Cakes That Bake Cakes: Dynamic Computation in CakeML

Cakes That Bake Cakes: Dynamic Computation in CakeML

THOMAS SEWELL, University of Cambridge, United Kingdom
MAGNUS O. MYREEN, Chalmers University of Technology, Sweden
YONG KIAM TAN, Unaffiliated, Singapore
RAMANA KUMAR, Unaffiliated, United Kingdom
ALEXANDER MIHAJLOVIC, Chalmers University of Technology, Sweden
OSKAR ABRAHAMSSON, Chalmers University of Technology, Sweden
SCOTT OWENS, Unaffiliated, United Kingdom

We have extended the verified CakeML compiler with a new language primitive, Eval, which permits evaluation of new CakeML syntax at runtime. This new implementation supports an ambitious form of compilation at runtime and dynamic execution, where the original and dynamically added code can share (higher-order) values and recursively call each other. This is, to our knowledge, the first verified run-time environment capable of supporting a standard LCF-style theorem prover design.

Modifying the modern CakeML compiler pipeline and proofs to support a dynamic computation semantics was an extensive project. We review the design decisions, proof techniques, and proof engineering lessons from the project, and highlight some unexpected complications.

CCS Concepts: • **Software and its engineering → Software verification**; *Compilers*; • **Theory of computation** → Higher order logic.

Additional Key Words and Phrases: compiler verification, dynamic computation, interactive theorem proving

ACM Reference Format:
Thomas Sewell, Magnus O. Myreen, Yong Kiam Tan, Ramana Kumar, Alexander Mihajlovic, Oskar Abrahamsson, and Scott Owens. 2023. Cakes That Bake Cakes: Dynamic Computation in CakeML. *Proc. ACM Program. Lang.* 7, PLDI, Article 152 (June 2023), 24 pages. https://doi.org/10.1145/3591266

1 INTRODUCTION

The CakeML project [Myreen 2021] consists of a formally specified language (CakeML) in the ML family, and a suite of formally verified tools for that language. A key part of the project is the CakeML compiler [Kumar et al. 2014; Tan et al. 2019], which is verified to correctly compile CakeML source programs down to various machine-code targets. Notably, CakeML supports *verified bootstrapping*, i.e., verified compilation of the CakeML compiler's own implementation in CakeML.

The `Eval` primitive operator, which we have added to the language, enables a CakeML program to dynamically compile and execute new CakeML syntax created at runtime. This *dynamic computation* mechanism can be used to write a REPL (a **r**ead-**e**valuate-**p**rint **l**oop) as a CakeML program. Providing such a REPL is a welcome addition to the features available in the CakeML project. However, the main goal from our perspective was to build Candle [Abrahamsson et al. 2022], a

Authors' addresses: Thomas Sewell, University of Cambridge, United Kingdom, tals4@cam.ac.uk; Magnus O. Myreen, Chalmers University of Technology, Sweden, myreen@chalmers.se; Yong Kiam Tan, Unaffiliated, Singapore, yongkiat@alumni.cmu.edu; Ramana Kumar, Unaffiliated, United Kingdom, ramana@member.fsf.org; Alexander Mihajlovic, Chalmers University of Technology, Sweden; Oskar Abrahamsson, Chalmers University of Technology, Sweden, oskar8192@gmail.com; Scott Owens, Unaffiliated, United Kingdom.

2475-1421/2023/6-ART152
https://doi.org/10.1145/3591266

```
(* -- Kernel signature -- *)
val reflexivity : term -> thm;  val transitivity : thm -> thm -> thm;

(* -- Library signature -- *)
type conv = term -> thm;
val ALL_CONV : conv;  val THEN_CONV : conv -> conv -> conv;

(* -- Proof Script -- *)
val conv_demo = (THEN_CONV conv1 conv2) '''x < y AND (y < z OR x > z)''';
fun xy_nf_conv tm = if can_lift_step tm
  then (THEN_CONV xy_lift_binder_step xy_nf_conv) tm
  else if can_flatten_step tm
  then (THEN_CONV xy_flatten_step xy_nf_conv) tm
  else ALL_CONV tm;
```

Fig. 1. Example code, reducing to the fictional "X-Y" normal form, in the LCF prover style

formally verified interactive theorem prover—the existence of a dynamic computation mechanism and a REPL-like interface are necessary for supporting interactive proofs.

Prior to this work, the CakeML compiler had developed into an ahead-of-time compiler with many optimisation passes and several intermediate languages (an earlier version had supported a REPL, to which we will compare later). The question which motivates this project is, how can we convert an ahead-of-time self-hosting compiler into a dynamic computation environment? Intuitively, we hope to simply rewrite `Eval` as follows, by including the compiler at runtime:

dynamic computation — compilation at runtime — dynamic loading and exec.

```
Eval ctxt prog ⟹ let val binary = compile ctxt prog in Execute binary end
```

The actual process of calling the compiler from a REPL is more complex than the above snippet suggests. However, it does illustrate the substantial questions we must answer. In a verified compiler which assigns semantics to every intermediate language, what does it mean to execute new machine code via `Execute`? What implications do this, and enabling compilation at runtime, have for the substantial existing body of compiler proofs?

We have developed a proof approach that addresses these questions, and gone on to update the semantics and compiler proofs. This evolved into a long-running project; one that passed between various contributors; and one that concludes with a version of CakeML that supports `Eval`, can compile a proven-safe REPL, and can compile a proven-correct interactive theorem prover.

1.1 A Verified LCF Successor

Why put this much effort into evolving CakeML from an ahead-of-time compiler to one with a dynamic computation capacity? CakeML is a language from the ML-family, a successor of the first ML "meta-language" [Gordon et al. 1978] which was co-developed with the LCF interactive theorem prover [Gordon et al. 1979]. LCF introduced a novel theorem prover design in which the proof kernel exposes logical operations on an opaque `thm` type, but not the constructor(s) for that type. Library code builds those base operations into more general mechanisms. User proof scripts, developed interactively, can also add custom functions that combine kernel and library features.

For instance, syntax rewriting passes can be expressed using the "conversion" mechanism [Paulson 1983] provided in various proof libraries. A conversion is a library function `f` such that `f`

`x` always returns a theorem of the form `⊢ x = y`. Example conversion code is sketched in Fig. 1, including a default conversion `ALL_CONV` which always returns the reflexivity theorem `⊢ x = x` constructed by a kernel operation. It also provides the essential `THEN_CONV` combinator, which combines a conversion that produces `⊢ x = y` and one that produces `⊢ y = z` into one that produces `⊢ x = z` using the kernel's transitivity operation. Proof scripts can build custom libraries of conversions, which can call and even recurse to themselves via combinators such as `THEN_CONV`.

Although all functions can pass and return `thm` objects, a key element of the LCF theorem prover design is to use the type system to enforce that only kernel operations can construct objects with `thm` type. This requires a type-safe language such as ML and an implementation where the user can conveniently supply experimental code outside the prover kernel. Thanks to the LCF design the system stays sound regardless of what code the user inputs.

Interactive theorem provers are still a major use-case for ML-influenced languages [Harrison 2009; Martínez et al. 2019; Nipkow et al. 2002; Slind and Norrish 2008], which makes them an obvious target for CakeML; in fact, CakeML is itself developed and verified in HOL4 [Slind and Norrish 2008], a prover in the LCF family. Building an LCF-style interactive theorem prover was one of the original goals of the CakeML project, and the verified Candle prover [Abrahamsson et al. 2022] completes that goal, building on the `Eval`-extended CakeML environment. The Candle use-case motivates some of the ambitious aspects of our `Eval` mechanism design, and we elaborate on the corresponding challenges here. In particular, we want dynamically added proof functions to interact in a higher-order and type-safe way with functions compiled in the prover core. There are other ways to achieve reasonable theorem prover performance, but this style is closely compatible with the broader CakeML approach. We elaborate on alternative and related approaches in Section 8.

1.2 Contributions

We have added the `Eval` mechanism to the CakeML system. This gives a verified language environment with a dynamic computation feature that permits a REPL implementation to be written as a source program, and which permits higher-order functional interaction between the original and dynamically added code. It features, to our knowledge, the first verified run-time environment capable of supporting a standard LCF-style theorem prover design, and also the first verified dynamically-executed compiler for a functional language that supports substantial optimisations.

Our run-time implementation is relatively straightforward: given that CakeML is self-hosting down to machine code, in principle, one needs to run the compiler dynamically, write its output instructions to code memory, and then jump to the relevant instructions. The key challenge lies in updating the language semantics and compiler proofs to support dynamic computation.

We introduce an oracle-based method for extending compiler verification to support dynamic computation, apply it to a large existing proof, and report on the aspects of this effort that were particularly challenging. As we will illustrate, for the lower-level compiler passes, the main effort is focused on the actual compilation of `Eval` to a mechanism that adds new code to memory, whereas for the higher-level intermediate representations, the challenge is to handle new complexity in the simulation proofs, especially knot-tying the CakeML source semantics based on `Eval` to the oracle-based semantics. Our approach has some important limitations:

- `Eval` is currently only supported on a small subset of CakeML target architectures (x64) and operating systems (Linux-based). Nevertheless, we expect that extending `Eval` to more of CakeML's targets is straightforward.
- Code added dynamically is not reclaimed, which is potentially an issue for very long-running REPL environments. Nevertheless, CakeML programs always halt with an out-of-memory error when they run out of heap, stack, or code memory.

```
fun l_to_mc p = let
    val p1 = l_to_l1 p;
    val p2 = l1_to_l2 p1;
    val (code, data) =
        l2_to_mc p2;
  in (code, data) end
```

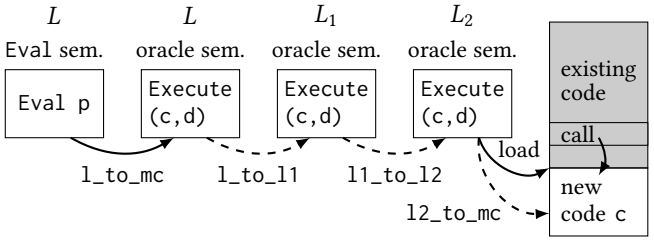

Fig. 2. (left) toy example compiler (right) intermediate languages, type of semantics, and compilation/execution steps using the toy compiler at runtime for `Eval`

- Making newly inserted code safe to execute requires special operations on some architectures (e.g., flushing instruction caches) and operating systems (e.g., requesting write and execute permission to code memory). These operations are implemented for the supported targets, but permissions and caches are not modeled in CakeML's target memory model, so their implementations are not verified.
- The correctness proof for the REPL binary must, in addition to assuming its code and data are loaded correctly, assume that it correctly reads additional data from the filesystem.

2 OVERVIEW OF THE APPROACH

The CakeML compiler function compiles CakeML abstract syntax into its output, which consists of machine code and some constant data, each of which is essentially a block of bytes to be placed at a particular address in memory. The key question for `Eval`, both practically and conceptually, is to give a "meaning" or semantics to this new binary program.

In some sense, the CPU architecture specifies precisely what a binary program means. This is a good starting point, but we must address two complications. Firstly, while the binary program is well-defined in the concrete machine universe, it is less clear what operations might mean in terms of the abstractions that exist at the source level, or in any other intermediate language. Secondly, while a self-contained binary might be well-defined, we want the dynamically added code to be able to call into the original, which means its semantics is tied to that of the original program.

2.1 Oracles and Checking Compilers

CakeML, like most compilers, operates as a pipeline of translation phases, each producing and consuming a program. There are (currently) 8 different intermediate languages (henceforth ILs) in which these programs are represented. The main issue with binary semantics appears in the inner ILs. To illustrate, let us consider a simplified example.

Consider a toy academic language L, which is compiled by the program `l_to_mc` in Fig. 2 (left). It is compiled through three passes and two ILs (named L_1 and L_2) into output machine code (and data). Without going into details, let us assume that L, like CakeML, is a sufficiently high-level language so that we can express `l_to_mc` in L. Thus, we can try to extend L with dynamic computation by adding `Eval` to the language, and compiling `Eval p` to `Execute (l_to_mc p)`. Figure 2 (right) gives an overview of the compilation steps at runtime; solid arrows in the figure represent the real compilation and loading steps that are executed at runtime, while the dashed arrows represent compilation steps that take place in the compiler oracle (explained next).

The first challenge in proving correctness of dynamic computation is to express the semantics of `Execute` in the languages L, L_1 and L_2: given machine code, we need to know which program (in the relevant intermediate language) is being evaluated. Computing an inverse to the compiler is not practical. One possibility is to keep the input program together with the compiled program, i.e., adjust `l_to_mc` to return both the output machine code and the input program, have `Execute` receive both representations, and have the semantics of `Execute` for output machine code be essentially `Eval` of the input program. To extend this to L_1 and L_2, `l_to_mc` needs to be adjusted further to return a collection of intermediate programs, which is a little unwieldly. The bigger problem is, while L is a high-level language, L_1 and L_2 may not be. The kind of datatype that conveniently expresses an input abstract syntax tree in L might be represented as a low-level encoded object in L_2. Following this concrete approach requires us to not only save every intermediate representation but also construct a decoder to abstract values for each language.

For CakeML, this would be even more complex: the current compiler has over 30 passes, so keeping every intermediate representation would be inefficient. Moreover, the existing simulation proofs done for each pass do not expose a value relation or decoder as part of their external interface. To follow this approach, we would have to implement some kind of decoder at every representation level and prove that possible decode actions are preserved by each compilation phase.

Instead, we pass the program being evaluated to `Execute` implicitly, i.e., we make the program available in the formal semantics but not the concrete program execution. The CakeML semantics is already organised to be deterministic and trace-based. Input and output operations such as reading the filesystem are permitted via the foreign-function interface (FFI). To make this deterministic, the semantics includes an oracle object that knows how the FFI will respond to any sequence of FFI calls. For the purposes of the proof, we introduce a new oracle object that knows the future trace of `Eval` events. In our toy example, we add these oracles to the evaluation state in the semantics of L_1 and L_2, and to a similar variant of the semantics of L. When we compile `Eval p` to `Execute (l_to_mc p)`, we also switch to the oracle semantics for L, using the determinism of L to peek ahead and record the trace of `Eval` events.

The semantics of `Execute`, in all the oracle semantics, are those sketched in Fig. 3: fetch the program from the oracle (`pop_oracle`), check the machine code being executed is the compilation of that program, and proceed to evaluate the semantics of the oracle-provided program. This also requires an abstract implementation of the compiler (for a given compilation phase) kept as part of the semantic state (`get_compiler`). We call this a checking compiler, used only for checking concrete machine code values against oracle-provided programs.

```
⟦ Execute (code, data) ⟧ ≡
 let
   val prog = pop_oracle();
   val compiler = get_compiler();
   val (c2, d2) = compiler prog;
   assert (code == c2);
   assert (data == d2);
 in
    ⟦ prog ⟧
 end
```

Fig. 3. Pseudocode Execute Semantics

This approach does not require L_2 to be able to encode L_2-programs as values; only concrete machine code values (essentially lists of bytes) need to be present in each language. Such a simple low-level type can be easily encoded in every IL (in this toy example or in CakeML), either as list values or as regions of bytes in memory. It also does not require saving every program representation, instead, each phase has its own oracle. The L_2 oracle, for instance, is constructed from the L_1 oracle by composing it with `l1_to_l2`, as shown by the dashed arrow in Fig. 2 (right). The checking compiler for L_1 can be produced by composing `l1_to_l2` with the checking compiler for L_2. The fact that the relevant oracles and checking compilers are connected via `l1_to_l2` is used in the simulation proof for `l1_to_l2`, when it compiles `Execute (code,data)` in L_1 to `Execute (code,data)` in L_2. More precisely, the inductive correctness proof

for `l1_to_l2` has a new case for correct compilation of `Execute (code,data)` from L_1 to L_2. This case holds by induction (on the evaluation semantics) because the L_2 oracle program is equal to `l1_to_l2` of the corresponding L_1 oracle program.

The proof strategy based on oracles and checking compilers was prototyped for a toy language much like L above, as the first step in the `Eval` project.

2.2 Compiling Binary Extensions

Let us clarify how the additional machine code produced by compilation at runtime relates to the existing machine code. We will use the following terminology: the *ahead-of-time* compiler refers to the static compiler that compiles programs to a static binary; the *in-program* compiler refers to the compiler that is dynamically called at runtime for `Eval`. These two compilers should be largely similar, with the notable exception that the ahead-of-time compiler will perform introduce library code that is needed at runtime, which the in-program compiler reuses.

Recall that we want newly compiled `Eval`-ed programs to be able to call functions and use values from the original program (or previous calls to `Eval`). CakeML does not support partial compilation and linking. If it did, we would presumably re-use that mechanism and the focus of the `Eval` project would be on making the linking mechanism work at runtime. Instead, we focus on an alternative design in which the new machine code works as an addition to what is already compiled. It will actually be dynamically loaded into memory directly after the existing code, as in Fig. 2 (right).

To illustrate the idea, note that many compiler phases need to allocate identifiers (IDs) in some global namespace. For instance, each new exception constructor is given a new unique ID, and many of the ILs have a global code table, with IDs for each function. The start addresses assigned to basic blocks of machine code by the final assembler are also a kind of global ID. In each of these cases, compilation of new code proceeds with the final ID allocation state from before, and issues new IDs in each namespace that follow those of the previous compilation unit. This is also why new code produced by the in-program compiler will be placed shortly after the existing code, because the assembler allocates a start address at that position.

For each compiler phase that allocates IDs in a new namespace, we modify the in-program compiler to maintain a mapping to those IDs, e.g., a map from the string identifying an exception constructor to its ID, or a map from the numeric ID of a function in the final code table to its start address. In this way, in-program compilation of new code will be able to look up and reference IDs from prior compilations. This applies to every namespace including final code addresses so newly compiled code can jump to known addresses of previously compiled code entries. The ahead-of-time compiler is responsible for producing the initial mapping passed to the in-program compiler.

Intuitively, the run-time (and proof) invariant here is that, at each `Eval` call site, the existing program is a valid self-contained program, and the `Eval`-extended program is also valid because, while its components were compiled at different times, they fit together as a self-contained program, as though they had all been compiled together. At each step, the machine code is self-contained, so there will never be a direct jump from compiled code into code that an `Eval` has yet to add. Nevertheless, function values can be passed in various ways, e.g., via references, which makes indirect jumps from older to newer code possible. In practice, the in-program compiler maintains more than just IDs. For example, it may also have an inlining pass which stores the bodies of some short functions. Like the ID mappings, this information is kept available by the in-program compiler, so that older function bodies may be inlined into newly `Eval`-ed function bodies.

2.3 Compiler State

The toy oracle design from Section 2.1 is elegant for a pure compiler, but unfortunately some CakeML compiler phases are not stateless pure functions. The stateful phases accumulate a number

```
do_eval vs es = case (es, vs) of
  | (SOME (EvalDecs s), [Env env id; st_v; decs_v; st_v2; bs_v; ws_v]) =>
    (case s.decode_decs decs_v of
       NONE => NONE
     | SOME decs =>
       if st_v = s.compiler_state && concrete_v decs_v &&
         compiler_agrees s.compiler (id,st_v,decs) (st_v2,bs_v,ws_v)
       then
         SOME (env, decs, SOME (EvalDecs
                 (add_decs_generation (s with compiler_state := st_v2))))
       else NONE)
  | ...
```

Fig. 4. A snippet from the Eval checking process

of bits of state, including the ID mapping or inlining tables described above. These state elements complicate aspects of the oracle design.

The states must also appear in the oracles in order for the checking compiler to produce exactly the same machine code as the in-program compiler. Producing the various oracles with the compiler state at each `Eval` point is not much more complex than capturing only the programs. However, various proofs of simulation make use of the compiler state in relating expressions before and after compilation. The future compiler states encoded in the oracle may now also need to be considered in these aspects of the proof, which is a substantial source of new proof complexity.

2.4 The First Checking Compiler

Compilation of the toy language L starts with a source-to-source pass that replaces `Eval` with `Execute (l_to_mc p)`. This requires the compiler to know its own text and insert it into the source language. This provides an elegant external interface to `Eval`, and works out nicely in the toy example. Unfortunately, in actual CakeML, this creates unnecessary complications. The program text of the CakeML compiler includes the definition of the compiler function as well as the definitions of its internal types, the types needed to define its input syntax, and also the text of the standard library. Stitching this into the target program in a way that avoids duplicating the standard library and shares the type definition of the input would be tricky to get right. In addition, the CakeML compiler text, as CakeML syntax, is generated from HOL4 rather than written by hand, which limits our ability to refactor it for this purpose.

The CakeML compiler is also permitted to fail in some unlikely situations. For instance, when assigning final addresses to basic machine-code blocks, it is possible that the jump distances cannot be encoded as valid offsets, resulting in a failure of the whole compilation. Some attempt should, in future, be made to handle such failures.

Instead of including the compiler automatically, we require users of the `Eval` mechanism to themselves include the compiler as a dependency of their code in the conventional way. In practice, REPLs and other `Eval`-using programs will be built above the CakeML text extracted from HOL4, likely using the same mechanism. To use `Eval`, these programs must call the compiler themselves, and handle any failure outcomes. The actual `Eval` primitive is then passed the source program and binary output from this call to the compiler. The primitive checks that this input/output pair agrees with what it expects the compiler to do. That is, the actual `Eval` primitive in CakeML implements the first checking compiler, prior to the appearance of any oracles.

```
evaluate : state -> env -> exp list -> (state * (v list) exc)
evaluate_decs : state -> env -> dec list -> (state * env exc)

type compiler_args = ((num * num) * v * dec list)
type compiler_fun = compiler_args -> (v * word8 list * word64 list) option

datatype eval_decs_state = {| compiler : compiler_fun ; compiler_state : v ;
  env_id_counter : (num * num * num) ; decode_decs : v ->  (dec list) option |}

datatype eval_state = EvalDecs eval_decs_state | EvalOracle eval_oracle_state
```

Fig. 5. HOL4 types of key semantics elements

Figure 4 shows a part of the `Eval` compiler check done in the CakeML source semantics specification. Here, `do_eval` checks that it can interpret its concrete inputs and outputs and that they agree with the result of the checking compiler (`s.compiler`) stored in the state. We will explain more details of this code snippet in Section 3.

The final step is to define a REPL that calls the CakeML compiler and then uses the `Eval` primitive, and prove that the resulting REPL is a safe program, i.e., that the compiler check always succeeds. This has been done for two flavours of REPL for CakeML: a simple REPL which processes CakeML source, and a variant of this needed by the Candle prover [Abrahamsson et al. 2022].

3 UPDATING THE CAKEML LANGUAGE SPECIFICATION

The CakeML language is given a formal semantics in HOL4 which specifies the meaning of CakeML programs and, consequently, the required behaviour of the compiler. The type signatures (in HOL4) of the evaluation functions which define the semantics of expressions (`evaluate`) and of declarations (`evaluate_decs`) are given in Fig. 5. These are recursive functions rather than relations as the CakeML semantics is defined in a so-called functional big-step style [Owens et al. 2016]; their recursion is terminating, with a decreasing clock counter within the state record, and a timeout exception being one of the possible special return values within the `'a exc` type.

A CakeML program consists of a list of declarations (`dec list`). The Eval project adds three new primitives to the language that manipulate declarations, two new operators `Eval` and `Env_id`, and a new primitive declaration `Denv`. This section describes their semantics.

To support `Eval`, a new field `st.eval_state : eval_state option` is added to the `state` type. The contents of the `Eval`-enabling variant of this type is shown in Fig. 5. The state can also be left as `None` to disable `Eval`, in which case all uses of `Eval` raise errors, or configured to an alternative oracle-based semantics used in a proof phase (we will discuss that phase in Section 5.1).

The in-program compiler can be wrapped into the `compiler_fun` type shown in Fig. 5. It consumes a CakeML program (`dec list`) and produces machine code and extra data (`word8 list` and `word64 list`). It also manages a state, which is left as an arbitrary uninterpreted CakeML value (`v`). The remaining argument is an environment ID.

3.1 Compiling Environment Objects to IDs

Evaluating either expressions or declarations in CakeML requires an environment (see Fig. 5). These environments are associated with IDs, and the operator `Env_id : env -> (num * num)` can expose those IDs. Environment objects are common in the CakeML semantics, mapping names (including compound names like `List.map`) to values. They are used to create closures, to capture

local let-bindings, and to capture the contents of the global namespace and the module namespaces within it. Mechanisms like modules and local blocks mostly amount to operations on (global) environments. The `Eval` operator also requires an environment parameter.

The `Denv` declaration exists so that a user program can capture a global environment as a variable and pass it to a subsequent `Eval` operation. It is provided as a declaration form (e.g. `Denv env_name`) rather than an operation (e.g. in `val env_name = Env ()`) so it only captures global environments.

The compilation scheme replaces environments with their IDs. For instance, IDs rather than environments appear in the `compiler_args` argument to the in-program compiler. These IDs are used because, both before and after the Eval project, there are no run-time values in the CakeML system which correspond to environment values from the source semantics. No run-time object maps source names to run-time values. An early phase of the compiler, named `source_to_flat`, lifts all global objects to a global scope with unique numeric identifiers. This eliminates modules and other namespace operations, and eliminates the need for global environment values. The local environment values used in closures are eliminated later in the closure-compilation phases.

The `source_to_flat` compiler does manage a "globalising" mapping from source names to global IDs. This is not a mapping from source names to anything meaningful at runtime: the global namespace in the `source_to_flat` output is renumbered multiple times by subsequent passes in the compiler pipeline. This design would probably have to change if CakeML were to support partial compilation or linking. We keep the design, but have the `source_to_flat` pass keep one more table in its state, an environment table that maps environment IDs to past states of its globalising mapping. A `Denv` declaration is compiled by storing the current globalising mapping in the environment table with a new environment ID, and compiling the `Denv` declaration as though it simply declared that ID as a global value. When an `Eval` occurs, only the ID from the environment is passed to the compiler, and it is used to index the environment table and to start the initial `source_to_flat` compilation with the correct mapping.

Since the `source_to_flat` phase is compiling environments into their IDs, it can compile `Env_id` into a no-op. The `evaluate` function of the source semantics handles `Eval` via the `do_eval` function seen previously in Fig. 4. It only uses the ID part of the environment argument to in the checking compiler step, but the full environment is used to evaluate the dynamic program. The correctness proof of `source_to_flat` must establish that the globalising map fetched via that ID agrees with the global environment from the value.

The `Eval` operation also returns the environment value after evaluation. The `source_to_flat` pass allocates a global reference for this purpose. It adds code snippet to the end of every `Eval`-ed program to write the ID to the reference and another snippet at the `Eval` site to read it again.

The IDs of environments are integer tuples rather than just integers, as is visible the various types in Fig. 5. Tuples solve an issue with ordering: the IDs are issued by both the `source_to_flat` compiler phase and the source semantics, and must match. However, the compiler and operational semantics pass over declarations in a different order. The implementation compiles the whole program supplied to `Eval` before executing any machine code, whereas the semantics works one declaration at a time.

For example, consider the program sketched in Fig. 6. The inner program `prog1` is run via `Eval`, and it then runs `prog2` via `Eval` also. Assume that `eval` takes care of calling the compiler and using `Eval` correctly (the actual REPL contains such a function). In source-semantics

```
val prog1 = parse '''
    val env4 = eval env1 prog2;
    val _ = print "goodbye"; ''';

val prog2 = parse '''
    Denv inner_env;
    val _ = print "hello"; ''';

Denv env1;
val env2 = eval env1 prog1;
Denv env3;
```

Fig. 6. Example Nested Eval

order, `prog1` and `prog2` complete before `Denv env3` is seen, and `env2` actually contains `env4`. However, the compiler has completed compiling the entire program, including assigning an ID to `env3`, before any code runs and before any `Eval` events. The IDs are issued with `prog1` being generation 1 and `prog2` being generation 2 (assuming no prior `Eval`). The IDs in generation 2 are $(2, 0)$ for `inner_env` and then $(2, 1)$ for the final environment of `prog2`, which is then named `env4` by `prog1`. The only member of generation 1 is the final environment of `prog1`, $(1, 0)$, which is then named `env2`. The IDs of `env1` and `env3` are $(0, 0)$ and $(0, 1)$.

The source semantics state tracks the integer triple `(cur_gen, next_id, next_gen)` in the field `env_id_counter`. `Eval` assigns a new generation as it starts and restores `cur_gen` and `next_id` as it returns. This keeps the compiler-issued and semantics-issued IDs in sync.

3.2 Providing the Abstract Compiler

The source semantics includes a compiler function in the (optional) `eval_state` component. To use `Eval` safely, this must be set to some function of the correct type, and a decoder from CakeML values to AST must also be provided. The approximate type of `Eval` is:

```
Eval : env -> 'st -> 'decs -> 'st -> word8 list -> word64 list -> env
```

The `Eval` arguments include the three input arguments and three output results of the compiler function. The semantics assert that the input/output values given agree with the abstract compiler (most of the code for this check was seen in Fig. 4). The approximation in `Eval`'s type is because the semantics do not specify the encoding of compiler states or the source syntax as CakeML values. Also, the CakeML type checker does not respect the above type, and rejects any program including `Eval`. No simple type check can ensure that the code that `Eval` runs is safe in any way, and it is essential that type-checked code always executes safely.

The type of the compiler state is not entirely arbitrary. The compiler correctness theorem requires that, if `Eval` is enabled, the abstract compiler in the state must be exactly the in-program CakeML compiler, wrapped with appropriate encoding and decoding functions. These encoding and decoding values are produced as a byproduct of HOL4-to-CakeML translation [Myreen and Owens 2014]. This translation is a standard part of the CakeML process. Instead of writing the CakeML compiler as a program in CakeML or some other language, it is defined as a terminating function in the logic of HOL4. This so-called *shallow embedding* is convenient for verification. It is a similar approach to CompCert [Leroy 2009a], with the compiler defined and verified in the logic prior to being extracted to an executable version. In CakeML, the extraction from a shallow embedding to executable CakeML is done by a proof-producing tool rather than a trusted one.

For each translated type, the translator defines a relation from that type to the general values in the semantics (the `v` type in Fig. 5). We have extended it to also provide encode and decode functions and proofs of their injectivity when that is possible. The compiler correctness proof requires injectivity of the encode/decode functions for aspects of the oracle construction to work.

Unfortunately this injective encoding is not possible for function types, or for types which contain function types. The CakeML compiler state does include some function-typed fields in the configuration of various features. We work around this by adding explicit projections to and from a simplified state, where the simplified state has no problematic fields, and the restored state replaces the function parts with defaults. Finally, we show that this restored configuration is sufficiently compatible with the original that the necessary proofs hold.

4 UPDATING THE COMPILER PIPELINE

The modern CakeML compiler consists of a pipeline of over 30 different compiler phases which operate on 8 different ILs. Figure 7 sketches the pipeline and highlights the regions we will focus

on here. An up-to-date figure showing the compiler pipeline is available at https://cakeml.org/ and an in-depth description of the pipeline is in Tan et al. [2019]. There are slight differences between the CakeML ahead-of-time compiler and the in-program compiler (recall Section 2.2). The ahead-of-time compiler introduces additional code, such as CakeML's verified bignum library or garbage collector that is used by the runtime. The in-program compiler is configured for reduced compilation time, e.g., it uses linear scan register allocation instead of (more expensive) graph colouring-based allocation. We elide discussion of these differences below.

Most of the compiler phases require no adjustment for `Eval` at all, besides their correctness proofs. In some cases the expression languages of the adjacent ILs are different but have similar `Eval` primitives, so the compiler needs to be adjusted to translate the operator named `Eval` in one language into the similarly named `Eval` operator in the next.

4.1 Compiling Eval in High-Level Intermediate Languages

Let us detail the cases where `Eval` is (non-trivially) modified. We have explained that the source `Eval` takes six arguments, two of which are the code and data that came from compiling the provided program, and the remaining four are needed for knot-tying `Eval` to the oracle semantics. The translation to the first IL, `source_to_flat`, discards the four knot-tying arguments, so `Eval` from `FlatLang` onwards operates only on compiled output and more closely resembles `Execute` in our toy description (Section 2.1). This two-argument `Eval` passes through the early compiler phases syntactically unchanged until the point of closure conversion. One important exception is an early compiler phase which performs global dead-code elimination—when `Eval` is present in the code, this pass is disabled entirely, as functions unused in a given ahead-of-time (or in-program) compilation might still be needed in a subsequent `Eval` step.

Closure conversion is a substantial task for the CakeML compiler, and involves multiple passes. Prior to this point, values of function type can be created dynamically, by function expressions (e.g. `val f = fn x => fn y => x + y + z`) or by partial application (e.g. `val g = f 12`). The closure conversion process lifts all functions into a global code table with numerical IDs. These expressions then create a closure object, one which records the ID of the function to be called and a flat vector containing the captured values to be given as arguments (e.g., the value `z` captured by `f` above, or `12` captured by `g` as the value of `x` in the partially applied body of `f`).

The closure-conversion process thus reduces a higher-order functional language to a first-order one. It also changes the calling style, with multi-argument function calls. It is essential for performance that this compilation be done carefully, avoiding the creation of closure objects wherever direct function calls would be possible instead. This is accomplished via numerous passes over the CakeML compiler's `ClosLang` IL [Owens et al. 2017].

Closure conversion also compiles `Eval code data` to `Call (Install code data)`. The new `Install` operation takes the same steps of fetching program text from an oracle and checking it against the compiled machine code. Unlike `Eval`, `Install` does not execute any of the code, i.e., it simply loads the new code into the global code table and returns the ID (code pointer) of the first new function. Here, `Call` is the usual IL primitive that executes a given code table entry. Thus, the dynamic execution associated with `Eval` has been compiled away to reuse existing functionality for making function calls. It remains to implement dynamic loading, i.e., `Install`.

The `Install` operation is passed unchanged through the middle compiler phases until `DataLang`. In the final ILs, the compilation of the current program reaches machine code, at which point it has "caught up" with the prior compilation of the machine code parameters to `Install` and is in the same byte- or word-level representation. In these final ILs, once the representations match, it is time to make use of the machine code parameters by compiling `Install` into the code that actually loads the compiled bytes into memory.

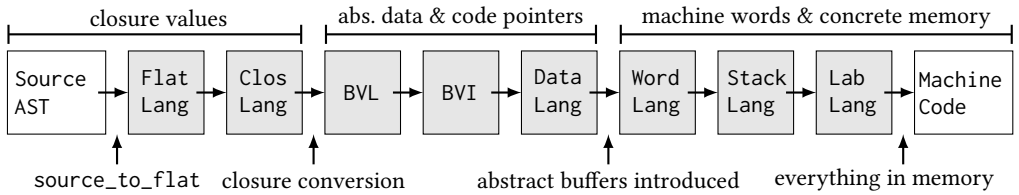

Fig. 7. Major tasks of the CakeML pipeline updated for `Eval`

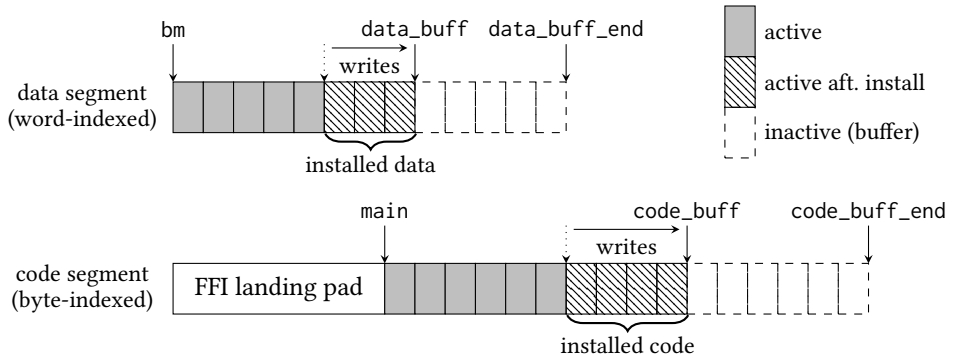

Fig. 8. Data and code segment layout for CakeML and their respective buffers. Data and code is dynamically installed by writing to the (contiguous) memory locations starting at their respective buffer pointers `data_buff` and `code_buff` and then incrementing the buffer pointers accordingly.

4.2 Compiling the Code and Data Installations

To understand the lower-level compilation steps for `Install`, it is perhaps best to start at the concrete, machine-code level and the compiler's design for code insertion at runtime. Figure 8 shows the run-time data and code segment layout for CakeML. The data segment (pointer `bm`) is a sequence of machine words containing liveness information necessary for run-time garbage collection (GC) and other constant text data, e.g., string constants. The code segment (pointer `main`) is a sequence of bytes corresponding to machine instructions as produced by the verified assembler; note that the code segment is prefixed by a FFI landing pad which is used to jump to foreign functions. To support `Install` of new data and code, the data and code segments are allocated with additional fixed-size buffers (pointers `data_buff` and `code_buff`, respectively).

In order to install new code (resp. data), the runtime executes a sequence of memory writes to the code (resp. data) buffer, while moving the buffer pointers accordingly. After this installation process, the runtime can execute loaded code by issuing a jump to the corresponding memory location in the code segment. Compilation of `Install` to match the concrete memory layout described above proceeds in several steps.

4.2.1 Compilation to Abstract Buffers. The first substantial step of compiling `Install` occurs in the compilation from `DataLang` to `WordLang` (see Fig. 7). In this phase, the representations of abstract values are concretized to machine word representations in memory. Accordingly, all operations on values are replaced by per-byte and per-word stateful operations. For example, big integer values are compiled into a heap pointer to an array of machine words, and operations on big integer values are replaced with calls to CakeML's verified bignum library.

In WordLang, the code and data buffers are introduced as abstract components of the semantic state, together with two new syntactic operations CodeBufferWrite and DataBufferWrite whose semantics are to write a byte or machine word to their respective buffers, while remembering the new values that have been written (but not yet read by an Install). The semantics of Install in WordLang "flushes" the code and data buffers, grabbing all the new contents to compare against the oracle in the usual way. It also checks that the code_ptr and code_len parameters it is given agree with the location of the "new" code buffer contents, and similarly for data_ptr and data_len.

Each Install code data operation from DataLang is compiled into a sequence of calls to two helper routines Install_code code and Install_data data, followed by a final call to Install with arguments code_ptr, code_len, data_ptr, and data_len. Here, code_ptr points to a position in the code buffer (with length code_len) and data_ptr points to a location in the data buffer (with length data_len).

The routines Install_code and Install_data are inserted by the ahead-of-time compiler. They are implemented as loops that read from the code and data arguments using normal CakeML operations and they copy into the code and data buffers using primitives CodeBufferWrite and DataBufferWrite. This is a common pattern in CakeML where compiler phases insert function bodies to implement mechanisms, e.g., the aforementioned bignum library is implemented as a family of such functions, roughly, one for each bignum operation (addition, multiplication, etc.).

The CodeBufferWrite, DataBufferWrite, and Install operations persist largely unchanged through all WordLang compilation phases. The notable exception is register allocation, which fixes the registers used to implement buffer writes and to store the necessary pointers into both buffers.

4.2.2 Concretizing the Data Buffer. The next compilation phase uses StackLang, the last IL before code is flattened in preparation for assembly. As the name suggests, StackLang's compilation phases focus on the task of replacing abstract stack and GC operations with concrete ones.

The StackLang IL's semantics includes abstract operations that manipulate an abstract stack. It also includes semantic switches, that is, it has immutable flags in its semantic state that enable and disable parts of the semantics. One of these switches disables the abstract stack operations, making all these operations simply raise errors, and another switch disables all the abstract GC operations. As the compiler phases replace the abstract stack and GC operations with concrete implementations, the semantic switches are turned off to clarify that the resulting program will no longer use the abstract features. Indeed, one of the key uses of the aforementioned data segment (Fig. 8) is to provide a bitmap containing stack frame liveness information used by the GC to safely walk structures in memory. Accordingly, the data buffer is used to build an incremental bitmap for liveness information that will arise from new function calls made possible by Install.

The compiler phase that concretely places the stack in memory also replaces the abstract DataBufferWrite with an ordinary memory write to the (now concretely placed) data segment. In this way, the abstract data buffer is concretely implemented into memory and subsequently disabled. The correctness argument for this concretization step relies on the fact that the semantics of Install and DataBufferWrite check that the abstract data segment's position pointer and length match those of their input arguments.

4.2.3 Concretizing the Code Buffer. The compilation of CodeBufferWrite to actual writes of the code region proceeds in essentially the same way as for the data buffer, although the switch to the memory-write semantics is done at a lower-level IL where all code is represented as machine-code byte sequences. Here, Install becomes a no-op on both the abstract data and code buffers.

4.2.4 Enabling the Code. The Install operation that remains when the concretization of both data and code buffers are complete is intended to enable the newly installed code as safe jump

targets. This is implemented as a call to a function in the C FFI which performs the necessary instruction-cache flushes for the target processor.

There are some caveats with this operation. In principle it should also add execute permissions in a fine-grained way. Instead, for the time being, the `Eval` mechanism depends on the entire code buffer having both write and execute permissions, which not all operating systems permit. `Eval` is also only implemented on a subset of target architectures. Finally, the relevant proof does not model the actual semantics of the caches or memory permissions, and thus the proof does not check that this operation is used correctly.

5 PROOFS AND PROOF ENGINEERING

The most challenging part of the `Eval` project, of course, was updating all the proofs. The CakeML development is one of the larger formal developments in existence with, at the time of writing, over 320,000 lines of definitions and proofs written in HOL4, of which the compiler implementation accounts for over 220,000 lines of definitions and proofs. The `Eval` project added over 450 commits, which sum to over 58,000 lines of file insertion and 40,000 lines of file deletions. These numbers give only the approximate size of the project: line counts are a poor measure of project size. Furthermore, the insertion/deletion numbers were computed per-commit and summed, and do not tell us to what degree the `Eval` project repeatedly revised its own additions. It is not simple to compute a single changeset for the project because it was repeatedly merged with updates to mainline CakeML. However, it is clear it was a substantial undertaking.

5.1 Conjuring the Oracle at Source

The `source_eval` proof pass replaces the standard source semantics with one that includes an oracle. It is performed on the source language prior to `source_to_flat`. This "pass" composes with the other compiler passes, and their proofs, like any other, although it leaves the program syntax unchanged.

We saw (in Fig. 5 in Section 3) that the `Eval`-related state of the semantics can have three forms, `None`, which disables `Eval` entirely, the `EvalDecs` case, which contains the parameters needed to configure `Eval`, and a `EvalOracle` case, which was left unexplained. The `EvalOracle` case is designed so that the `source_eval` pass can switch to it.

The `EvalOracle` case includes an oracle field of type `num -> compiler_args`, a function from positive numbers to compiler input, where that input `type compiler_args = ((num * num) * v * dec list)` includes an environment ID, an encoded compiler state, and the input program. It also includes a flexible parameter that specifies how the `Eval` check is done and how it updates the oracle, so that this semantic variant can be configured for two different uses in this pass. The `EvalOracle` variant is visible in the CakeML language semantics, but is only expected to be used in the compiler proof. It could be hidden by producing a largely duplicated private copy of the source semantics, but this seemed unnecessary.

The simulation proof for `source_eval` is done in two steps. In the first pass, the `EvalDecs` semantics are replaced with an `EvalOracle` variant that behaves exactly the same as the `EvalDecs` case, but populates the oracle object, recording the input arguments each time `Eval` is called. The simulation proof shows that these two versions of the semantics stay exactly in lockstep, with the latter generating a useful oracle instance. Note that the per-trace CakeML semantics is terminating, with a clock parameter setting the maximum number of some step types. Thus, in each trace, the recording semantics records only a finite list of oracle information.

The next proof replaces the recording semantics with the typical oracle semantics used in other CakeML ILs, with the input arguments to the `Eval` operation ignored, the abstract compiler inputs fetched from the oracle, and the binary outputs checked. The oracle-recording semantics and the

oracle-consulting semantics are proven to stay exactly in step, as long as the oracle that would be recorded is a prefix of the (potentially infinite) oracle installed in the oracle-consulting semantics.

A technical proof step is done to construct an infinite oracle, if necessary, by taking the limit of the finite oracles constructed with increasing clock bounds. A further technicality arises when an infinite oracle cannot be constructed, because the program always calls `Eval` a finite number of times, or perhaps not at all. It is easy to pad the oracle with an infinite stream of trivial programs, but the oracle must also contain a valid sequence of compiler states. These states could be generated by applying the compiler function, except that the compiler is permitted to fail.

Fortunately, all the phases that can fail are late in the compiler pipeline, and all the phases where we need to claim validity of the whole oracle contents are earlier. The solution is to define a variant of the compiler which compiles as far into the pipeline as possible, with all its phases guaranteed to succeed. The final oracle, the one actually used in the semantic state after `source_eval`, is constructed by extracting the infinite limit of recorded oracles, discarding the state components, padding out the result with trivial syntax if it is finite, and reconstructing the compiler states by running the variant. This oracle state is correct for all phases present in the variant. It can also be proven that this oracle is an extension of the one recorded, since each successful recorded `Eval` event required the full compiler to succeed. Once this oracle is installed, the source semantics are now configured in a compatible way to the oracle-based semantics of the subsequent ILs.

5.2 Updating Simulation Proofs

All simulation proofs for each compiler pass needed updating for the new oracle and in-program compiler. Most CakeML compiler correctness results are proved using forward simulation theorems that have the following shape. Here `evaluate_IL1` is the functional big-step semantics of the source IL of this transformation from IL1 to IL2. The compilation function for this pass is `compile`. Variables `s` and `s1` are source states, variables `t` and `t1` are states of the target language. The source and target states are related by `state_rel`, which relates, e.g., the store, FFI and other configurations that need to stay in sync; `res_rel` similarly relates the execution results.

```
∀e s res s1 t.
  evaluate_IL1 e s = (res,s1) ∧ res ≠ Fail ∧
  state_rel s t ⇒
  ∃res1 t1.
    evaluate_IL2 (compile e) t = (res1,t1) ∧
    res_rel res res1 ∧ state_rel s1 t1
```

We updated the definition of each state relation to relate the new oracle and in-program compiler fields of the IL states. Below is a typical relation in the simple setting where the compilation function, `compile`, is stateless. The in-program compiler of state `s`, i.e. `s.compile`, is `compile` attached to the front of the compiler from state `t` downards, i.e. `t.compile`. Similarly, but backwards, the oracle that predicts programs at the target, i.e. `t.compile_oracle`, is the function that one gets by first running oracle for the source, i.e. `s.compile_oracle`, and then `compile`.

```
state_rel s t =
  ... ∧
  s.compile = pure_cc compile t.compile ∧
  t.compile_oracle = pure_co compile ∘ s.compile_oracle
```

The functions `pure_cc` and `pure_co` are defined as follows:

```
pure_cc f cc = (λcfg prog. cc cfg (f prog))

pure_co f = (λ(cfg, prog). (cfg, f prog))
```

The correctness proof for `compile` proceeds by induction on IL1's evaluation semantics, with most of the cases in the proof remaining unchanged. The new case for `Eval` (or its relevant variant) follows by induction because `s.compile` and `t.compile_oracle` are chosen so that whenever an oracle-program `e` is evaluated in IL1, the corresponding program `compile e` is evaluated in IL2.

Stateless compilation passes followed the simple sketch above, but many others required a compiler that depends on context and prior state. Such compilers make use of the compiler configuration component, `cfg`, which is simply ignored by `pure_cc` and `pure_co` definitions above.

5.3 Low-Level Intermediate Languages

Section 4.2 discussed the way in which the lower-level ILs compile the `Install` operator progressively into copy loops which update the data and code buffers followed by a flush operation. The reader has probably noticed that, although Section 4.2 is notionally about the design of the implementation, it also implicitly describes the structure of the proof in detail.

For instance, the `DataBufferWrite` operation is eventually compiled to a memory write. In a compiler not designed for verification, this operation would almost certainly not exist with its own name, and be generated as a memory write in the first instance. The distinction allows the IL semantics to highlight one of the essential proof obligations, which is the way the data buffer is split between a read-only region and a region being written. Data reads from the data buffer (when it is placed in memory) must always read from the already-written read-only part.

This is perhaps the ideal case for compiler verification. We do not mean to claim that the proof is ideal. We mean that the problem permits us to structure the (difficult) proof into a collection of steps whose nature can be easily understood by reading the source of the compiler phases themselves and the relevant parts of the semantics of the various ILs.

5.4 Higher-Level Intermediate Languages

Compiling `Eval` through the earlier phases of the compiler pipeline, through the functional-style ILs, is straightforward. The impact on the proofs turns out to be much less straightforward. We will highlight two particular complications as examples.

5.4.1 Assigning Code Identifiers. After closure conversion, function bodies are moved to the code table and called by their identifier. Clearly these identifiers must be unique or the wrong code will be called. New identifiers added by an `Install` event must not already be in the code table.

The IL representation at this point is a list of numbered function bodies, `prog : (int * expr) list`. Prior to the addition of `Eval`, the proof of correctness of some compiler phases had the explicit assumption `distinct (map fst prog)` that the identifiers in the input program are distinct.

These identifiers are allocated in a complicated way, with numerous compiler passes renumbering the program. In most cases this is to allow a compiler pass to add some functions it uses to implement a feature, which is done by incrementing all input identifiers by some constant n to make the identifiers $0..n$ available. In two cases, a compiler phase goes further, and uses an even/odd interleaving or similar to make space available for a larger collection of additional functions.

Adding `Eval`, and a program oracle `oracle`, adds three new name-distinctness assertions:

```
∀i. distinct (map fst (oracle i))

∀i. disjoint (set (map fst prog)) (set (map fst (oracle i)))

∀i < j. disjoint (set (map fst (oracle i))) (set (map fst (oracle j)))
```

Each phase of the compiler transforms both the program `prog` and the program oracle `oracle`, and each of these constraints must be shown to be preserved. The new distinction between ahead-of-time compilation and compilation at runtime expands this technical aspect of the proof substantially.

To make this slightly less tedious, we have employed some standard proof engineering strategies. For instance, we gather some of the disjointness facts into a single statement using a new combinator:

```
oracle_monotonic (set ∘ map fst) (λ x y. x ≠ y) (set (map fst prog)) oracle
```

This combinator captures the fact that the IDs associated with generations of syntax are distinct or ordered in the sense of some relation (here `λ x y. x ≠ y`). The combinator is also used with the relation `λ x y. x < y`, where the "monotonic" name fits better. The point of the combinator is that, in addition to proving facts about `set (map fst (compile_X prog))` for a compiler phase `compile_X`, we can express the preservation of disjointness as a single lemma (`compile_X_dyn` being the dynamic per-`Eval` variant of `compile_X`):

```
oracle_monotonic (set o map fst) (λ x y. x ≠ y) (set (map fst prog)) oracle
⟹ oracle_monotonic (set o map fst) (λ x y. x ≠ y)
        (set (map fst (compile_X prog))) (compile_X_dyn o oracle)
```

This lemma can be applied automatically via syntax-directed tactics when the `compile_X` proof is composed with those of its neighbours, far more easily than a fact about `set (map fst (compile_X prog))`. This simple change, and other similar ones, make it easier to integrate our proof changes, and hopefully will reduce proof maintenance effort in the future.

5.4.2 Trying to Avoid Syntax Restrictions. The previous section describes a problem with function IDs. The correctness proofs of the compiler phases for ILs with `Install` depend on assumptions about the IDs in the input syntax to the compiler phase. These assumptions link the proof together in a non-local way, as multiple compiler phases may lie betweeen the point at which function IDs are generated (during closure conversion) to the point at which the relevant assumption is needed.

It would be nice to avoid these syntax restrictions, and instead depend only on the assumption that the input program passes all the checks in the semantics. Furthermore, the `Install` operator does check that the function IDs being installed do not overlap any present in the code table.

Unfortunately, following this line of thinking turned out to be a dead end. We suspect that the syntax restrictions for function IDs could be removed, with some refactoring of various proofs, but we have no solution to the problem of *variable* IDs.

```
fun read_file fname = ... ;

fun parse_config input = ... ;

val config = parse_config
    (read_file
        "config.txt");

fun f x = let
    val y = get_param
        config "y";
    ...
  in result end;
```

Fig. 9. Interleaving of functions and globals in CakeML

5.4.3 Assigning Global Variable IDs. We have mentioned that the `source_to_flat` phase eliminates global environments, assigning unique numeric IDs instead. This applies both to functions and to global values.

The code block in Fig. 9 shows the interleaving between function definitions and global value declarations that is possible in CakeML, and other ML-family languages. A global value such as `config` is immutable and constant once created, but its value is computed from input data via a previously-defined function, and it is not a compile-time constant.

The `source_to_flat` pass maps global names to unique numeric IDs. The ID of `config` denotes a cell in a "globals array", where its value will be stored. Eventually the closure conversion passes will move all code into function bodies, at which point the code snippet that computes `config` will be moved into the body of some function. That function will execute exactly once, and write the value of `config` (and maybe other globals) to the array cell.

Like the code and data buffers, the "globals array" is initially represented as a special array type accessed with special operations, with the guarantee that each cell in the array can only be written (at most) once, and can only be read after it has been written. In a later pass this is replaced with a normal mutable data structure.

As we mentioned in Section 4.1, closure conversion is a key phase of compilation, and one which is critical to performance. The compiler makes substantial efforts to avoid creating closures when it is possible to make direct function calls instead. This optimisation depends on a known-value analysis (similar to global constant-propagation) which detects expressions that describe constants. Functions (such as `f` in Fig. 9) are values also, and their names are converted to call IDs by `source_to_flat` in the same way. To permit a direct call to a function, the known-value analysis must compute what value will be written to the globals array.

This known-value analysis scans the whole program to find the one expression that writes any given cell, e.g. the write of the cell for `f`. In the absence of `Eval`, it can then be assumed that any read of that cell will read the value written by that expression. Unfortunately, `Eval` complicates this argument. There might currently be only one site at which `append` can be written, but what if an `Eval` event introduces new syntax?

The only successful approach we found to this problem was to depend on disjointness again. Each `Eval` event introduces syntax which (possibly) writes cell IDs in the globals array. These IDs are allocated incrementally, forming another pointwise monotonic sequence. This monotonicity can be proven, but the proof is tedious, as it involves all the syntax of the program representation rather than just the top-level function IDs.

It should perhaps have been clear that these IDs would create complications. The known-value analysis is global, and `Eval` clearly confuses any global analysis. Unfortunately, this clarity comes in hindsight, and in the actual project the problem was discovered slowly in understanding the difficulties of updating the CakeML proofs as a whole.

6 PUTTING IT ALL TOGETHER

The compiler correctness proof guarantees that any semantically safe program will be compiled into an equivalent binary by the CakeML compiler. The final step is to show that this is feasible by demonstrating a user program that makes use of `Eval` and is proven to pass the checks in the source semantics.

Many CakeML programs can be proven safe by simple type checking. Unfortunately, the type checker simply rejects programs with `Eval`, as it cannot check the program that `Eval` will evaluate.

Figure 10 shows the actual REPL loop which is added to the main CakeML program shortly after the translation of the top-level CakeML compiler. As explained in Section 3.2, the translation of the compiler and its associated types also generates the decode functions for the compiler state and CakeML AST which are required to instantiate the `Eval`-related semantics state. The safety of the REPL code, which uses the `Eval` primitive indirectly via the `eval` function, can only be established with respect to these decode functions.

This generic REPL has been proven safe to execute, which means that the REPL, all its helpers, and the functions it executes dynamically via `Eval`, all use CakeML's language features safely. There are three essential components to this proof. One is simple: aside from the `eval` step, all the other subfunctions are themselves type safe, as can be checked statically by the type checker. Next, the `safe_decs` really are safe, because the `check_and_tweak` function runs the type checker dynamically after "tweaking" the program to include the necessary print statements. Finally, the `eval` function safely calls `Eval`, that is, it calls the CakeML compiler with the correct compiler state and presents the inputs and outputs of the compiler to the `Eval` primitive.

```
fun repl (parse, types, conf, env, decs, input_str) =
  (* input_str is passed in here only for error reporting purposes *)
  case check_and_tweak (decs, (types, input_str)) of
    Inl msg => repl (parse, types, conf, env, report_error msg, "")
  | Inr (safe_decs, new_types) =>
    (* here safe_decs are guaranteed to not crash;
       the last declaration of safe_decs calls !Repl.readNextString *)
    case eval (conf, env, safe_decs) of
      Compile_error msg => repl (parse, types, conf, env, report_error msg, "")
    | Eval_exn e new_conf =>
      repl (parse, roll_back (types, new_types),
            new_conf, env, report_exn e, "")
    | Eval_result new_env new_conf =>
      (* check whether the program that ran has loaded in new input *)
      if !Repl.isEOF then () (* exit if there is no new input *) else
        let val new_input = !Repl.nextString in
          (* if there is new input: parse the input and recurse *)
          case parse new_input of
            Inl msg => repl (parse, new_types,
                    new_conf, new_env, report_error msg, "")
          | Inr new_decs => repl (parse, new_types,
                    new_conf, new_env, new_decs, new_input)
        end
```

Fig. 10. The central REPL function. The variable `decs` ("declarations") is the next AST snippet to be evaluated, `conf` contains the compiler state, and `types` the type checker state.

One significant complication remains: the first compiler state to be used by the in-program compiler must be the final state of the ahead-of-time compiler which compiled the whole REPL program (including a copy of the in-program compiler). That state cannot be encoded into the compiled program without creating a circular dependency. The solution is to encode the state into a file, and read the file contents via the FFI as the REPL starts. This adds an assumption to the correctness proof, that the byte array read from the file is the one that was saved.

The subfunctions of the `repl` function above must be type-correct, but they can be stateful. As suggested by the comment about `!Repl.nextString`, some of the steps of the REPL loop can be replaced by adjusting global references. The first step taken by the basic REPL is to evaluate the contents of the file `repl_boot.cml`, which is an ordinary type-correct CakeML source file that installs additional REPL features, such as a customisable prompt.

The Candle REPL builds on the same `repl` loop, instead using a parser for OCaml-style source. Candle requires some additional checks to protect the prover kernel types, the details of which are described in Abrahamsson et al. [2022].

7 EVALUATION

The new CakeML REPL is now available as a default feature of the CakeML binary compiler. The current binary should always be available via https://cakeml.org/ and will operate in REPL mode when invoked via `cake --repl`. At the time of writing, the only supported platform is the Linux operating system running on an x64 CPU. Figure 11 lists example input and output of this REPL. In

```
[linux]> ./cake --repl
Welcome to the CakeML read-eval-print loop.
...
val pp_exn = <fun>: exn -> pp_data   val isFile = <fun>: string -> bool
(* preamble loaded *)
> val x = (1, 2, "hello");
(* inserted code: 1720 bytes (94745 used); data: 33 machine words (1459 used) *)
val x = (1, 2, "hello"): int * int * string
> fun f i = if i < 2 then (i, "true") else f (i - 2);
(* inserted code: 1540 bytes (96285 used); data: 26 machine words (1485 used) *)
val f = <fun>: int -> int * string
> f 123;
(* inserted code: 1382 bytes (97667 used); data: 23 machine words (1508 used) *)
val it = (1, "true"): int * string
> List.map f [1,2,3,4,5,6,7,8];
(* inserted code: 1688 bytes (99355 used); data: 52 machine words (1560 used) *)
val it = [(1, "true"); (0, "true"); (1, "true"); (0, "true");
  (1, "true"); (0, "true"); (1, "true"); (0, "true")]: (int * string) list
```

Fig. 11. Demo use of the REPL

this input/output sequence, the input declarations such as `> val x = (1, 2, "hello");` were each compiled, installed, and run as machine code dynamically.

The current design does not yet permit reclamation of code or data written into the code and data buffers, and the buffer sizes are fixed at compile time. In our simple tests, small input programs typically install a few kilobytes of code and a few hundred bytes of data (see comments below each line in Fig. 11). Note that more code is being compiled in each of these steps, as the REPL does its printing by inserting print code into the program being `Eval`-ed. This suggests that a buffer size in the tens of megabytes provides space for tens of thousands of `Eval` events, which is enough for simple interactive use, and larger buffer sizes can be configured if necessary. We hope, in future work, to explore a JIT-style hybrid REPL which interprets rather than compiles program snippets that do not create new functions.

While there is plenty of room for future engineering upgrades, the verified REPL built here is not a toy prototype. The Candle prover is usable for interactive proofs. In a recent test, loading its standard basis theory requires 4356 `Eval` operations and takes 98 seconds (on a standard x86-64 i7-7700K running at 4.20 GHz). This theory is a port of the basis theory of HOL Light [Harrison 2009], and is compatible with HOL Light (based on OCaml), which loads it in 74.94 seconds.

We have not otherwise investigated the performance of `Eval`-installed functions. Apart from the absence of global dead-code elimination, the in-program CakeML compiler is largely similar to the standard ahead-of-time version, and should give similar performance.

8 ALTERNATIVE APPROACHES AND RELATED WORK

8.1 Alternative Approaches

This paper has focused on presenting the `Eval` design as it is implemented and verified in current CakeML. The verification turned out to be challenging, which raises the question, could it have been simplified by a more restrictive design? Prior to the `Eval` project, the first version of the CakeML compiler [Kumar et al. 2014] supported a REPL by compiling to a verified x64 bytecode interpreter.

This REPL was verified via a simpler argument, in which the compiler and REPL run in a logically separate universe to the dynamically evaluated programs. Only simple values can pass back and forth between these universes, much like the situation with the C FFI in the modern CakeML system. The REPL steps were wrapped around two instances of the compiler proof, switching between them as a special case.

The goal of the `Eval` work is to support a standard LCF-style prover design, where dynamically evaluated code can define new tactic functions, and those functions can manipulate theorem and term values from the prover kernel and call to inference functions of the prover kernel. The tactics may even recurse to themselves via higher-order tactic combinators provided by the prover library, which creates complex chains of calls between dynamically added tactic code and key functions from the prover core. The original CakeML design could only support such a use-case if the entire theorem prover implementation was placed in the dynamic side of the world split, which means it would have to be parsed and rebuilt at runtime, and this parsing would be part of the correctness argument. Modifying the REPL might improve the situation, but that is complicated by the fact that the REPL design is fixed by the proof. The current design, in which the REPL is verified as a CakeML program, is far more flexible.

More exotic designs were briefly considered, such as running the dynamic text interpreted, or running the dynamic program in a separate world with a remote protocol for manipulating theorem values in the prover world without sharing types. Our conclusion was that all these designs are workarounds, and that it is far preferable to pursue the current design, with its difficulties, which is more general. One simplification worth considering was forbidding recursion of `Eval` itself, i.e., forbidding further `Eval` events while an `Eval` is in progress, which would solve some order-of-evaluation issues we encountered in the proof. This would be compatible with an LCF-style prover, but would rule out a design we want to explore in future work, one where the prover kernel exposes a verified proof-by-evaluation mechanism that makes use of `Eval`.

8.2 Related Work

Verified JIT Compilation. There are a number of other verified environments that provide dynamic evaluation or a just-in-time compiler. Jitawa [Myreen and Davis 2011] is a verified just-in-time compiler from an untyped bytecode to x86 machine code, which had a simpler proof-of-concept precursor [Myreen 2010]. Compared to CakeML, Jitawa achieves good performance with less complex optimisations, which means its proof is correspondingly simpler. Jitk [Wang et al. 2014] adapts the final phases of the CompCert verified compiler infrastructure [Leroy 2009a] to build a verified just-in-time compiler for plugging various monitoring and filtering functions safely into the Linux kernel. Like this work, this project adapts an existing ahead-of-time compiler into a just-in-time case. Its design contains a similar world separation to the original CakeML project, which is required in this case, as the compiled Linux plugins must run in a specifically constrained environment. Barrière et al. [Barrière et al. 2021] report on a verified study of a more conventional just-in-time compiler design. They study the interesting problem of optimising a program based on dynamic observations of its behaviour, and then undoing the optimised compilation when the dynamic behaviour changes. This ambitious design allows some just-in-time compilers to get better performance than their ahead-of-time counterparts, especially for high-level untyped languages like JavaScript. The related project FM-JIT [Barrière et al. 2023] builds a simple JIT compiler for a high-level language, with JIT-specific optimisations, using the CompCert compiler as part of the backend. The CompCert compiler, unlike CakeML, is not self-hosting, so a major task for the project is to interleave the verified pure functions of the compiler (running via OCaml in the implementation) with the generated native code and special JIT mechanisms (modelled via a special monad in the verification). By contrast, we provide no specific optimisations for our dynamically

execute code, and are content to have it run with the same performance as if it were statically compiled. We also require fewer trusted code extraction components for our verified binary.

Compositional Compiler Correctness. At a high-level, our work is broadly similar to extant research in *compositional* compiler correctness , where statically compiled code modules are guaranteed to correctly inter-operate with other compiled code modules, possibly produced by a different compiler or from a different source language [Kang et al. 2016; Neis et al. 2015; Perconti and Ahmed 2014; Song et al. 2020; Stewart et al. 2015; Wang et al. 2020]. Many of these prior efforts extend CompCert with new forms of compositional correctness guarantees, including verified separate compilation which has been part of CompCert since version 2.7 [Kang et al. 2016]. Correct compilation of `Eval` also requires producing inter-operable code, but the novel difficulty is to develop verification guarantees for code that is *dynamically* compiled, loaded, and executed at runtime; this is the key challenge solved by our oracle-based proof strategy. Nevertheless, correctness of `Eval` is also (partly) simpler than compositional correctness in that the dynamically produced code comes from the same source CakeML language, and uses the same in-program CakeML compiler. Lightweight proof techniques for separate compilation that were used in CompCert have analogues in our proofs. For example, several intra-functional analysis passes in CakeML were straightforwardly adapted by extending their respective inductive proofs with a case for `Eval` (see [Kang et al. 2016, Section 2.2]); stateful compilation passes were more challenging as they interact intricately with how the oracles are chosen (Section 2.3). An interesting line of future work is to make our in-program compiler configurable so that certain optimization parameters can be switched on or off at runtime. The challenge is that different executions of the in-program compiler no longer run in lock-step, which complicates the verification [Kang et al. 2016, Section 2.3].

Oracle Semantics and Proofs. Oracle(-like) approaches have been used in formal semantics, e.g., as a means of factoring out the behavior of concurrently executing threads [Hobor et al. 2008; Pohjola et al. 2022] or modeling external nondeterminism and interactivity [Leroy 2009a; Owens et al. 2016; Xia et al. 2020]; Milner [1975] describes the role of oracles in process semantics as (paraphrased) "fictitious agents providing a sequence of truth values used to resolve arbitrary choices occurring in a (process) behavior". In our work, the `Eval` oracle is added solely as a proof gadget, i.e., it does not appear in the source or target semantics; the CakeML and CompCert compilers both use similar gadgets in their backend proofs [Leroy 2009b; Tan et al. 2019]. Similar to these prior approaches, the `Eval` oracle serves as a proof engineering tool to decouple reasoning about source-level `Eval` semantics from the rest of the compiler phases.

9 CONCLUSION

We have extended CakeML with `Eval`, resulting in a verified language environment with a dynamic computation feature permitting general interaction between the original and dynamically installed/executed code. It is, to our knowledge, the first verified run-time environment capable of supporting a standard LCF-style theorem prover design.

ACKNOWLEDGMENTS

We thank Rini Banerjee, Raj Troll, Peter Sewell, the PLDI anonymous reviewers, and our paper shepherd for their feedback on this paper. This work was supported by the Swedish Foundation for Strategic Research, by the Swedish Research Council (grant no. 2021-05165), by Trustworthy Systems and CSIRO's Data61, and by the European Research Council (ERC) via the "ELVER" grant (grant agreement no. 789108).

DATA-AVAILABILITY STATEMENT

The Eval extension is now included in the main version of CakeML. CakeML is free software, and we encourage other authors to use or extend our work. A software artefact is available at https://zenodo.org/record/7813942 [Sewell et al. 2023] containing pre-built binaries of the REPL and the Candle prover that can be used to reproduce our versions at the time of writing of this paper. The CakeML repository https://github.com/CakeML/cakeml will host future versions and contributions.

REFERENCES

Oskar Abrahamsson, Magnus O. Myreen, Ramana Kumar, and Thomas Sewell. 2022. Candle: A Verified Implementation of HOL Light. In *ITP (LIPIcs, Vol. 237)*, June Andronick and Leonardo de Moura (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:17. https://doi.org/10.4230/LIPIcs.ITP.2022.3

Aurèle Barrière, Sandrine Blazy, Olivier Flückiger, David Pichardie, and Jan Vitek. 2021. Formally verified speculation and deoptimization in a JIT compiler. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–26. https://doi.org/10.1145/3434327

Aurèle Barrière, Sandrine Blazy, and David Pichardie. 2023. Formally Verified Native Code Generation in an Effectful JIT: Turning the CompCert Backend into a Formally Verified JIT Compiler. *Proc. ACM Program. Lang.* 7, POPL (2023), 249–277.

Michael J. C. Gordon, Robin Milner, F. Lockwood Morris, Malcolm C. Newey, and Christopher P. Wadsworth. 1978. A Metalanguage for Interactive Proof in LCF. In *POPL*, Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski (Eds.). ACM Press, 119–130. https://doi.org/10.1145/512760.512773

Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. 1979. *Edinburgh LCF*. LNCS, Vol. 78. Springer. https://doi.org/10.1007/3-540-09724-4

John Harrison. 2009. HOL Light: An Overview. In *TPHOLs (LNCS, Vol. 5674)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer, 60–66. https://doi.org/10.1007/978-3-642-03359-9_4

Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle Semantics for Concurrent Separation Logic. In *ESOP (LNCS, Vol. 4960)*, Sophia Drossopoulou (Ed.). Springer, 353–367. https://doi.org/10.1007/978-3-540-78739-6_27

Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight verification of separate compilation. In *POPL*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 178–190. https://doi.org/10.1145/2837614.2837642

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *POPL*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 179–192. https://doi.org/10.1145/2535838.2535841

Xavier Leroy. 2009a. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

Xavier Leroy. 2009b. A Formally Verified Compiler Back-end. *J. Autom. Reason.* 43, 4 (2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. 2019. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In *ESOP (LNCS, Vol. 11423)*, Luís Caires (Ed.). Springer, 30–59. https://doi.org/10.1007/978-3-030-17184-1_2

Robin Milner. 1975. Processes: A Mathematical Model of Computing Agents. In *Logic Colloquium '73*, H.E. Rose and J.C. Shepherdson (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 80. Elsevier, 157–173. https://doi.org/10.1016/S0049-237X(08)71948-7

Magnus O. Myreen. 2010. Verified just-in-time compiler on x86. In *Principles of Programming Languages (POPL)*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 107–118.

Magnus O. Myreen. 2021. The CakeML Project's Quest for Ever Stronger Correctness Theorems (Invited Paper). In *ITP (LIPIcs, Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1:1–1:10. https://doi.org/10.4230/LIPIcs.ITP.2021.1

Magnus O. Myreen and Jared Davis. 2011. A Verified Runtime for a Verified Theorem Prover. In *ITP (LNCS, Vol. 6898)*, Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk (Eds.). Springer, 265–280. https://doi.org/10.1007/978-3-642-22863-6_20

Magnus O. Myreen and Scott Owens. 2014. Proof-producing Translation of Higher-order logic into Pure and Stateful ML. *Journal of Functional Programming (JFP)* 24, 2-3 (May 2014), 284–315. https://doi.org/10.1017/S0956796813000282

Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: a compositionally verified compiler for a higher-order imperative language. In *ICFP*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 166–178. https://doi.org/10.1145/2784731.2784764

Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. LNCS, Vol. 2283. Springer. https://doi.org/10.1007/3-540-45949-9

Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *ESOP (LNCS, Vol. 9632)*, Peter Thiemann (Ed.). Springer, 589–615. https://doi.org/10.1007/978-3-662-49498-1_23

Scott Owens, Michael Norrish, Ramana Kumar, Magnus O. Myreen, and Yong Kiam Tan. 2017. Verifying efficient function calls in CakeML. *Proc. ACM Program. Lang.* 1, ICFP (2017), 18:1–18:27. https://doi.org/10.1145/3110262

Lawrence C. Paulson. 1983. A Higher-Order Implementation of Rewriting. *Sci. Comput. Program.* 3, 2 (1983), 119–149. https://doi.org/10.1016/0167-6423(83)90008-4

James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-language Semantics. In *ESOP (LNCS, Vol. 8410)*, Zhong Shao (Ed.). Springer, 128–148. https://doi.org/10.1007/978-3-642-54833-8_8

Johannes Åman Pohjola, Alejandro Gómez-Londoño, James Shaker, and Michael Norrish. 2022. Kalas: A Verified, End-To-End Compiler for a Choreographic Language. In *ITP (LIPIcs, Vol. 237)*, June Andronick and Leonardo de Moura (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 27:1–27:18. https://doi.org/10.4230/LIPIcs.ITP.2022.27

Thomas Sewell, Magnus O. Myreen, Yong Kiam Tan, Ramana Kumar, Alexander Mihajlovic, Oskar Abrahamsson, and Scott Owens. 2023. *Cakeml+Eval Artifact*. https://doi.org/10.5281/zenodo.7813942

Konrad Slind and Michael Norrish. 2008. A Brief Overview of HOL4. In *TPHOLs (LNCS, Vol. 5170)*, Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar (Eds.). Springer, 28–32. https://doi.org/10.1007/978-3-540-71067-7_6

Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2020. CompCertM: CompCert with C-assembly linking and lightweight modular verification. *Proc. ACM Program. Lang.* 4, POPL (2020), 23:1–23:31. https://doi.org/10.1145/3371091

Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *POPL*, Sriram K. Rajamani and David Walker (Eds.). ACM, 275–287. https://doi.org/10.1145/2676726.2676985

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. 2019. The verified CakeML compiler backend. *J. Funct. Program.* 29 (2019), e2. https://doi.org/10.1017/S0956796818000229

Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. 2014. Jitk: A Trustworthy In-Kernel Interpreter Infrastructure. In *OSDI*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 33–47.

Yuting Wang, Xiangzhe Xu, Pierre Wilke, and Zhong Shao. 2020. CompCertELF: verified separate compilation of C programs into ELF object files. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 197:1–197:28. https://doi.org/10.1145/3428265

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. https://doi.org/10.1145/3371119

Received 2022-11-10; accepted 2023-03-31