Candle: A Verified Implementation of HOL Light

Oskar Abrahamsson ✉
Chalmers University of Technology, Gothenburg, Sweden

Magnus O. Myreen ✉
Chalmers University of Technology, Gothenburg, Sweden

Ramana Kumar ✉
London, UK

Thomas Sewell ✉
University of Cambridge, UK

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.

2012 ACM Subject Classification Software and its engineering → Software verification

Keywords and phrases Prover soundness, Higher-order logic, Interactive theorem proving

Digital Object Identifier 10.4230/LIPIcs.ITP.2022.3

Supplementary Material *Software (Proofs and Prebuilt Binaries)*: `https://cakeml.org/candle`

Funding *Oskar Abrahamsson*: Swedish Foundation for Strategic Research.
Magnus O. Myreen: Swedish Foundation for Strategic Research.

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 [10] 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.

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.

13th International Conference on Interactive Theorem Proving (ITP 2022).
Editors: June Andronick and Leonardo de Moura; Article No. 3; pp. 3:1–3:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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. 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 [14]. Our proofs and binaries of Candle are available at: `https://cakeml.org/candle`.

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 [9]. In that work, we proved that CakeML implementations of HOL Light's kernel

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

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. 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);
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. 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.

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.

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.

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 [16] 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.

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

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

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

Figure 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 [11] as it processes these types, stating that the deep-embedded CakeML values v are refinements of the shallow-embedded HOL values: ty, tm, th.

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

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

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

Figure 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.

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 1 shows the definition of inferred, and some of the rules defining v_ok are shown in Figure 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 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 3.6), and sets up the Eval mechanism (described in Section 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 [11]. 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 [9], 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.

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. 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.

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 $ctxt$.

$$\begin{array}{l}\mathsf{ok\_event}\ (\mathsf{IO\_event}\ n\ out\ y)\ \stackrel{\mathsf{def}}{=}\\ \quad n = \mathsf{kernel\_ffi} \Rightarrow \exists\, ctxt\ th.\ \mathsf{THM}\ ctxt\ th \wedge \mathsf{thm\_to\_string}\ ctxt\ th = out\end{array}$$

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 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.

▶ **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.*

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

```
# 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 3 Output of `print_thm` applied to the theorem $\vdash \mathsf{T}$. 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 $\mathsf{T} \overset{\text{def}}{=} (\lambda\, p.\ p) = (\lambda\, p.\ p)$, which is its definition in HOL Light; equality `=` is a constant of type $\alpha \to \alpha \to \mathsf{bool}$; types `bool` and `fun` are defined. Some excess output is elided (...) above.

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.

4.1 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

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

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.

4.2 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. This section attempts to explain the code shown in Figure 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 $res \neq$ 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 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];;
```

As can be seen on line 3 in Figure 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. Running 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.

4.3 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 from the proofs about stepping through the operational semantics. It is worth noting that the REPL implementation uses the type

```
# 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 5 Sample interaction with the Candle REPL in the OCaml syntax of HOL Light.

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*.*

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

This theorem is proved by rewriting and evaluation. It is used in Section 5.

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 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 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 5 also shows our quote filter in action: it processes the quoted terms `` `...` `` correctly.

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*.*

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

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*).*

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

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.

$$\begin{array}{l}
\vdash e \in \mathsf{events\_of}\ res_1 \land res_1 \in sem_1 \land sem_1 \subseteq \mathsf{extend\_with\_resource\_limit}\ sem_2 \Rightarrow \\
\quad \exists\, res_2.\ e \in \mathsf{events\_of}\ res_2 \land res_2 \in sem_2
\end{array}$$

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 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*.*

$$\begin{array}{l}
\vdash res \in \mathsf{machine\_sem}\ (\mathsf{basis\_ffi}\ cl\ fs)\ ms \land \mathsf{repl\_ready\_to\_run}\ cl\ fs\ ms \Rightarrow \\
\quad res \neq \mathsf{Fail} \land \forall\, e.\ e \in \mathsf{events\_of}\ res \Rightarrow \mathsf{ok\_event}\ e
\end{array}$$

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.

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 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.

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:

```
# 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 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.

- 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.
- 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`.

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). 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`.

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.

7.1 Higher-order logic

Harrison [8] 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. [9]. They build on Harrison's work and expand it along several dimensions; they formalise the definitional mechanisms omitted by Harrison [8] 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 [13, 7] further extend the work by Kumar et al. [9], 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 [6]. We see no reason that our work should not build on their kernel implementation in the future.

Nipkow and Roßkopf [12] 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.

7.2 First-order logic

The most comprehensive ITP verification result prior to ours is Milawa by Davis and Myreen [5]. 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 [10] 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 [4] 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 [3]. 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 [3]. 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.

7.3 Dependent type theory

Barras [2] 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 [8], by defining a set-theoretic model of the calculus, and proving its soundness wholly inside Coq itself.

Sozeau et al. [15] 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 [8], Kumar et al. [9] and Barras [2], 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 [1] 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.

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.

References

1 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*. Springer, 2014. doi:10.1007/978-3-319-08970-6_3.

2 Bruno Barras. Sets in Coq, Coq in sets. *J. Formaliz. Reason.*, 3(1), 2010. doi:10.6092/issn.1972-5787/1695.

3 Mario Carneiro. Specifying verified x86 software from scratch. *CoRR*, abs/1907.01283, 2019. arXiv:1907.01283.

4 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*. Springer, 2020. doi:10.1007/978-3-030-53518-6_5.

5 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. doi:10.1007/s10817-015-9324-6.

6 Arve Gengelbach and Johannes Åman Pohjola. A verified cyclicity checker. In *Interactive Theorem Proving (ITP)*. LIPIcs, 2022.

7 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*, 2020. doi:10.4204/EPTCS.332.1.

8 John Harrison. Towards self-verification of HOL Light. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning (IJCAR)*, volume 4130 of *LNCS*. Springer, 2006. doi:10.1007/11814771_17.

9 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. doi:10.1007/s10817-015-9357-x.

10 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*. Springer, 2011. doi:10.1007/978-3-642-22863-6_20.

11 Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *J. Funct. Program.*, 24(2-3), 2014. doi:10.1017/S0956796813000282.

12 Tobias Nipkow and Simon Roßkopf. Isabelle's metalogic: Formalization and proof checker. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction (CADE)*, volume 12699 of *LNCS*. Springer, 2021. doi:10.1007/978-3-030-79876-5_6.

13 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*. EasyChair, 2020. doi:10.29007/413d.

14 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. doi:10.1007/978-3-540-71067-7_6.

15 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), 2020. doi:10.1145/3371076.

16 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)*. ACM, 2015. doi:10.1145/2897336.2897344.