesis tools [50].

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 [50] has been generalised. In particular, we explain how the separation logic from our previous work on characteristic formulae [29] 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 [53], 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.

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

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 [50] 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 [50].

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

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 [50]. 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 [29], 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.

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

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

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).

```
example_state =
  ⟪ ref1 : int;  farray1 : int list;  rarray1 : int list;  stdio : IO_fs ⟫
```

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 [50] 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 [50] 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 [12] 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 [35] 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 [37] 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 [41]. 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 [44]. 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. [14] 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 [43] 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 [14, 43] 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, 36, 48, 50]. 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 [42]. 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 [50] (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. [48] 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 [48].

Hupel and Nipkow [36] 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 [37]. 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) [41] 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 [57]. 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 [35] 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 [61]. 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 [37]. 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 [41] 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. $\Rightarrow$, $\wedge$, $\vee$, $\neg$, ...) 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 [37] 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 [57], HOL Light [34], ProofPower-HOL [8]. 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. [41]. The kernel is implemented as a collection of monadic functions [64] 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 [33].

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 [47] and OCaml [45]. 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 [61]. 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 [35] 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 [34] 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 [38].

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 [35] 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:

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

The specifics of the symbols involved in this theorem are outside the scope of this paper; see e.g. [35]. 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:

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

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 [29]. 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 [29].

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 [22]. The models used for reasoning about the machine code of these targets were specified using the L3 specification language [21], 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 [41].

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. [41] 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. [41, 33] 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 [41]:

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

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:

$$
\begin{array}{l}
\mathsf{OBJ}\ ctxt\ obj = \\
\quad \mathsf{case}\ obj\ \mathsf{of} \\
\quad\ \ \mathsf{List}\ xs\ \Rightarrow\ \mathsf{every}\ (\mathsf{OBJ}\ ctxt)\ xs \\
\quad \mid \mathsf{Type}\ ty\ \Rightarrow\ \mathsf{TYPE}\ ctxt\ ty \\
\quad \mid \mathsf{Term}\ tm\ \Rightarrow\ \mathsf{TERM}\ ctxt\ tm \\
\quad \mid \mathsf{Thm}\ thm\ \Rightarrow\ \mathsf{THM}\ ctxt\ thm \\
\quad \mid \mathsf{Var}\ (n,ty)\ \Rightarrow\ \mathsf{TERM}\ ctxt\ (\mathsf{Var}\ n\ ty) \wedge \mathsf{TYPE}\ ctxt\ ty \\
\quad \mid \_\ \Rightarrow\ \mathsf{T}
\end{array}
$$

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

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

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

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

Preservation theorem Finally, we take care of Step (iii). We prove the following preservation theorem, which guarantees that THM holds for all sequents

in the program state, at all times during execution:

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

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 [39], built using three Standard ML compilers: MLton [20], Poly/ML [46] and Moscow ML [1]. 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 [39], 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. [41], and uses the CakeML toolchain [35, 61, 42] 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 [37] includes a tool called the OpenTheory tool [39], 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 [3] 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 [56]) 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 [40].

Ridge and Margetson [56] 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 [51], 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 [18]) 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 [40]. 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 [18], 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)
```

Chapter 4

Candle: A Verified Implementation of HOL Light

Oskar Abrahamsson, Magnus O. Myreen, Ramana Kumar, and Thomas Sewell

Abstract. This paper presents a fully verified interactive theorem prover for higher-order logic, more specifically: a fully verified clone of HOL Light. Our verification proof of this new system results in an end-to-end correctness theorem that guarantees the soundness of the entire system down to the machine code that executes at runtime. Our theorem states that every exported fact produced by this machine-code program is valid in higher-order logic. Our implementation consists of a read-eval-print loop (REPL) that executes the CakeML compiler internally. Throughout this work, we have strived to make the REPL of the new system provide a user experience as close to HOL Light's as possible. To this end, we have, e.g., made the new system parse the same variant of OCaml syntax as HOL Light. All of the work described in this paper has been carried out in the HOL4 theorem prover.

Published in *Interactive Theorem Proving*, 2022.

4.1 Introduction

Interactive theorem provers (ITPs) for higher-order logic, such as HOL4, HOL Light, Isabelle/HOL and ProofPower, are designed to be as sound as possible. Their implementations follow an LCF-style architecture, which means that each prover has a small kernel that implements the inference rules of the hosted logic (higher-order logic) and the rest of the system is set up in such a way that all soundness-critical inferences must be performed by the functions inside the small kernel. The beauty of this approach is that there is not much soundness-critical source code, which means that this code can quite easily be manually inspected (or even verified). As a result, soundness bugs in these ITPs are very rare.

In search of ever stronger assurance guarantees, one might ask: is it really the case that only the code of the kernel needs to be trusted in order to trust the soundness of an entire ITP implementation? At the level of source code, the answer is yes. However, source code is not what runs on real machines. As a result, one should also take into consideration the implementation of the programming language that hosts the ITP. All of the ITPs mentioned above rely on complex implementations of functional programming languages, such as Poly/ML and OCaml, and they rely on their interactive implementations, where users can (and do) provide new program text while the ITP is running. The implementations of these hosting functional languages are far more complex than the kernels of these ITPs.

In this paper we address the question: is it possible to develop an ITP for higher-order logic (HOL) for which soundness can be proved down to the machine code that runs it? In prior work, soundness has been proved for kernels of ITPs, but not for entire HOL ITPs. Our question requires us to consider a proof of soundness for the prover including the *at runtime* user-provided source code, and beyond that, for the *interactive* implementation of the underlying functional programming language. Our answer is: yes, it is possible to develop such an end-to-end verified ITP for HOL, as we explain in this paper.

Contributions This paper's contribution is a new ITP called Candle[1], which consists of a clone of HOL Light running on top of a proved-to-be-safe CakeML-based read-eval-print loop (REPL).

- Our verification efforts result in a machine-code program, the Candle prover, for which we have proved that any theorem statement that it outputs follows by the inference rules of HOL (and, since these rules are sound, is valid by the semantics of HOL). To the best of our knowledge, this is the most comprehensive soundness result proved for a HOL ITP.
- This development can be seen as a major case study of the CakeML project, since it touches on almost every aspect of the CakeML project.

[1] The name Candle comes from the combination of CakeML and HOL Light.

This work is the first use of CakeML's new source primitive called Eval, which allows compilation and execution of user-provided program text at runtime.

- The resulting Candle ITP is designed to provide a user experience as close to HOL Light's as possible. For this purpose, we have made the new system parse the same variant of OCaml syntax as HOL Light. Our aim is to make it as straightforward as possible to port HOL Light developments to Candle.

The work described in this paper has been carried out in the HOL4 theorem prover [57]. Our proofs and binaries of Candle are available at: `https://cakeml.org/candle`.

4.2 Approach

This section provides a high-level outline of our work on Candle. Subsequent sections provide more details.

Prior work that we build on The results described in this paper build on substantial prior work. In particular, we build on our prior work on construction of a proved-to-be-sound implementation of a HOL Light-like kernel [41]. In that work, we proved that CakeML implementations of HOL Light's kernel functions are sound w.r.t. a formalisation of higher-order logic. Our new work on Candle also depends on many parts of the CakeML ecosystem: in particular, our proved-to-be-safe REPL relies on the CakeML compiler's ability to compile itself (bootstrapping).

Overview of new work The new work in this paper can be divided into the following three high-level steps:

1. We prove at the source level that any reasonable program that contains the Candle kernel as a prefix can only output facts that follow from the inference rules of HOL (Sec. 4.3);
2. We use CakeML's new Eval primitive to construct a proved-to-be-safe read-eval-print loop (REPL) that is sufficient for HOL Light-like interaction (Sec. 4.4);
3. Using CakeML's compiler correctness theorem, we transport the source-level soundness results down to the machine code that is the real implementation (Sec. 4.5).

The work for Step 1 centres around a whole-program simulation proof which establishes that only acceptable values, v_ok, are present in the system. Here a value v is considered v_ok if all the soundness-critical values, i.e., the

values representing HOL types, terms and theorems, within v are valid in the current logical context maintained by the Candle kernel. These soundness-critical values flow around unmodified outside of the kernel. The interesting case is when one of the kernel's functions is called. At these calls, we make use of our prior work on the soundness of the kernel functions. Throughout these proofs, a layer of complexity is added by the fact that CakeML's operational semantics is (almost completely) untyped.

Even though the proofs for Step 1 are mostly about maintaining v_ok throughout execution, the final soundness theorem proved in Step 1 is not about values. Instead, it is about what can be seen on the externally facing foreign-function interface (FFI). This is because our compiler correctness theorem talks about events on the FFI channels. As a result, the whole-program soundness theorem states that every output on a special theorem-printing FFI channel will only ever contain valid theorem statements.

In Step 2, the challenge was to build a REPL that allows the kind of interactivity that an ITP requires. Here we make use of an evaluate primitive, Eval, that has recently been added to the CakeML source language. This Eval primitive evaluates, at runtime, arbitrary user-provided code, which is exactly what one needs to build a program that implements a REPL. (We hope that our REPL is sufficiently similar to HOL Light's to be usable.) A key insight in this part of the work is that a full functional specification for the REPL is not required. For the purposes of our soundness theorem, it suffices to prove that the REPL is safe, i.e., it never gets the (untyped) operational semantics stuck.

Step 3 is an important step, even though it only takes a few lines of proof to complete. This step is a straightforward application of the compiler correctness theorem to: a theorem describing an in-logic evaluation of the compiler; the Candle soundness theorem proved in Step 1; and the safety theorem for the REPL from Step 2.

4.3 Proving source-level soundness of the Candle prover

This section explains our work for Step 1, i.e., how we prove, at the CakeML source level, that any reasonable program built from the Candle kernel is sound.

4.3.1 Idea: soundness-critical values only produced by kernel functions

The idea of LCF-style ITPs is that the soundness of the kernel functions together with the programming language-based protection of the soundness-critical datatypes (such as the datatypes representing HOL types, terms and theorems) imply that no malformed or false types, terms or theorems can be constructed in the ITP, no matter what user-provided code is executed at runtime.

The Candle ITP follows the LCF tradition. Our task is thus to formally show, in HOL4, that this design makes the Candle ITP sound. More specifically, in our case, the task is to prove that any reasonable program that contains the Candle kernel can only produce well-formed and sound types, terms and theorems of HOL.

4.3.2 Setting: CakeML's untyped operational semantics

In an LCF-style ITP, protection of soundness-critical datatypes is usually achieved by the type system of the implementation language. In ML languages, the usual route is to make the type, term, and theorem datatypes into abstract types using the module system. We take a hybrid approach, where the type system provides some of the protection, while the rest comes from syntactic safety-checks imposed by the REPL at runtime.

A source of complication arises, in our proofs, from the fact that CakeML's operational semantics is almost entirely untyped. CakeML's operational semantics is written in a functional big-step style that takes a CakeML program as input and either succeeds, returning a value or a raising exception; or gets stuck with a runtime type error. CakeML values include literals, vectors, type constructors, and function values. A function value contains code and a semantic environment, but very little type information. The CakeML operational semantics lacks information such as the type of function values.

The soundness of our type system and its inferencer implementation [62] allows us to limit ourselves to considering only programs with a non-erroneous semantics in our theorems. However, this does not rule out ill-typed programs from our proofs, as non-erroneous programs can still have ill-typed parts, as long as those parts are never executed.

4.3.3 Target: a theorem about externally observable events

The top-level correctness statement needs to be in terms of externally visible I/O events on the CakeML compiler's foreign-function interface (FFI). This goes against the natural way of thinking of soundness in terms of what values can and cannot be constructed during the execution of a program.

This theorem should state that whenever a value of the HOL theorem type is rendered as text and output on an FFI channel, then that value is indeed a true theorem of HOL. To achieve this, we need to separate the output of theorems rendered as text from any other output of the REPL, because the REPL can (and does) print all sorts of text during runtime; indeed, a user may instruct it to print any string of text that looks like a theorem, but isn't. Worse, the HOL Light pretty-printer is user-customisable and installed at runtime; we have no way of statically reasoning about this function in our proofs.

We put our soundness story on rock solid foundations by printing theorems on a special kernel-controlled FFI channel using a printer function which sits within the kernel. The output from this printer function is difficult to read,

$$\frac{f \in \mathsf{kernel\_funs}}{\mathsf{inferred}\ ctxt\ f} \qquad \frac{\mathsf{TYPE}\ ctxt\ ty \quad \mathsf{TYPE\_TYPE}\ ty\ v}{\mathsf{inferred}\ ctxt\ v}$$

$$\frac{\mathsf{TERM}\ ctxt\ tm \quad \mathsf{TERM\_TYPE}\ tm\ v}{\mathsf{inferred}\ ctxt\ v} \qquad \frac{\mathsf{THM}\ ctxt\ th \quad \mathsf{THM\_TYPE}\ th\ v}{\mathsf{inferred}\ ctxt\ v}$$

Figure 4.1. The defining rules of the inferred predicate. TYPE, TERM and THM are predicates from the Candle soundness development, stating that a value is a well-formed type, term, or theorem, with respect to a logical context. TYPE_TYPE, TERM_TYPE and THM_TYPE are relations invented by the CakeML code synthesis tool [50] as it processes these types, stating that the deep-embedded CakeML values v are refinements of the shallow-embedded HOL values: ty, tm, th.

but it is unambiguous and invertible, meaning that a theorem and its logical context can be recovered from the text.

4.3.4 Proof: values stay wellformed

The Candle kernel uses datatypes to represent soundness-critical HOL values: types, terms and theorems (sequents). It also defines functions that consume and produce values of these datatypes. Syntactic safety checks imposed by the REPL prevent values from being created using the HOL type constructors. Thus, the only way to create new values of these datatypes at runtime is by using the kernel functions.

We wish to establish the soundness of this design formally: that any reasonable program executed from a state which contains only well-formed HOL values should arrive in a state which contains only well-formed values. Values and functions defined inside the kernel are well-formed. So are types containing only defined type operators, well-typed terms containing only known constants, and theorems for which there is a derivation in the HOL proof calculus.

We say that a value is *safe*, written v_ok, if it contains only well-formed HOL values, written inferred v; or, if it is not a HOL specific value, all of its sub-values are v_ok. Type constructors for HOL values are not v_ok (or they would satisfy inferred), nor are references maintained by the kernel, as they could be used to modify the kernel state. Figure 4.1 shows the definition of inferred, and some of the rules defining v_ok are shown in Figure 4.2.

We say that code is safe, written safe_dec, if it does not directly mention the kernel FFI channel, nor call the constructors for the HOL datatypes. Safe code is still allowed to pattern match on HOL constructors and call the kernel functions.

We lift the v_ok predicate to environments and semantics states. An environment is env_ok if its values are v_ok, and if it maps the HOL constructor

$$\frac{\text{inferred } ctxt\ v}{\text{kernel\_vals } ctxt\ v} \qquad \frac{\text{kernel\_vals } ctxt\ v}{\text{v\_ok } ctxt\ v} \qquad \frac{\text{every } (\text{v\_ok } ctxt)\ vs}{\text{v\_ok } ctxt\ (\text{Vectorv } vs)}$$

$$\frac{\text{kernel\_vals } ctxt\ f \quad \text{v\_ok } ctxt\ v \quad \text{do\_partial\_app } f\ v = \text{Some } g}{\text{kernel\_vals } ctxt\ g}$$

$$\frac{\begin{array}{c}\text{every } (\text{v\_ok } ctxt)\ vs \\ \forall\, tag\ x.\ opt = \text{Some } (\text{TypeStamp } tag\ x) \Rightarrow x \notin \text{kernel\_types}\end{array}}{\text{v\_ok } ctxt\ (\text{Conv } opt\ vs)}$$

Figure 4.2. A few of the defining rules of the v_ok predicate. Here do_partial_app constructs a partial application, but fails if the function is fully applied. Conv is a constructor value, and Vectorv is a vector value, illustrating the recursive definition of v_ok.

names to the correct HOL types. The state predicate, state_ok, maintains that all non-kernel references contain v_ok values, and ensures that all kernel-owned references point to values that are refinements of the references in the state of the shallow-embedded kernel. It also guarantees that all events on the kernel FFI channel come from well-formed theorems, written ok_event (defined in Section 4.3.6), and sets up the Eval mechanism (described in Section 4.4.1) in such a way that it rejects code that is not safe_dec. Furthermore, state_ok asserts that the state has come far enough in its type numbering to not reuse a type number belonging to the kernel types.

We can now state and explain the proof of the following simulation result.

Theorem 1. Any execution of safe code (safe_dec), starting from a safe state (state_ok), in a safe environment (env_ok) either:

- diverges, producing a (potentially infinite) trace of ok_event I/O events; or
- ends in a state_ok post-state and results in an env_ok environment.

The majority of the proof of Theorem 1 follows any run-of-the-mill CakeML simulation proof. The most interesting case is when a kernel function is applied to an argument, since this is the only case where soundness-critical values are not simply being propagated.

For each kernel function, we prove a safety result stating that, when the kernel function is applied to v_ok arguments, it produces v_ok results. For these function-specific safety proofs, we make use of theorems from prior work on verification of the functions of a HOL kernel.

- The CakeML code implementing the kernel's functions is automatically generated by a proof-producing code synthesis tool [50]. This tool proves,

for each shallow embedding of a kernel function f, that, if the CakeML code generated for f is given arguments of the correct form/type, then the CakeML code will compute the same result as an application of f to those arguments (at the level of the shallow embedding of f).

- From prior work [41], we have a soundness theorem for each kernel function. These theorems state that when they are applied to well-formed HOL values (i.e. satisfying inferred), then they produce well-formed HOL values.

However, there is a challenge here brought by the untyped setting and the assumption "if the CakeML code generated for f is given arguments of the correct form/type". Our untyped setting means that we cannot always immediately know that the arguments passed to the kernel functions are of the right form/type. Instead, all we know is that they satisfy v_ok.

Our solution is to insert dead code that makes the operational semantics perform a dynamic type check. For example, the kernel function `ASSUME` has type `term -> thm`, but the operational semantics does not see that it will only be applied to values that are terms. We insert a `case`-expression that pattern matches on a top-level constructor of the term type (`Var`). This `case`-expression triggers a dynamic type check in our semantics.

```
fun ASSUME tm = ((case tm of Var _ _ => () | _ => ()); ...);
```

The inserted code, i.e. `(case tm of Var _ _ => () | _ => ())`, has no impact on performance since the compiler removes it as dead code.

4.3.5 Towards a top-level soundness theorem

The Candle ITP program is made up from the CakeML basis library, the Candle kernel, and the user-facing REPL. The safety of the REPL is discussed separately in Section 4.4. To instantiate Theorem 1, we need to show that the initial state and environments satisfy state_ok and env_ok, respectively.

The program starts by running the basis library, for which state_ok does not hold: at this stage, the kernel references are not yet allocated, and the counter for type numbering has not yet reached the kernel types. Hence, Theorem 1 is not applicable.

We prove a separate simulation theorem, stating that evaluation of the basis program produces an environment that is env_ok, and a state which contains only v_ok values and with a next type number counter set to the number used by the first HOL datatype definition. The type counter and the number of references grow monotonically during execution, and all values produced by a program refer only to type numbers and reference locations that do not exceed the counts kept in state. Thus all values produced by this execution are trivially v_ok. From the resulting state, it is possible to define the kernel types and its references, and end up in a state that is state_ok.

Showing that the Candle post-state and post-environment (where the REPL starts executing) is state_ok and env_ok is straightforward but tedious. We automate the process by making use of some simple facts about env_ok environments:

- All kernel functions and values are v_ok by definition.
- When two env_ok environments are merged, the result is also env_ok.
- When one adds a v_ok value to an env_ok environment, the result is also env_ok.

The result is a small piece of custom proof automation which steers HOL4 to a proof showing that the concrete environments of the kernel values are env_ok, thereby allowing us to establish state_ok and env_ok for the setting in which the REPL program executes.

We use these theorems together with Theorem 1 to show that the safety invariants v_ok, env_ok and state_ok are preserved in any subsequent code executed by the program.

4.3.6 Source-level soundness theorem

Our source-level soundness proof builds up to the following top-level soundness theorem stated in terms of CakeML's observable semantics, semantics_prog.

Here semantics_prog returns a set of behaviours. A behaviour is Fail (for type error), Terminate k l (for termination) or Diverge ll (for a non-terminating run). Here l is a list of I/O events performed by the run and ll is a potentially infinite list of I/O events.

We prove that each generated I/O event must be well-formed according to ok_event, which is defined to require that any event that communicates on the special kernel_ffi channel must contain output that can be produced using a thm_to_string function applied to a sequent th that can be derived (THM) by the inference rules of higher-order logic in a context $ctxt$.

ok_event (IO_event n out y) =
  n = kernel_ffi $\Rightarrow$ $\exists$ $ctxt$ th. THM $ctxt$ th $\wedge$ thm_to_string $ctxt$ th = out

Since the thm_to_string function is crucial for our soundness theorem, we have made sure that its output is invertible. The actual output is not particularly human readable in most cases, but it is unambiguous. A small sample output is shown in Figure 4.3.

Our source-level soundness theorem states that every event satisfies ok_event, for any non-Fail behaviour and for any program that consists of declarations candle_code ++ $prog$, where $prog$ is any list of declarations that syntactically satisfies every safe_dec.

```
# The following is a theorem of higher-order logic

(Sequent nil (Const T (Tyapp bool)))

# which is proved in the following context

(ConstSpec
   ((T ...))
   (Comb
      (Comb ...)
      (Comb ...)))

(NewConst = (Tyapp fun (Tyvar A) (Tyapp fun (Tyvar A) (Tyapp bool))))

(NewType bool 0)

(NewType fun 2)
```

Figure 4.3. Output of `print_thm` applied to the theorem $\vdash$ T. Here Sequent contains `nil` to indicate that there are no hypothesis on this theorem. This theorem is true in the context where T is defined as T = $(\lambda\, p.\ p) = (\lambda\, p.\ p)$, which is its definition in HOL Light; equality = is a constant of type $\alpha \to \alpha \to$ bool; types `bool` and `fun` are defined. Some excess output is elided (. . .) above.

Theorem 2. Any non-Fail behaviour res that is in the behaviours of candle_code ++ $prog$ will only contain externally visible events that satisfy ok_event.

$$\vdash res \in \text{semantics\_prog}\ (\text{init\_eval\_state\_for}\ cl\ fs)\ \text{init\_env}\ (\text{candle\_code}\ +\!+\ prog) \wedge \text{every safe\_dec}\ prog \wedge res \neq \text{Fail} \Rightarrow \forall e.\ e \in \text{events\_of}\ res \Rightarrow \text{ok\_event}\ e$$

4.4 Construction of a proved-to-be-safe REPL for Candle

The previous section explained how we have proved that any reasonable program, one that satisfies every safe_dec, constructed from the Candle kernel leads to a sound prover. This section explains how we have built a program that satisfies every safe_dec and manages to implement a proved to-be-safe read-eval-print loop (REPL) that provides the kind of user-interaction that theorem proving with HOL Light requires.

4.4.1 CakeML's new Eval source primitive

The most technically demanding part of a REPL is the implementation of the "E" in REPL, i.e., the part that evaluates user input. This "E" must always run safely and efficiently.

To the best of our knowledge, most HOL Light users use HOL Light via the standard OCaml REPL[2] where the "E" compiles user input into bytecode and then interprets the bytecode. For CakeML and Candle, we implement the "E" in REPL as: compile, at runtime, the user input to machine code, drop that machine code into the code segment of the running process and execute the new machine code by performing a jump to it.

Such runtime compilation can be achieved in CakeML by using CakeML's new Eval source primitive. The exact details, implementation and verification of the new Eval source primitive will be the subject of a different publication. However, for this paper, it suffices to have an approximate understanding of its semantics and to know that the Eval primitive has been fully integrated into the CakeML compiler and its proofs.

From a bird's eye view, CakeML's Eval primitive has the following semantics: it expects as input (among other things): a value representing the AST for CakeML source declarations to execute, and a value holding a semantic environment (mappings from names to values and type information); if called correctly, Eval evaluates the given declarations using the supplied environment, and, on successful completion, returns a value holding a new environment that can be used for subsequent calls to Eval.

4.4.2 Building a REPL in CakeML source code

The Eval primitive enables us to implement our REPL conveniently in CakeML source code. The source code for the main loop of the REPL is shown in Figure 4.4. This section attempts to explain the code shown in Figure 4.4.

When planning this implementation, a key insight was that we do not need to prove any input-output-style functional correctness theorem of the REPL. Instead, for the purposes of our top-level soundness theorem, it suffices to implement a REPL that we can prove to be safe. This safety proof needs to result in a theorem stating that a semantics_prog run of the REPL program can never result in the Fail behaviour, as can be seen in the assumption $res \neq \mathsf{Fail}$ in Theorem 2. This insight means that we can leave it up to the user to decide how input is to be read and can leave the pretty printing code quite open ended too, i.e., mostly unverified.

We will illustrate the working of the code in Figure 4.4 using an example. For the sake of the example suppose the `repl` function is given the AST of the following CakeML declaration as the `decs` argument.

```
let x = [1] @ [2];;
```

[2]HOL Light adjusts the OCaml REPL so that it uses a custom HOL Light-specific OCaml parser.

```
01: fun repl (parse, types, conf, env, decs, input_str) =
02:   (* input_str is passed in here only for error reporting purposes *)
03:    case check_and_tweak (decs, types, input_str) of
04:      Inl msg => repl (parse, types, conf, env, report_error msg, "")
05:    | Inr (safe_decs, new_types) =>
06:      (* here safe_decs are guaranteed to not crash;
07:       the last declaration of safe_decs calls !Repl.readNextString *)
08:      case eval (conf, env, safe_decs) of
09:      Compile_error msg => repl (parse, types, conf, env, report_error msg, "")
10:      | Eval_exn e new_conf =>
11:      repl (parse, roll_back (types, new_types), new_conf, env, show_exn e, "")
12:      | Eval_result new_env new_conf =>
13:       (* check whether the program that ran has loaded in new input *)
14:       if !Repl.isEOF then () (* exit if there is no new input *) else
15:          let val new_input = !Repl.nextString in
16:            (* if there is new input: parse the input and recurse *)
17:            case parse new_input of
18:              Inl msg =>
19:            repl (parse, new_types, new_conf, new_env, report_error msg, "")
20:             | Inr new_decs =>
21:            repl (parse, new_types, new_conf, new_env, new_decs, new_input)
22:          end
```

Figure 4.4. CakeML code (in CakeML syntax) implementing the main loop of the new REPL.

As can be seen on line 3 in Figure 4.4, `check_and_tweak` will be applied to the `decs`. We can also see that the `types` argument (the state of the type inferencer) and `input_str` are also passed to `check_and_tweak`. This `check_and_tweak` function will run the type inferencer on the given `decs`. If the type inferencer rejects them, then an error message is returned. If the type inferencer accepts `decs`, then the `check_and_tweak` function will return a tweaked version of the original declarations. For this example, the tweaked declarations are approximately the following. (In reality, the second line uses more specialised functions.)

```
let x = [1] @ [2];;
let _ = print ("x" ^ pp_list pp_int x ^ ": int list\n");;
let _ = (!Repl.readNextString)();;
```

Here the first line is, in this case, exactly the user's input; the second line causes the computed value to be printed to stdout; and, the third line runs a user-settable function for reading the next input. In the general case, the `check_and_tweak` function also adds definitions of pp-functions to the given declarations.

Once these adjusted declarations, called `safe_decs` on line 5, have been generated, the `repl` function hands them over to `eval`, which runs them. Run-

ning these declarations can result in one of three outcomes: the compiler might not be able to compile them (linking error or similar); the evaluation might have caused a top-level exception, which `eval` catches and returns as `e` on line 10; or, if all goes well, the evaluation will return a new declaration-level semantic environment `new_env` and a new compiler configuration `new_conf` on line 12.

From line 12, the `repl` function continues by reading a few references that the execution of `!Repl.readNextString` is to have assigned new values to; a `true` in `!Repl.isEOF` indicates that there is no new input; if input exists, then the new input is in `!Repl.nextString`. In the case of new input, the parser is called on the content of `!Repl.nextString` and the loop begins from the top again.

The loop starts off by evaluating the declarations that correspond to the concrete syntax for `let _ = (!Repl.readNextString)();;`. The initial value of this `Repl.readNextString` reference is a function that returns the content of a user-modifiable start-up file `candle_boot.ml`. This file is supposed to install an appropriate new function in `Repl.readNextString`, which includes a user-configurable parser for ‘ . . . ‘-terms, and support for special file loading directives.

One can argue that some aspects of the implementation of the `repl` function seem peculiar, e.g., that the call to `!Repl.readNextString` is always appended to the declarations sent to `eval`. Our design of `repl` is arrange this way in order to collect all state changing code into the execution of `eval`, since such a design makes the safety proof simpler.

4.4.3 Proving safety of the REPL

As mentioned above, we need to prove that the REPL is safe to execute. More specifically, that semantics_prog cannot give the REPL program the Fail behaviour.

The conventional way to prove safety of a CakeML program is via type inference: if the type inferencer accepts a program, then the program is typeable and, by type soundness, we know that the program is safe, i.e., does not have Fail behaviour. Unfortunately, we cannot take this route because the Eval primitive, in its current form, does not fit with CakeML's type system, since static typing information is not enough to show that the Eval won't get stuck when run. As a result, we prove safety of the REPL via an interactive proof.

We prove the following safety theorem for the REPL program called repl_source_prog.

Theorem 3. The REPL program does not have Fail behaviour.

$$\vdash \mathsf{has\_repl\_flag}\ (\mathsf{tl}\ cl) \wedge \mathsf{basis\_init\_ok}\ cl\ fs \Rightarrow \\ \mathsf{Fail} \notin \mathsf{semantics\_prog}\ (\mathsf{init\_eval\_state\_for}\ cl\ fs)\ \mathsf{init\_env}\ \mathsf{repl\_source\_prog}$$

The most challenging aspect of the proof of Theorem 3 is that it requires bringing together results from different parts of the CakeML ecosystem. Fortunately, all proofs to do with type inference could quite cleanly be separated

```
# let T_DEF = new_basic_definition `T = ((\p:bool. p) = (\p:bool. p))`;;
val T_DEF = |- T <=> (\p. p) = (\p. p): thm
# let th1 = SYM T_DEF
  and th2 = REFL `\p:bool. p`;;
val th1 = |- (\p. p) = (\p. p) <=> T: thm
val th2 = |- (\p. p) = (\p. p): thm
# let TRUTH = EQ_MP th1 th2;;
val TRUTH = |- T: thm
```

Figure 4.5. Sample interaction with the Candle REPL in the OCaml syntax of HOL Light.

from the proofs about stepping through the operational semantics. It is worth noting that the REPL implementation uses the type inferencer to establish safety of the user-provided code, which means that the user can unfortunately not currently mention the Eval primitive because Eval is not typeable.

We also prove the following syntactic property about the REPL program in order to meet the assumptions of the Candle soundness theorem, Theorem 2.

Theorem 4. The REPL program has candle_code as a prefix and satisfies every safe_dec.

$$\vdash \exists \mathit{prog}.\ \mathsf{repl\_source\_prog} = \mathsf{candle\_code} \mathbin{+\!\!+} \mathit{prog} \wedge \mathsf{every\ safe\_dec}\ \mathit{prog}$$

This theorem is proved by rewriting and evaluation. It is used in Section 4.5.

4.4.4 A REPL with a parser for HOL Light-style OCaml syntax

The sharp-eyed reader might have noticed that the CakeML code of Figure 4.4 does not use the OCaml syntax that HOL Light users expect. Instead it uses the standard way to write CakeML code, i.e., in syntax that is aligned with Standard ML. In order to make the user experience as close as possible to that of HOL Light, we have equipped the Candle REPL with a parser for HOL Light's version of OCaml syntax. Figure 4.5 shows a snippet of an interaction with the Candle REPL, where one can see a glimpse of OCaml-style concrete syntax supported by Candle. Figure 4.5 also shows our quote filter in action: it processes the quoted terms ‘ . . . ‘ correctly.

4.5 Proving soundness for the machine-code implementation

In this section, we apply the compiler to the source-level REPL implementation of Candle, and transport the safety and soundness proofs down to the level of the machine code that runs when the Candle prover is used.

We evaluate the CakeML compiler on the repl_source_prog program inside HOL4 in order to arrive at the concrete machine_code implementation of the REPL by proof in the logic. The resulting theorem is the following.

Theorem 5. The CakeML compiler produces machine_code when applied to repl_source_prog.

$$\vdash \text{compile init\_conf repl\_source\_prog} = \text{Some (machine\_code,ro\_data,result\_conf)}$$

We use the CakeML compiler's correctness theorem to transport correctness properties down to the level of machine code. Theorem 6 below is an instantiated version of the relevant CakeML compiler correctness theorem. In this theorem, we collect a bunch of assumptions into a constant repl_ready_to_run $cl\ fs\ ms$ which, among other things, requires that the generated machine_code is installed in memory in machine state ms and the program counter of ms points at the start of the code.

Theorem 6. If the source-level program repl_source_prog does not have Fail behaviour, then any machine-code level execution starting from a repl_ready_to_run machine state ms can only produce behaviours that are contained in the set of source-level behaviours (extended with the possibility of early exits due to hitting resource limits, extend_with_resource_limit).

$$\begin{aligned}\vdash\ & \text{Fail} \notin \text{semantics\_prog (init\_eval\_state\_for } cl\ fs) \text{ init\_env repl\_source\_prog} \wedge \\ & \text{repl\_ready\_to\_run } cl\ fs\ ms \Rightarrow \\ & \text{machine\_sem (basis\_ffi } cl\ fs)\ ms \subseteq \\ & \text{extend\_with\_resource\_limit} \\ & \quad (\text{semantics\_prog (init\_eval\_state\_for } cl\ fs) \text{ init\_env repl\_source\_prog})\end{aligned}$$

We will not go into details of extend_with_resource_limit, but only note that it is trivial to prove the following interaction between events_of and extend_with_resource_limit.

$$\begin{aligned}\vdash\ & e \in \text{events\_of } res_1 \wedge res_1 \in sem_1 \wedge sem_1 \subseteq \text{extend\_with\_resource\_limit } sem_2 \Rightarrow \\ & \exists\, res_2.\ e \in \text{events\_of } res_2 \wedge res_2 \in sem_2\end{aligned}$$

We now have all of the parts required to prove a soundness theorem for Candle that relates the level of machine code to the ok_event from Section 4.3.

Theorem 7. Any behaviour res of a machine execution from a repl_ready_to_run machine state ms will not Fail, and any event e in res will always satisfy ok_event.

$$\begin{aligned}\vdash\ & res \in \text{machine\_sem (basis\_ffi } cl\ fs)\ ms \wedge \text{repl\_ready\_to\_run } cl\ fs\ ms \Rightarrow \\ & res \neq \text{Fail} \wedge \forall\, e.\ e \in \text{events\_of } res \Rightarrow \text{ok\_event } e\end{aligned}$$

The proof of this theorem is a simple combination of the source-level soundness theorem (Theorem 2), the two theorems about the source-level REPL

program (Theorems 3 and 4), and the instantiated compiler correctness theorem (Theorem 6).

The theorem above states that any program built inside the Candle REPL can only ever export statements that are true according to the inference rules of higher-order logic.

4.6 Porting HOL Light scripts to Candle

Candle aims to be a verified clone of HOL Light. While the previous sections have focused on the verified part of the system, it is important to note that there is much more to HOL Light than the kernel and the basic setup of the REPL. In this section, we describe our efforts to port HOL Light's standard library to Candle.

At the time of writing, our porting efforts are still work in progress. Candle runs the majority of the scripts HOL Light executes at startup, as well as many proof scripts in the `100` and `Library` directories. Figure 4.6 shows a side-by-side comparison of the Candle and HOL Light REPLs.

The rest of this section describes the changes and additions we make when porting HOL Lights scripts to Candle.

4.6.1 Changes necessary in HOL Light scripts

With our new parser, the CakeML language supports most, but not all, of the language features HOL Light expects of its compiler. Here are the adaptations that we have made to the original HOL Light sources in order to make them compatible with Candle:

- The OCaml stdlib and CakeML's basis library uses different naming conventions. The effect of this on our efforts is mostly mitigated by HOL Light's own 'standard library' implementation `lib.ml`. However, some functions are present in both CakeML and OCaml (e.g. `String.sub`) but with different type signatures or semantics. In such cases, we replace OCaml names with the corresponding CakeML name.

- HOL Light makes use of OCaml's polymorphic comparison operator (`Pervasives.compare`). Where possible, we have replaced all such code with concretely typed comparison operators (e.g., `Int.compare` for integers).

- HOL Light makes use of OCaml's polymorphic hash function `Hashtbl.hash` which maps arbitrary data to the integers. We don't have anything of the sort, and have to rewrite or remove this code.

- CakeML's type system imposes a value restriction. This is mostly a nuisance, but some code has to be re-structured as a consequence.

```
# g `!(x:A) y z. (x = y)
      /\ (y = z) ==> (x = z)`;;
1 subgoal (1 total)

`!x y z. x = y /\ y = z ==> x = z`

val it = (): unit
# e (REPEAT STRIP_TAC);;
1 subgoal (1 total)

  0 [`x = y`]
  1 [`y = z`]

`x = z`

val it = (): unit
# e (PURE_ASM_REWRITE_TAC []);;
1 subgoal (1 total)

  0 [`x = y`]
  1 [`y = z`]

`z = z`

val it = (): unit
# e REFL_TAC;;
No subgoals

val it = (): unit
```

```
# g `!(x:A) y z. (x = y)
      /\ (y = z) ==> (x = z)`;;
val it : goalstack = 1 subgoal (1 total)

`!x y z. x = y /\ y = z ==> x = z`

# e (REPEAT STRIP_TAC);;
val it : goalstack = 1 subgoal (1 total)

  0 [`x = y`]
  1 [`y = z`]

`x = z`

# e (PURE_ASM_REWRITE_TAC []);;
val it : goalstack = 1 subgoal (1 total)

  0 [`x = y`]
  1 [`y = z`]

`z = z`

# e REFL_TAC;;
val it : goalstack = No subgoals
```

Figure 4.6. Side-by-side comparison of an interactive tactic proof in Candle (left) and HOL Light (right). Here `g` sets up a new proof goal and `e` applies a given tactic to the top goal.

- Our parser does not deal with `let rec` forms without explicit arguments. We have to give fresh arguments to such definitions manually.
- At present, CakeML does not support `open` or `include`. We have to manually bring values into scope.
- A few files in the HOL Light basis make use of OCaml's record syntax. CakeML does not support record types at present. We omit these files in our current builds, but must either implement record types in CakeML or rework these files in order to include them.
- All special pragmas recognisable by the OCaml REPL (e.g. for installing pretty-printers) are removed. The CakeML REPL has a different way of dealing with pretty printers.

- We have made minor changes throughout HOL Light files: `hol.ml`, `system.ml`, and `lib.ml`.

4.6.2 Additional scripts

Here are the additions required for our REPL to be able to support HOL Light:

- Added file: `candle_pretty.ml` (Replacement for `Format`)

  We implement a functional pretty printer from a tree of pretty-printer tokens to a tree of string lists. We build an imperative interface on top of the functional printer, modelled after (a small subset of) the interface provided by the `Format` module. (250 loc.)

- Added file: `candle_nums.ml` (Replacement for `Num`)

  The `Num` library integrates both arbitrary precision integers and arbitrary precision rational numbers in one type. All integers in CakeML are arbitrary precision, and CakeML has a library for rational number arithmetic. We build a small wrapper around the CakeML integers and rationals, and provide the interface which HOL Light expects. (285 loc.)

- Added file: `candle_boot.ml` (REPL code)

  We build a read-eval-print loop (REPL) on top of the functionality provided by the CakeML compiler (see Section 4.4). The REPL splits user input into chunks separated by `;;`-tokens at the top-level. It supports multi-line editing, and configurable quote substitution, and a mechanism for file loading that can deal with recursive load calls.

- Added file: `candle_kernel.ml` (essentially the same as `open Kernel`)

  Our current workaround for CakeML's lack of support for `open`, as in `open Kernel`.

4.7 Related work

In this section, we describe related work in the area of verification of interactive theorem provers and their logics. We observe that Candle seems to be the first verified interactive theorem prover that combines: an expressive hosted logic (higher-order logic), an interactive implementation, and an end-to-end soundness theorem that reaches down to the machine code that executes the prover implementation.

4.7.1 Higher-order logic

Harrison [33] formalised a version of the HOL Light logic (omitting its definitional mechanisms) as well as its set-theoretic semantics, in HOL Light itself.

The actual artefact being verified is a shallow-embedded implementation of HOL Light, shown to be sound with respect to the semantics. Two consistency results are proved: HOL without the axiom of infinity is shown consistent in HOL; and HOL is shown consistent in HOL extended with a larger universe of sets. However, the scope of verification does not extend past the shallow embedding; there is no formal connection between it and the actual system, which runs on interpreted OCaml and its C runtime.

Our work rests heavily on the work by Kumar et al. [41]. They build on Harrison's work and expand it along several dimensions; they formalise the definitional mechanisms omitted by Harrison [33] using contexts, and contribute a sequent calculus which is proven sound with respect to the HOL semantics. A shallow-embedded implementation is shown to refine the proof calculus. Notably, this shallow embedding can be extracted to machine code using the CakeML ecosystem, establishing a formal connection between the model-theoretic semantics and the machine code executing the kernel functions.

Gengelbach and Åman Pohjola [54, 24] further extend the work by Kumar et al. [41], adding support for ad-hoc overloading of constant definitions. This sort of mechanism is used by e.g. Isabelle/HOL to let one logical constant receive different meanings depending on what concrete types the variables in its type signature are instantiated to. At the time of writing, work on a verified cyclicity checker, which is required to ensure the soundness of instantiations of overloaded constants, has recently been completed [23]. We see no reason that our work should not build on their kernel implementation in the future.

Nipkow and Roßkopf [52] have formalised the meta-logic of Isabelle, as well as its proof-terms, and a proof checker for its proof terms. The meta-logic is used in Isabelle to define its many object logics. They formalise a proof calculus (but not a semantics) for the meta-logic in Isabelle/HOL, and implement and verify the correctness of a proof-checker for Isabelle proof-terms. Using Isabelle's (unverified) code extraction, they are able to obtain an executable checker in Standard ML. This checker can be used to check real proofs of Isabelle theorems within Isabelle, but relies on an unverified translation from Isabelle's actual proof structures into the proof-terms used by the checker. In addition, one must trust the Poly/ML compiler and its C++ runtime, which hosts both Isabelle and the checker artefact.

4.7.2 First-order logic

The most comprehensive ITP verification result prior to ours is Milawa by Davis and Myreen [18]. Milawa implements a quantifier-free fragment of first-order logic with recursive functions in the spirit of Nqthm and ACL2. It can execute on top of the verified Jitawa [49] Lisp runtime, and is proven sound with respect to a formal semantics. By verifying and implementing both the prover and its runtime within HOL4, the authors are able to obtain a soundness theorem which shows the soundness of the machine code that executes the Milawa system at runtime. The scope of Milawa's verification is similarly far-reaching

to ours. However, it implements a fragment of first-order logic with functions, which is simpler than HOL, and relies on a Lisp runtime which is considerably less complex than the ML compilers used by LCF-style systems. An interesting feature of Milawa is its ability to bootstrap itself by successively performing conservative extensions of its own proof checker; no LCF-style system can accomplish this as far as we know.

MetaMath Zero by Carneiro [16] is a proof checker for a many-sorted first-order logic. Its logic is intended to act as a host for other object logics. A bootstrapping effort is ongoing; the proof rules of MetaMath Zero have been formalised within MetaMath Zero itself, as well as a model of the x86-64 ISA [15]. However, there is no formal connection between the formalised proof rules and any machine code refinement of said rules: the current checker implementation is unverified. As it lacks a formal semantics, verification is limited to correctness of the proof calculus, leaving out soundness. Compared to our system, MetaMath Zero is very low-level; its logic is simpler, and it offers no interactivity or proof automation, and cannot be extended at runtime. In return, one does not have to trust nor verify a complex programming language implementation. Still, the language appears practical enough to host itself and a ISA artefact [15]. Because it lacks interactivity, users do not interact directly with MetaMath Zero. Instead, they see a higher-level (unverified) ITP-like system called MM1, which produces proofs that are checked by MetaMath Zero; a design somewhat similar in spirit to that of LCF-style systems.

4.7.3 Dependent type theory

Barras [10] formalised the Calculus of Constructions (CC), a simpler version of the Calculus of Inductive Constructions (CIC) which is the type theory used by Coq. Barras' formalisation is done in the same spirit as Harrison's work [33], by defining a set-theoretic model of the calculus, and proving its soundness wholly inside Coq itself.

Sozeau et al. [59] present a formalisation as well as a proven-correct efficient type checker implementation for a substantial part of CIC. Their work continues the tradition of Harrison [33], Kumar et al. [41] and Barras [10], by formalising the meta-theory of Coq in Coq. The fragment of CIC under consideration omits modules and template polymorphism. An OCaml version of the verified type checker can be obtained using the unverified Coq extraction mechanism. Due to faults in the implementation of Coq's code extraction, Sozeau et al. implement and verify their own extraction mechanism, which can be used to obtain an executable checker. However, the formal verification of the extraction is done against an untyped λ-calculus, and not the actual OCaml language.

Anand and Rahli [7] have formalised the proof calculus and semantics of Nuprl in Coq, and proved soundness of the Calculus. They do not provide a verified program which implements the calculus, but their plan is extract a verified implementation from the formalisation of their calculus, and to implement and verify a type checker for a large part of Nuprl.

4.8 Summary

The result of our efforts is an interactive theorem prover called Candle that:

1. has been proved to be sound down to the machine code that runs it (the binary is guaranteed to only output facts that are sound w.r.t. the rules of higher-order logic);

2. offers a user experience that we have made as similar as possible to that of HOL Light (Candle supports the same syntax and interactive proof manager as HOL Light).

To the best of our knowledge, Candle is the most complete and comprehensive verified LCF-style interactive theorem prover to date.

Future work All of the proofs about the Candle prover have been completed, but some practical challenges remain before Candle can be considered a drop-in replacement for HOL Light. Most importantly, we need to port the remainder of the HOL Light base libraries.

Acknowledgements We want to thank Freek Wiedijk and Yong Kiam Tan. We are grateful for Freek Wiedijk's question at ITP'11. Following a presentation about the verification of a runtime for Milawa [49] at ITP'11, Wiedijk asked: "Can you do the same for HOL Light, please?" Wiedijk's question can be seen as the seed that set us thinking about the possibility of a verified HOL Light implementation and eventually lead us to construct the verified Candle ITP, presented in this paper. We want to thank Yong Kiam Tan for helping with some proofs involving the the CakeML type inferencer. These proofs were part of the proof of safety of CakeML's new read-eval-print loop.

Chapter 5

Fast, Verified Computation for Candle

Oskar Abrahamsson and Magnus O. Myreen

Abstract. This paper describes how we have added an efficient function for computation to the kernel of the Candle interactive theorem prover. Candle is a CakeML port of HOL Light which we have, in prior work, proved sound w.r.t. the inference rules of the higher-order logic. This paper extends the original implementation and soundness proof with a new kernel function for fast computation. Experiments show that the new computation function is able to speed up certain evaluation proofs by several orders of magnitude.

Technical report, 2022.

5.1 Introduction

Interactive theorem provers (ITPs) include facilities for computing within the hosted logic. To illustrate what we mean by such a feature, consider the following function, sum, which sums a list of natural numbers:

$$\mathsf{sum}\ xs = \mathsf{if}\ xs = []\ \mathsf{then}\ 0\ \mathsf{else}\ \mathsf{hd}\ xs + \mathsf{sum}\ (\mathsf{tl}\ xs)$$

A facility for computing within the logic can be used to automatically produce theorems such as the following, where sum $[5;\ 9;\ 1]$ was given as input, and the following equation is the output, showing that the input reduces to 15:

$$\vdash \mathsf{sum}\ [5;\ 9;\ 1] = 15 \tag{5.1}$$

The ability to compute such equations in ITPs is essential for use of verified decision procedures, for proving ground cases in proofs, and for running a parser, pretty printer or even compiler inside the logic for a smaller TCB.

Higher-order logic (HOL) does not have a primitive rule for (or notion of) computation. Instead, HOL ITPs such as HOL Light [34], HOL4 [57], and Isabelle/HOL [51] implement computation as a derived rule using rewriting, which in turn is a derived rule implemented outside their trusted kernels. As a result, computation is slow in these systems.

To understand why computation is so sluggish in HOL ITPs, it is worth noting that the primitive steps taken for the computation of Example (5.1) are numerous:

- At each step, rewriting has to match the subterm that is to be reduced next (according to a call-by-value order) against each pattern it knows (the left-hand side of the definitions of sum, hd, tl, if-then-else and more); when a match is found, it needs to instantiate the equation whose left-hand-side matched, and then reconstruct the surrounding term.
- Computation over natural numbers is far from constant-time, since 5, 9 and 1 are syntactic sugar for numerals built using the constructors Bit0, Bit1 and 0. For example, 5 = Bit1 (Bit0 (Bit1 0)). Deriving equations describing the evaluation of simple operations such as $+$ requires rewriting with lemmas such as these:

$$\begin{aligned}
&\mathsf{Bit1\ m} + \mathsf{Bit0\ n} = \mathsf{Bit1}\ (m + n)\\
&\mathsf{Bit1\ m} + \mathsf{Bit1\ n} = \mathsf{Bit0}\ (\mathsf{Suc}\ (m + n))\\
&\mathsf{Suc}\ (\mathsf{Bit0\ n}) = \mathsf{Bit1\ n}\\
&\mathsf{Suc}\ (\mathsf{Bit1\ n}) = \mathsf{Bit0}\ (\mathsf{Suc}\ n)\\
&\ldots
\end{aligned}$$

HOL ITPs employ such laborious methods for computation in order to keep their soundness critical kernel as small as possible: the small size and simplicity of the kernel is key to the soundness argument.

This paper is about how we have added a fast function for computation to the Candle HOL ITP[1]. Candle has a different soundness argument that allows it to move away from being simple in order to be trustworthy: Candle has been proved (in HOL4) to be sound w.r.t. a formal semantics of higher-order logic [2].

With this new function for computation, proving equations via computation is cheap. For the sum example:

- The input term is traversed once, and is converted to a datatype better suited for fast computation. In this representation, each occurrence of sum, hd, tl, etc. can be expanded directly without pattern-matching.

- The representation makes use of host-language integers, so that the expression $5 + (9 + (1 + 0))$ can be computed using three native addition operations.

- Once the computation is complete, the result is converted back to a HOL term and an equation similar to (5.1) is returned to the user.

Our function for computation works on a first-order, untyped, monomorphic subset of higher-order logic. Our implementation interprets terms of this subset using a call-by-value strategy while utilizing host-language (CakeML) features such as arbitrary precision integer arithmetic.

In our experiments, we observe speed gains of several orders of magnitude when comparing our new compute function against in-logic computation implementations used by other HOL ITPs (Sec. 5.8).

Contributions

We make the following contributions:

- We implement a fast interpreter for terms as a user-accessible primitive in the Candle kernel. The implementation allows users to supply code equations dictating how user-defined (recursive) functions are to be interpreted.

- The new primitive has been proven correct with respect to the inference rules of higher-order logic, and has been fully integrated into the existing end-to-end soundness proof of the Candle ITP.

- Our compute function is, in our experiments, significantly faster than the equivalent runs of in-logic compute facilities provided by other HOL ITPs.

[1]Kernel functions are analogous to inference rules in HOL implementations.

Notation: $=$ and $=_c$, $\vdash$ and $\vdash_c$, etc.

This paper contains syntax at multiple, potentially confusing levels. The Candle logic is formalized inside the HOL4 logic. Symbols that exist in both logics are suffixed by a subscript $_c$ in its Candle version; as an example, $=$ denotes equality in the HOL4 logic, and $=_c$ denotes equality in the embedded Candle logic. Likewise, a theorem in HOL4 is prefixed by $\vdash$, while a Candle theorem is prefixed by $\vdash_c$.

Source code and proofs

Our sources are at code.cakeml.org/candle/prover/compute, and the Candle project is hosted at cakeml.org/candle.

5.2 Approach

This section describes the approach we have taken to add a new function for computation to Candle.

First, we introduce a new computation friendly internal representation (IR) for expressions that we want to do computation on. On entry to the new compute primitive, the given input term is translated into this new IR. We use an IR that is separate from the syntax of HOL (theorems, terms and types), since the datatypes used by HOL ITPs are badly suited for efficient computation.

We perform computation on the terms of our IR via interpretation. On termination, this interpretation arrives at a return value. This return value is translated to a HOL term r. The new compute primitive returns, to the user, a theorem stating that the input term is equal to the result of computation r. This series of steps is illustrated by the solid arrows in Figure 5.1.

The new compute primitive is a user-accessible function in the Candle kernel and must therefore be proved to be sound, i.e., every theorem it returns must follow by the primitive inference rules of higher-order logic (HOL).

We prove the correctness of our computation function by showing that there is some way of using the inference rules of HOL to mimic the operations of the interpreter. Our use of the inference rules is essentially the same as building a proof by rewriting in the logic of Candle.

The connection established by the existentially quantified proof is illustrated by the dashed arrow in Figure 5.1. All reasoning about the interpreter (the lower horizontal arrow) must be wrt. the view of the interpreter provided by the translations to and from the IR (the vertical arrows). Nearly all of our theorems are stated in terms of the arrow upwards, i.e. from IR to HOL.

Overview

The development of our new compute primitive for Candle was staged into increasingly complex versions.

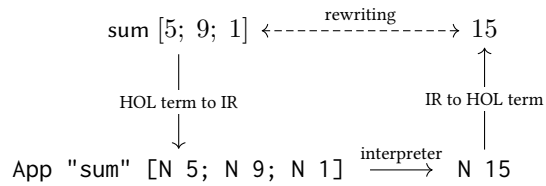

Figure 5.1. Diagram illustrating the approach we take to embedding logical terms into compute expressions and evaluating them using an interpreter.

1. Version 1 (Sec. 5.3) was a proof-of-concept Candle function for computing the result of additions of concrete natural numbers. This function was implemented as a conversion[2] in the Candle kernel that given a term $m \ +_{\mathsf{c}} \ n$ computes the result of the addition r, and returns a theorem $\vdash_{\mathsf{c}} \ m \ +_{\mathsf{c}} \ n \ =_{\mathsf{c}} \ r$ to the user. Internally, the implementation makes use of the arbitrary precision integer arithmetic of the host language, i.e. CakeML. The purpose of Version 1 was to establish the concepts needed for this work rather than producing something that is actually useful from a user's point of view.

2. Version 2 (Sec. 5.4) improved on Version 1 by replacing the type of natural numbers by a datatype for binary trees with natural numbers at the leaves, and by supporting structured control-flow (if-then-else), projections (fst, snd) and the usual arithmetic operations. This version supports nesting of expressions.

3. Version 3 (Sec. 5.5) extended Version 2 with support for user-supplied code equations for user-defined constants. The code equations are allowed to be recursive and thus the interpreter had to support recursion. This extension also brought with it variables: from Version 3 and on, all interpreters are able to interpret input terms containing variables.

4. Version 4 (Sec. 5.6) replaced the naive interpreter with one that is designed to evaluate with less overhead. This version uses $O(1)$ operations to look up to code equations and uses environments rather than substitutions for variable bindings. This is the version we perform benchmarks on (Sec. 5.8).

5. The final Version 5 (Sec. 5.7) is, at the time of writing, left as future work. In Version 5, our intention is to split the compute function into stages so that users can initialize and feed in code equations separately from calls to the main compute function. This should make repeated calls to the compute facility faster.

[2] A *conversion* is a proof procedure that takes a term t as input and proves a theorem $\vdash \ t = t'$ for some interesting t'.

At the time of writing, Version 4 (Sec. 5.6) is integrated into the existing Candle implementation and end-to-end soundness proof.

5.3 Adding Natural Numbers (Version 1)

In this section, we describe how we implemented and verified a function for computing addition on natural numbers in the Candle kernel. This is the first step towards a proven-correct function for computation. The approach can be reused to produce computation functions for other kinds of binary operations (multiplication, subtraction, division, etc.) on natural numbers, and it can be used to build evaluators for arithmetic inside more general expressions (Sec. 5.4).

Input and output

In Version 1, the user can input terms such as $3 +_\mathsf{c} 5$ or $100 +_\mathsf{c} 0$, i.e., terms consisting of one addition applied to two concrete numbers. The numbers are shown here as 3, 5, 100, 0, even though they are actually terms in a binary representation based on the constant 0_c, and the functions Bit0 and Bit1 in the Candle logic.

The output is a theorem equating the input with a concrete natural number. For the examples above, the function returns the following equations. The subscript $_\mathsf{c}$ is used below to highlight that these are theorems in the Candle logic.

$$\vdash_\mathsf{c} 3 +_\mathsf{c} 5 =_\mathsf{c} 8 \quad \text{or} \quad \vdash_\mathsf{c} 100 +_\mathsf{c} 0 =_\mathsf{c} 100$$

The results 8 and 100 are computed using addition outside the logic. The challenge is to show that the same computation can be derived from the equations defining $+_\mathsf{c}$ (in Candle) using the primitive inference rules of the Candle logic.

Key soundness lemma

In order to prove the soundness of Version 1 (required for its inclusion in the Candle kernel), we need to prove the following theorem, which states: if the arithmetic operations are defined as expected (num_thy_ok) in the current Candle theory Γ, then the addition $(+_\mathsf{c})$ of the binary representations (mk_num) of two natural numbers m and n is equal $(=_\mathsf{c})$ to the binary representation of $(m + n)$, where + is HOL4 addition.

$$\begin{array}{l} \vdash \mathsf{num\_thy\_ok}\ \Gamma\ \Rightarrow \\ \quad \Gamma \vdash_\mathsf{c} \mathsf{mk\_num}\ m +_\mathsf{c} \mathsf{mk\_num}\ n =_\mathsf{c} \\ \qquad \mathsf{mk\_num}\ (m + n) \end{array} \tag{5.2}$$

To understand the theorem statement above, let us look at the definitions of mk_num and num_thy_ok. The function mk_num converts a HOL4 natural

number into the corresponding Candle natural number in binary representation:

$$\begin{array}{l}\mathsf{mk\_num}\ n\ =\\ \quad \mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{0_c}\\ \quad \mathsf{else\ if}\ \mathsf{even}\ n\ \mathsf{then}\ \mathsf{Bit0}\ (\mathsf{mk\_num}\ (n\ \mathsf{div}\ 2))\\ \quad \mathsf{else}\ \mathsf{Bit1}\ (\mathsf{mk\_num}\ (n\ \mathsf{div}\ 2))\end{array}$$

The definition of num_thy_ok asserts that various characterizing equations hold for the Candle constants $+_c$, Bit0 and Bit1 (the complete definition is not shown below). Here m and n are natural number typed variables in Candle's logic:

$$\begin{array}{l}\mathsf{num\_thy\_ok}\ \Gamma\ =\\ \quad \Gamma \vdash_c \mathsf{0_c} +_c \mathsf{n} =_c \mathsf{n} \wedge\\ \quad \Gamma \vdash_c \mathsf{Suc\ m} +_c \mathsf{n} =_c \mathsf{Suc}\ (\mathsf{m} +_c \mathsf{n}) \wedge\\ \quad \Gamma \vdash_c \mathsf{Bit0\ n} =_c \mathsf{n} +_c \mathsf{n} \wedge\\ \quad \Gamma \vdash_c \mathsf{Bit1\ n} =_c \mathsf{Suc}\ (\mathsf{n} +_c \mathsf{n}) \wedge \ldots\end{array}$$

We use num_thy_ok as an assumption in Theorem (5.2), since the computation function is part of the Candle kernel, which does not include these definitions when the prover starts from its initial state (and thus the user might define them differently).

A closer look at num_thy_ok reveals that $+_c$ is characterized by its simple Suc-based equations and Bit1 is characterized in terms of Suc and $+_c$. As a result, a direct proof of Theorem (5.2) would be awkward at best.

To keep the proof of Theorem (5.2) as neat as possible, we defined the expansion of a HOL number into a tower of Suc applications to 0_c:

$$\begin{array}{l}\mathsf{mk\_suc}\ n\ =\\ \quad \mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{0_c}\\ \quad \mathsf{else\ Suc}\ (\mathsf{mk\_suc}\ (n-1))\end{array}$$

and split the proof of Theorem (5.2) into two lemmas. The first lemma is a mk_suc variant of Theorem (5.2):

$$\begin{array}{l}\vdash \mathsf{num\_thy\_ok}\ \Gamma\ \Rightarrow\\ \quad \Gamma \vdash_c \mathsf{mk\_suc}\ m +_c \mathsf{mk\_suc}\ n =_c\\ \qquad \mathsf{mk\_suc}\ (m+n)\end{array} \tag{5.3}$$

and the second lemma $=_c$-equates mk_num with mk_suc:

$$\begin{array}{l}\vdash \mathsf{num\_thy\_ok}\ \Gamma\ \Rightarrow\\ \quad \Gamma \vdash_c \mathsf{mk\_num}\ n =_c \mathsf{mk\_suc}\ n\end{array} \tag{5.4}$$

The proof of Theorem (5.3) was done by induction on m, and involved manually constructing the $\vdash_c$-derivation that connects the two sides of $=_c$ in Theorem (5.3). The proof of Theorem (5.4) is a complete induction on n and uses Theorem (5.3) when $+_c$ is encountered. Finally, the proof of Theorem (5.2) is a manually constructed $\vdash_c$-derivation that uses the Theorems (5.4) and (5.3), and symmetry of $=_c$.

From Candle terms to natural numbers

The development described above is in terms of functions (mk_num, mk_suc) that map HOL4 natural numbers into Candle terms, but the implementation also converts in the opposite direction: on initialization, the computation function converts the given input term into its internal representation (see the leftmost arrow in Figure 5.1).

We use the following function, dest_num, to extract a natural number from a Candle term. This function traverses terms, and recognizes the function symbols used in Candle's binary representation of natural numbers:

$$
\begin{array}{l}
\mathsf{dest\_num}\ tm\ = \\
\quad \mathsf{case}\ tm\ \mathsf{of} \\
\quad |\ 0_{\mathsf{c}}\ \Rightarrow\ \mathsf{Some}\ 0 \\
\quad |\ \mathsf{Bit0}\ r\ \Rightarrow\ \mathsf{option\_map}\ (\lambda\, n.\ 2 \times n)\ (\mathsf{dest\_num}\ r) \\
\quad |\ \mathsf{Bit1}\ r\ \Rightarrow\ \mathsf{option\_map}\ (\lambda\, n.\ 2 \times n + 1)\ (\mathsf{dest\_num}\ r) \\
\quad |\ \_\ \Rightarrow\ \mathsf{None}
\end{array}
$$

One should read the application $\mathsf{Bit}b\ \ bs$ as a natural number in binary with least significant bit b and other bits bs.

The correctness of dest_num is captured by the following theorem, which states that $=_{\mathsf{c}}$ is preserved when moving from Candle terms to natural numbers in HOL4, and back:

$$
\begin{array}{l}
\vdash\ \mathsf{num\_thy\_ok}\ \Gamma\ \wedge \\
\quad \mathsf{dest\_num}\ t\ =\ \mathsf{Some}\ t'\ \Rightarrow \\
\quad\quad \Gamma\ \vdash_{\mathsf{c}}\ \mathsf{mk\_num}\ t'\ =_{\mathsf{c}}\ t
\end{array}
\tag{5.5}
$$

Version 1 of the computation function also has a function for taking apart a Candle term with a top-level addition $+_{\mathsf{c}}$:

$$
\begin{array}{l}
\mathsf{dest\_add}\ tm\ = \\
\quad \mathsf{case}\ tm\ \mathsf{of} \\
\quad |\ (x\ +_{\mathsf{c}}\ y)\ \Rightarrow\ \mathsf{Some}\ (x,y) \\
\quad |\ \_\ \Rightarrow\ \mathsf{None}
\end{array}
$$

Equipped with the functions dest_num and dest_add, and the Theorems (5.2) and (5.5), it is easy to prove the following soundness result. This theorem states: if a term t can be taken apart using dest_add and dest_num, then the term constructed by mk_num and the HOL4 addition, $+$, can be used as the right-hand side of an equation that is $\vdash_{\mathsf{c}}$-derivable.

$$
\begin{array}{l}
\vdash\ \mathsf{num\_thy\_ok}\ \Gamma\ \Rightarrow \\
\quad \mathsf{dest\_add}\ t = \mathsf{Some}\ (x,y)\ \wedge \\
\quad \mathsf{dest\_num}\ x = \mathsf{Some}\ m\ \wedge \\
\quad \mathsf{dest\_num}\ y = \mathsf{Some}\ n\ \Rightarrow \\
\quad \Gamma\ \vdash_{\mathsf{c}}\ t\ =_{\mathsf{c}}\ \mathsf{mk\_num}\ (m\ +\ n)
\end{array}
\tag{5.6}
$$

This theorem can be used as the blueprint for an implementation that uses dest_add, dest_num and mk_num.

Checking num_thy_ok

Note that Theorem (5.6) assumes num_thy_ok, which requires certain equations to be true in the current theory Γ. To be sound, an implementation of our computation function must check that this assumption holds.

We deal with this issue in a pragmatic manner, by requiring that the user provides a list of theorems corresponding to the equations of num_thy_ok on each invocation of our computation function. This approach makes num_thy_ok easy to establish, but causes extra overhead on each call to the computation function. Subsequent versions will remove this overhead (Sec. 5.7).

Soundness of CakeML implementation

Throughout this section, we have treated functions in the logic of HOL4 as if they were the implementation of the Candle kernel. We do this because the actual CakeML implementation of the Candle kernel is automatically synthesized from these functions in the HOL4 logic.

Updating the entire Candle soundness proof for the addition of Version 1 of the compute function was straightforward, once Theorem (5.6) was proved and the code for checking num_thy_ok was verified.

5.4 Compute Expressions (Version 2)

This section describes Version 2, which generalizes the very limited Version 1. While Version 1 only computed addition of natural numbers, Version 2 can compute the value of any term that fits in a subset of Candle terms that we call *compute expressions*. Compute expressions operate over a Lisp-inspired datatype which we call *compute values*; in Candle, this type is called cval.

Even though this second version might at first seem significantly more complicated than the first, it is merely a further development of Version 1. The approach is the same: the soundness theorems we prove are very similar looking. Technically, the most significant change is the introduction of a datatype, cexp, that is the internal representation of all valid input terms, i.e., compute expressions.

Compute values

To the Candle user, the following cval datatype is important, since all terms supplied to the new compute function must be of this type. The cval datatype is a Lisp-inspired binary tree with natural numbers (num) at the leaves:

$$\begin{aligned} \mathsf{cval} = {} & \mathsf{Pair_c}\ \mathsf{cval}\ \mathsf{cval} \\ \mid {} & \mathsf{Num_c}\ \mathsf{num} \end{aligned}$$

Compute expressions

The other important datatype is cexp, which is the internal representation that user input is translated into:

```
cexp = Pair cexp cexp
     | Num num
     | If cexp cexp cexp
     | Uop uop cexp
     | Binop binop cexp cexp

uop = Fst | Snd | IsPair

binop = Add | Sub | Mul | Div | Mod | Less | Eq
```

The cexp datatype is extended for Version 3, in Section 5.5.

Input terms

On start up, the compute function maps the given term into the cexp type. For example, given this term as input:

$$\mathsf{cif_c}\ (\mathsf{Num_c}\ 1)\ (\mathsf{Num_c}\ 2)\ (\mathsf{fst_c}\ (\mathsf{Pair_c}\ (\mathsf{Num_c}\ 3)\ (\mathsf{Num_c}\ 4)))$$

the function will create this cexp expression:

$$\mathsf{If}\ (\mathsf{Num}\ 1)\ (\mathsf{Num}\ 2)\ (\mathsf{Uop}\ \mathsf{Fst}\ (\mathsf{Pair}\ (\mathsf{Num}\ 3)\ (\mathsf{Num}\ 4)))$$

This mapping assumes that certain functions in the Candle logic (e.g. $\mathsf{fst_c}$) correspond to certain constructs in the cexp datatype (e.g. Uop Fst). Note that there is nothing strange about this: in Version 1, we assumed that $+_\mathsf{c}$ corresponds to addition. We formalize the assumptions about $\mathsf{fst_c}$, etc., next.

Context assumption: cexp_thy_ok

Just as in Version 1, Version 2 also has an assumption on the current theory context. In Version 1, the assumption num_thy_ok ensured that the Candle definition of $+_\mathsf{c}$ satisfied the relevant characterizing equations. For Version 2, this assumption was extended to cover characterizing equations for all names that the conversion from user input to cexp recognizes: $\mathsf{cif_c}$, $\mathsf{fst_c}$, etc.

These characterizing equations fix a semantics for the Candle functions that correspond to constructs of the cexp type. For simplicity, all of the Candle functions take inputs of type cval and produce outputs of type cval.

Our implementation makes no attempt at ensuring that functions are applied to sensible inputs. Consequently, it is perfectly possible to write strange terms in this syntax, such as $\mathsf{fst_c}\ (\mathsf{Num_c}\ 3)$, or $\mathsf{add_c}\ (\mathsf{Num_c}\ 3)\ (\mathsf{Pair_c}\ p\ q)$. We resolve such cases in a systematic way:

- Operations that expect numbers as input treat $\mathsf{Pair_c}$ values as $\mathsf{Num_c}\ 0$.

- Operations that expect a pair as input return $\mathsf{Num_c}\ 0$ when applied to $\mathsf{Num_c}$ values.

This treatment of the primitives can be seen in the assumption, called cexp_thy_ok, that we make about the context for Version 2. Below, x and y are variables in the Candle logic with type cval. The lines specifying $\mathsf{add_c}$ are:

$$\begin{aligned}&\mathsf{cexp\_thy\_ok}\ \Gamma\ =\\ &\ \ldots\ \wedge\\ &\ \Gamma \vdash_c \mathsf{add_c}\ (\mathsf{Num_c}\ \mathsf{m})\ (\mathsf{Num_c}\ \mathsf{n}) =_c \mathsf{Num_c}\ (\mathsf{m} +_c \mathsf{n})\ \wedge\\ &\ \Gamma \vdash_c \mathsf{add_c}\ (\mathsf{Pair_c}\ \mathsf{x}\ \mathsf{y})\ (\mathsf{Num_c}\ \mathsf{n}) =_c \mathsf{Num_c}\ \mathsf{n}\ \wedge\\ &\ \Gamma \vdash_c \mathsf{add_c}\ (\mathsf{Num_c}\ \mathsf{m})\ (\mathsf{Pair_c}\ \mathsf{x}\ \mathsf{y}) =_c \mathsf{Num_c}\ \mathsf{m}\ \wedge\ \ldots\end{aligned}$$

The lines specifying $\mathsf{fst_c}$ are:

$$\begin{aligned}&\Gamma \vdash_c \mathsf{fst_c}\ (\mathsf{Pair_c}\ \mathsf{x}\ \mathsf{y}) =_c \mathsf{x}\ \wedge\\ &\Gamma \vdash_c \mathsf{fst_c}\ (\mathsf{Num_c}\ \mathsf{n}) =_c \mathsf{Num_c}\ 0_c\ \wedge\ \ldots\end{aligned}$$

The following characteristic equations for $\mathsf{cif_c}$ illustrate that we treat $\mathsf{Num_c}\ 0_c$ as false and all other values as true:

$$\begin{aligned}&\Gamma \vdash_c \mathsf{cif_c}\ (\mathsf{Num_c}\ 0_c)\ \mathsf{x}\ \mathsf{y} =_c \mathsf{y}\ \wedge\\ &\Gamma \vdash_c \mathsf{cif_c}\ (\mathsf{Num_c}\ (\mathsf{Suc}\ \mathsf{n}))\ \mathsf{x}\ \mathsf{y} =_c \mathsf{x}\ \wedge\\ &\Gamma \vdash_c \mathsf{cif_c}\ (\mathsf{Pair_c}\ \mathsf{x'}\ \mathsf{y'})\ \mathsf{x}\ \mathsf{y} =_c \mathsf{x}\ \wedge\ \ldots\end{aligned}$$

Comparison primitives return $\mathsf{Num_c}\ 1$ for true.

Soundness

The following theorem summarizes the operations and soundness of Version 2. If a term t can be successfully converted (using dest_term) into a compute expression $cexp$, then t is equal to a Candle term created (using mk_term) from the result of evaluating $cexp$ using a straightforward evaluation function (cexp_eval):

$$\begin{aligned}&\vdash \mathsf{cexp\_thy\_ok}\ \Gamma\ \Rightarrow\\ &\quad \mathsf{dest\_term}\ t = \mathsf{Some}\ cexp\ \Rightarrow\\ &\quad \Gamma \models_c t =_c \mathsf{mk\_term}\ (\mathsf{cexp\_eval}\ cexp)\end{aligned} \tag{5.7}$$

Note the similarity between the Theorems (5.6) and (5.7). Where Theorem (5.6) uses $+$, Theorem (5.7) calls cexp_eval. The evaluation function cexp_eval is defined to traverse the cexp bottom-up in the most obvious manner, respecting the evaluation rules set by the characterizing equations of cexp_thy_ok.

CakeML code and integration

The functions dest_term, cexp_eval and mk_term are the main workhorses of the implementation of Version 2. Corresponding CakeML implementations

are synthesized from these functions. The definition of the evaluator function cexp_eval uses arithmetic operations ($+$, $-$, $\times$, div, mod, $<$, $=$) over the natural numbers. Such arithmetic operations translate into arbitrary precision arithmetic operations in CakeML.

Updating the Candle proofs for Version 2 was a straightforward exercise, given the prior integration of Version 1.

5.5 Recursion and user-supplied code equations (Version 3)

Version 3 of our compute function for Candle adds support for (mutually) recursive user-defined functions. The user supplies function definitions in the form of *code equations*.

Code equations

In our setting, a code equation for a user-defined constant c is a Candle theorem of the form:

$$\vdash_{\mathsf{c}} \mathsf{c}\ v_1\ \ldots\ v_n = e$$

where each variable v_i has type cval and the expression e has type cval. Furthermore, the free variables of e must be a subset of $\{v_1, \ldots, v_n\}$. Note that any user-defined constants, including c, are allowed to appear in e in fully applied form.

Updated compute expressions

We updated the cexp datatype to allow variables (Var), applications of user-supplied constants (App), and at the same time we added let-expressions (Let):

```
cexp = Pair cexp cexp
     | Num num
     | Var string
     | App string (cexp list)
     | Let string cexp cexp
     | If cexp cexp cexp
     | Uop uop cexp
     | Binop binop cexp cexp
```

Variables are present to capture the values bound by the left-hand sides of code equations and by let-expressions.

The interpreter for Version 3 of our compute function uses a substitution-driven semantics, and it stores code equations in a list that is indexed by function names. This style of semantics maps well to the Candle logic's substitution primitive, thus simplifying verification, but at a price:

- At each let-expression or function application, the entire body of the let-expression or the code equation corresponding to the function may be traversed an additional time, to substitute out variables.
- At each function application, the code equation corresponding to the function name is found using linear search, making the interpreter less efficient as more code equations are used.

We address these shortcomings in Version 4 of our compute function, in Section 5.6.

Soundness

The following theorem is the essential part of the soundness argument for Version 3. The user supplies the Version 3 compute function with: a list of theorems that allows it to establish cexp_thy_ok, a list eqs of code equations, and a term t to evaluate. Every theorem in eqs must be a Candle theorem ($\vdash_{\mathsf{c}}$). Definitions $defs$ are extracted from the given code equations eqs. A compute expression $cexp$ is extracted from the given input term w.r.t. the available definitions $defs$. An interpreter, interpret, is run on the $cexp$, and its execution is parameterized by $defs$ and a clock which is initialized to a large number init_ck. If the interpreter returns a result res, i.e. Some res, then an equation between the input term t and mk_term res can be returned to the user.

$$\begin{array}{l} \vdash \mathsf{cexp\_thy\_ok}\ \Gamma\ \Rightarrow \\ \quad (\forall eq.\ \mathsf{mem}\ eq\ eqs\ \Rightarrow\ \Gamma \vdash_{\mathsf{c}} eq) \wedge \\ \quad \mathsf{dest\_eqs}\ eqs = \mathsf{Some}\ defs \wedge \\ \quad \mathsf{dest\_tm}\ defs\ t = \mathsf{Some}\ cexp \wedge \\ \quad \mathsf{interpret}\ \mathsf{init\_ck}\ defs\ cexp = \mathsf{Some}\ res\ \Rightarrow \\ \quad \Gamma \vdash_{\mathsf{c}} t =_{\mathsf{c}} \mathsf{mk\_term}\ res \end{array} \tag{5.8}$$

There are a few subtleties hidden in this theorem that we will comment on next.

First, the statement of Theorem 5.8 includes an assumption that the user-provided code equations eqs are theorems in the context Γ. The user is not in any way obliged to prove this: the fact that they can supply the compute primitive with a list of theorems means that they are valid in Candle's context at that point. Candle's soundness result allows us to discharge this assumption where Theorem 5.8 is used.

Second, the functions dest_eqs and dest_term perform sanity checks on their inputs. For example, dest_eqs checks that all right-hand sides in the equations eqs mention only constants for which there are code equations in eqs.

Third, the interpret function, which is used for the actual computation, takes a clock (sometimes called fuel parameter) in order to guarantee termination. This clock is not strictly necessary, but made it easier to use the existing CakeML

code synthesis tools. The clock is decremented by interpret on each function application (i.e. App). If the clock is exhausted, interpret returns None.

CakeML code

As with previous versions, the CakeML implementation of the computation function is synthesized from the HOL4 functions. For efficiency purposes, the generated CakeML code for interpret avoids returning an option and instead signals running out of clock using an ML exception. We note that it is very unlikely that a user has the patience to wait for a timeout since the value of init_ck is very large (maximum smallnum).

Integration

Updating the Candle proofs for Version 3 required more work than Versions 1 and 2, since we had to verify the correctness of the sanity checks performed on the user-provided list of code equations.

5.6 Efficient interpreter (Version 4)

For Version 4, we replaced the interpreter function, interpret, with compilation to a different datatype for which we have a faster interpreter.

The new datatype for representing programs is called ce. It uses de Bruijn indexing for local variables, and represents function names as indices into a vector of function bodies. Vector lookups are executed in constant time.

```
ce = Const num
   | Var num
   | Let ce ce
   | If ce ce ce
   | Monop (cval → cval) ce
   | Binop (cval → cval → cval) ce ce
   | App num (ce list)
```

Rather than representing primitive functions by names, the ce datatype represents primitive functions as (shallowly embedded) function values that can immediately be applied to the result of evaluating the argument expressions.

The interpreter for the ce datatype addresses the two main shortcomings of Version 3. First, it drops the substitution semantics in favor of de Bruijn variables and an explicit environment, so that variable substitution can be deferred until (and if) the value bound to a variable is needed. Second, all function names are replaced by an index into a vector which stores all user-provided code equations.

When updating Version 3 to Version 4, we simply replaced the following line in the implementation:

$$\mathsf{interpret\ init\_ck}\ \mathit{defs}\ \mathit{cexp}$$

with the line below, which calls the compilers compile_all and compile (these translate cexp into ce, turning variables and function names into indices) and then runs exec, which interprets the program represented in terms of ce:

$$\mathsf{exec\ init\_ck}\ []\ (\mathsf{compile\_all}\ \mathit{defs})\ (\mathsf{compile}\ \mathit{defs}\ []\ \mathit{cexp})$$

Updating the proofs for Version 4 was a routine exercise in proving the correctness of the compilers compile_all and compile. In this proof, compiler correctness is an equality: the new line computes exactly the same result as the line that it replaced (under some assumptions that are easily established in the surrounding proof). The adjustments required in the existing proofs were minimal.

5.7 Staged set up (Version 5)

At the time of writing, Version 5 is not yet implemented. However, the plan is to reduce the overhead of calling the compute function. In Versions 1–4, the user has had to provide the compute function with all of the necessary theorems at each invocation. This makes their implementation simple and stateless, but leads to duplication of work across several calls. In the stateless version, the characteristic equations need to be checked (i.e., establishing cexp_thy_ok) and the user-supplied code equations must be compiled, on each call to the compute primitive.

For Version 5, the plan is to allow a staged setup, since it suffices to perform these checks once. This version will add additional state to the Candle kernel: user-supplied code equations that have passed checks are stored persistently, and a Boolean reference tells the compute function whether cexp_thy_ok has been previously established. The Candle kernel will receive new entry points for: (1) initializing the compute primitive, i.e., establishing cexp_thy_ok and updating the Boolean flag; (2) installing new code equations; and (3) calling the compute function on a given term.

5.8 Evaluation

In this section, we report on our results comparing our new compute function to the in-logic interpreters of HOL4 and HOL Light. We tested the interpreters on three example programs in the HOL logic:

- the factorial function,

- enumeration of primes,
- generating and reversing a list of numbers.

The tests were carried out on a Intel i7-7700K 4.2GHz with 64Gb RAM running Ubuntu 20.04.

Results show that our new compute function runs several orders of magnitude faster than the derived rules of HOL4 and HOL Light, across all three examples, and all input sizes tested. In fact, it was difficult to choose input sizes large enough for us to gather meaningful measurements from our computation function, while keeping the runtimes of its derived counterparts within minutes. For this reason, we added one large data point to the end of each experiment.

Factorial The first example is a standard, non-tail-recursive factorial function, tested on inputs of various sizes. The results of the tests are shown in Table 5.1.

Table 5.1. Running times for evaluating fact n for different values of n.

n	Candle	HOL4	HOL Light
256	<1 ms	2.3 s	0.6 s
512	<1 ms	4.1 s	3.5 s
1024	<1 ms	127.5 s	17.6 s
2048	11 ms	684.7 s	86.1 s
32768	0.9 s	—	—

Prime enumeration The second example, primes_upto, enumerates all primes up to n and returns them as a list. We chose to implement the checks for primality using trial division, since it is challenging to compute division and remainder efficiently inside the logic. The results of the tests are shown in Table 5.2.

Table 5.2. Running times for evaluating primes_upto n for different values of n.

n	Candle	HOL4	HOL Light
256	<1 ms	0.5 s	1.3 s
512	<1 ms	1.6 s	5.2 s
1024	2 ms	6.3 s	20.7 s
2048	9 ms	24.2 s	83.4 s
32768	1.7 s	—	—

List reversal The third example performs repeated list reversals. The function rev_enum creates a list of the natural numbers $[1, 2, \ldots, n]$ and then calls

a tail-recursive list reverse function rev on this list 1000 times. The results of the tests are shown in Table 5.3.

Table 5.3. Average running times for evaluating rev_enum n for different values of n.

n	Candle	HOL4	HOL Light
256	0.02 s	1.1 s	66.2 s
512	0.03 s	2.3 s	250.8 s
1024	0.07 s	4.7 s	1005 s
2048	0.1 s	9.5 s	4203 s
32768	2.5 s	—	—

5.9 Related Work

In this section, we discuss related work in the area of computation in interactive theorem provers.

HOL4 Barras implemented a fast interpreter for terms in HOL4 [9], usually referred to as Eval. Eval implements an extended version of Crégut's abstract machine KN [17], and performs strong reduction of open terms, and supports user-defined datatypes and pattern-matching, and rewriting using user-supplied conversions.

Unlike our work, Eval operates directly on HOL terms. The HOL4 kernel was modified by Barras to make this as efficient as possible: the HOL4 kernel uses de Bruijn terms and explicit substitutions to ensure that Eval runs fast. However, true to LCF tradition, all interpreter steps are implemented using basic kernel inferences.

HOL Light A HOL Light port of Eval exists [58]. However, unlike HOL4, the HOL Light kernel has not been optimized for running Eval; HOL Light uses name-carrying terms without explicit substitutions, making this port comparably slow.

Isabelle/HOL Isabelle/HOL supports two mechanisms for efficient evaluation, both due to Haftmann and Nipkow. A code extraction feature [30, 31] can be used to synthesize ML and Haskell programs from closed terms, which can then be compiled and executed efficiently. We borrow the concept of code equations (Sec. 5.5) from their work, but note that Isabelle's code equations are more general than ours.

The second option is based on normalization-by-evaluation (NBE) mechanism [4] and synthesizes ad-hoc ML interpreters over an untyped lambda

calculus datatype from (possibly open) HOL terms. The ML code is compiled and executed by an ML compiler, and the resulting values are reinterpreted as HOL terms.

Both methods support a rich, higher-order, computable fragment of HOL. However, both also escape the logic, make use of unverified functions for synthesizing functional programs, and rely on unverified compilers and language runtimes for execution.

Dependent type theories Computation is an integral part of ITPs based on higher-order type theories, such as Coq [63], and Lean [19]. Their logics identify terms up to normal form and must reduce terms as part of their proof checking (i.e., type checking). Consequently, their trusted kernels must implement an interpreter.

Coq supports proof by computation using its interpreter (accessible via vm_compute), as well as native code extraction to OCaml (accessible via native_compute). Internally, Coq's interpreter implements an extended version of the ZAM machine used in the interactive mode of the OCaml compiler [28], but with added support for open terms.

A formalization of the abstract machine used in the interpreter exists [28], but the actual Coq implementation is completely unverified.

First-order logic ACL2 is an ITP for a quantifier-free first-order logic with recursive, un-typed functions. It axiomatizes a purely functional fragment of Common Lisp, which doubles as term syntax and host language for the system. As a consequence, some terms can be compiled and executed at native speed. However, this execution speed comes at a cost: no verified Lisp compiler exists that can host ACL2, its soundness critical code encompasses essentially the entire theorem prover.

5.10 Conclusion

We have added a new verified function for computation to the Candle ITP. The new computation function was developed in stages through different versions. For each version, we proved the new function only produces theorems that follow by the inference rules of HOL. In our experiments, Candle's new computation functionality produced performance numbers that are several orders of magnitude faster than in-logic evaluation mechanisms provided by mainstream HOL ITPs.

Our new compute function requires all functions that it uses to be first-order functions that perform all computations using a Lisp-inspired datatype for compute values (cval). We leave it to future work to relax this requirement.

At present, the performance numbers suggest that we do not need to go to the trouble of replacing our interpreter-based solution with a solution that com-

piles the given input to native machine code for extra performance. However, future case studies might lead us to explore such options too.

Bibliography

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

[2] Oskar Abrahamsson, Magnus O. Myreen, Ramana Kumar, and Thomas Sewell. Candle: A verified implementation of HOL Light. In June Andronick and Leonardo de Moura, editors, *Interactive Theorem Proving (ITP)*, volume 237 of *LIPIcs*, pages 3:1–3:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[3] Mark Adams. Introducing HOL Zero - (extended abstract). In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *International Congress on Mathematical Software (ICMS)*, volume 6327 of *Lecture Notes in Computer Science*, pages 142–143. Springer, 2010.

[4] Klaus Aehlig, Florian Haftmann, and Tobias Nipkow. A compiled implementation of normalisation by evaluation. *J. Funct. Program.*, 22(1):9–30, 2012.

[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 *The International Workshop on Coq for Programming Languages (CoqPL)*, 2017.

[6] Abhishek Anand, Simon Boulier, Nicolas Tabareau, and Matthieu Sozeau. Typed Template Coq–Certified Meta-Programming in Coq. In *The International Workshop on Coq for Programming Languages (CoqPL)*, pages 1–2, 2018.

[7] Abhishek Anand and Vincent Rahli. Towards a formally verified proof assistant. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving (ITP)*, volume 8558 of *LNCS*, pages 27–44. Springer, 2014.

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

[9] Bruno Barras. Programming and computing in HOL. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 1869 of *Lecture Notes in Computer Science*, pages 17–37. Springer, 2000.

[10] Bruno Barras. Sets in Coq, Coq in sets. *J. Formaliz. Reason.*, 3(1):29–48, 2010.

[11] Heiko Becker, Nikita Zyuzin, Raphael Monat, Eva Darulova, Magnus O. Myreen, and Anthony Fox. A verified certificate checker for finite-precision error bounds in Coq and HOL4. In *Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2018.

[12] Sandrine Blazy, Benoît Robillard, and Andrew W. Appel. Formal verification of coalescing graph-coloring register allocation. In Andrew D. Gordon, editor, *European Symposium on Programming (ESOP)*, volume 6012 of *Lecture Notes in Computer Science*, pages 145–164. Springer, 2010.

[13] Brandon Bohrer, Yong Kiam Tan, Stefan Mitsch, Magnus O. Myreen, and André Platzer. VeriPhy: verified controller executables from verified cyber-physical system models. In Jeffrey S. Foster and Dan Grossman, editors, *Programming Language Design and Implementation (PLDI)*, pages 617–630. ACM, 2018.

[14] Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. Imperative functional programming with Isabelle/HOL. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5170 of *Lecture Notes in Computer Science*, pages 134–149. Springer, 2008.

[15] Mario Carneiro. Specifying verified x86 software from scratch. *CoRR*, abs/1907.01283, 2019.

[16] Mario Carneiro. Metamath Zero: Designing a theorem prover prover. In Christoph Benzmüller and Bruce R. Miller, editors, *Intelligent Computer Mathematics (CICM)*, volume 12236 of *LNCS*, pages 71–88. Springer, 2020.

[17] Pierre Crégut. An abstract machine for lambda-terms normalization. In Gilles Kahn, editor, *Conference on LISP and Functional Programming (LFP)*, pages 333–340. ACM, 1990.

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

[19] Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *International Conference on Automated Deduction (CADE)*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021.

[20] Matthew Fluet. The MLton compiler webpages. Accessed: 26-Oct-2019.

[21] Anthony C. J. Fox. Directions in ISA specification. In Lennart Beringer and Amy P. Felty, editors, *Interactive Theorem Proving (ITP)*, volume 7406 of *Lecture Notes in Computer Science*, pages 338–344. Springer, 2012.

[22] Anthony C. J. Fox, Magnus O. Myreen, Yong Kiam Tan, and Ramana Kumar. Verified compilation of CakeML to multiple machine-code targets. In Yves Bertot and Viktor Vafeiadis, editors, *Certified Programs and Proofs (CPP)*, pages 125–137. ACM, 2017.

[23] Arve Gengelbach and Johannes Åman Pohjola. A verified cyclicity checker: For theories with overloaded constants. In June Andronick and Leonardo de Moura, editors, *Interactive Theorem Proving (ITP)*, volume 237 of *LIPIcs*, pages 15:1–15:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[24] Arve Gengelbach, Johannes Åman Pohjola, and Tjark Weber. Mechanisation of model-theoretic conservative extension for HOL with ad-hoc overloading. In Claudio Sacerdoti Coen and Alwen Tiu, editors, *Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP)*, volume 332 of *EPTCS*, pages 1–17, 2020.

[25] Milad Ketab Ghale, Dirk Pattinson, and Ramana Kumar. Verified certificate checking for counting votes. In Ruzica Piskac and Philipp Rümmer, editors, *Verified Software. Theories, Tools, and Experiments (VSTTE)*, volume 11294 of *Lecture Notes in Computer Science*. Springer, 2018.

[26] Michael J. C. Gordon. Introduction to the HOL system. In Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *International Workshop on the HOL Theorem Proving System and its Applications*, pages 2–3. IEEE Computer Society, 1991.

[27] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[28] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Mitchell Wand and Simon L. Peyton Jones, editors, *International Conference on Functional Programming (ICFP)*, pages 235–246. ACM, 2002.

[29] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. Verified characteristic formulae for cakeml. In Hongseok Yang, editor, *European Symposium on Programming (ESOP)*, volume 10201 of *Lecture Notes in Computer Science*, pages 584–610. Springer, 2017.

[30] Florian Haftmann. *Code generation from specifications in higher-order logic*. PhD thesis, Technical University Munich, 2009.

[31] Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming (FLOPS)*, volume 6009 of *Lecture Notes in Computer Science*, pages 103–117. Springer, 2010.

[32] Thomas C. Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason M. Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the Kepler conjecture. *CoRR*, abs/1501.02155, 2015.

[33] John Harrison. Towards self-verification of HOL Light. In Ulrich Furbach and Natarajan Shankar, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 4130 of *LNCS*. Springer, 2006.

[34] John Harrison. HOL Light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66. Springer, 2009.

[35] 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 Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 10900 of *Lecture Notes in Computer Science*, pages 646–662. Springer, 2018.

[36] Lars Hupel and Tobias Nipkow. A verified compiler from Isabelle/HOL to CakeML. In Amal Ahmed, editor, *European Symposium on Programming (ESOP)*, volume 10801 of *Lecture Notes in Computer Science*, pages 999–1026. Springer, 2018.

[37] Joe Hurd. The OpenTheory standard theory library. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods (NFM)*, volume 6617 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2011.

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

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

[40] 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.

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

[42] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *Principles of Programming Languages (POPL)*, pages 179–192. ACM, 2014.

[43] Peter Lammich. Refinement to Imperative/HOL. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving (ITP)*, volume 9236 of *Lecture Notes in Computer Science*, pages 253–269. Springer, 2015.

[44] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In Vivek Sarkar, Barbara G. Ryder, and Mary Lou Soffa, editors, *Programming Language Design and Implementation (PLDI)*, pages 24–35. ACM, 1994.

[45] 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.

[46] David Matthews. The Poly/ML compiler. Accessed: 26-Oct-2019.

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

[48] Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. Œuf: minimizing the coq extraction TCB. In June Andronick and Amy P. Felty, editors, *Certified Programs and Proofs (CPP)*, pages 172–185. ACM, 2018.

[49] Magnus O. Myreen and Jared Davis. A verified runtime for a verified theorem prover. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving (ITP)*, volume 6898 of *LNCS*, pages 265–280. Springer, 2011.

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

[51] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[52] Tobias Nipkow and Simon Roßkopf. Isabelle's metalogic: Formalization and proof checker. In André Platzer and Geoff Sutcliffe, editors, *International Conference on Automated Deduction (CADE)*, volume 12699 of *LNCS*, pages 93–110. Springer, 2021.

[53] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional big-step semantics. In Peter Thiemann, editor, *European Symposium on Programming (ESOP)*, volume 9632 of *Lecture Notes in Computer Science*, pages 589–615. Springer, 2016.

[54] Johannes Åman Pohjola and Arve Gengelbach. A mechanised semantics for HOL with ad-hoc overloading. In Elvira Albert and Laura Kovács, editors, *Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 73 of *EPiC Series in Computing*, pages 498–515. EasyChair, 2020.

[55] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Symposium on Logic in Computer Science (LICS)*, pages 55–74. IEEE Computer Society, 2002.

[56] Tom Ridge and James Margetson. A mechanically verified, sound and complete theorem prover for first order logic. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 3603 of *Lecture Notes in Computer Science*, pages 294–309. Springer, 2005.

[57] Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5170 of *LNCS*. Springer, 2008.

[58] Alexey Solovyev. HOL Light's computelib. Accessed: 2022-06-11.

[59] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Proc. ACM Program. Lang.*, 4(POPL):8:1–8:28, 2020.

[60] Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. cake_lpr: Verified propagation redundancy checking in CakeML. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2021.

[61] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *J. Funct. Program.*, 29:e2, 2019.

[62] Yong Kiam Tan, Scott Owens, and Ramana Kumar. A verified type system for CakeML. In Ralf Lämmel, editor, *Implementation and Application of Functional Programming Languages (IFL)*, pages 7:1–7:12. ACM, 2015.

[63] The Coq Development Team. The Coq reference manual. Accessed 2022-06-11. `https://coq.inria.fr/distrib/current/refman/`.

[64] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.