Verifying Efficient Function Calls in CakeML

SCOTT OWENS, University of Kent, United Kingdom
MICHAEL NORRISH, Data61, CSIRO / ANU, Australia
RAMANA KUMAR, Data61, CSIRO / UNSW, Australia
MAGNUS O. MYREEN, Chalmers University of Technology, Sweden
YONG KIAM TAN, Carnegie Mellon University, USA

We have designed an intermediate language (IL) for the CakeML compiler that supports the verified, efficient compilation of functions and calls. Verified compilation steps include batching of multiple curried arguments, detecting calls to statically known functions, and specialising calls to known functions with no free variables. Finally, we verify the translation to a lower-level IL that only supports closed, first-order functions.

These compilation steps resemble those found in other compilers (especially OCaml). Our contribution here is the design of the semantics of the IL, and the demonstration that our verification techniques over this semantics work well in practice at this scale. The entire development was carried out in the HOL4 theorem prover.

CCS Concepts: • **Software and its engineering → Compilers**; **Formal software verification**;

Additional Key Words and Phrases: Compiler verification, ML, verified optimisations

ACM Reference format:
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, Article 18 (September 2017), 27 pages.
https://doi.org/10.1145/3110262

1 INTRODUCTION

A compiler for a functional or object-oriented language must be able to compile function calls without much static information about the function being called. However, when more is known about the called function, the compiler should generate more efficient code for the call. In the case of an ML-like language, several specific challenges are present: the function being called might not be statically known; curried functions are passed arguments one at a time; and functions with free variables need access to their values from the closure. In this paper, we describe a particular solution for efficiently compiling ML-style function calls, and show how we verified its correctness.

A verified compiler is a compiler accompanied with a proof — almost certainly a machine checked proof built in an interactive theorem proving system such as Coq, HOL4, or Isabelle — that the semantics of the source language are respected by execution of the generated code when run on the target machine. The CompCert C compiler is one example [Leroy 2009], as is the CakeML compiler [Kumar et al. 2014], which we build on here.

CakeML is a call-by-value, impure functional language based on Standard ML. It supports modules, signatures, mutually recursive functions, pattern matching, exceptions, datatypes, references, mutable arrays, immutable vectors, strings, foreign function calls, etc. We recently released a new

2475-1421/2017/9-ART18
https://doi.org/10.1145/3110262

backend with a series of 12 intermediate languages (ILs) designed to support optimisations at many different levels of abstraction [Tan et al. 2016]. The large number of ILs separates out orthogonal aspects of compilation so that the correctness proof for a particular transformation only focusses on one thing at a time. This is a practical architecture for a verified compiler, and CompCert also has a large number of ILs. Here we focus on a particular IL called CloSLang (short for Closure Language).

In this paper, we present optimisations on CloSLang that introduce true multiple-argument functions (neither curried nor tupled), and improve call sites where the function being called is statically known. Although similar optimisations are common, and CloSLang shares some features with OCaml's Clambda IL, our focus is on the verification techniques and CloSLang's formal semantics.

Our main contribution is a coherent set of design choices that have allowed us to formally verify, in a practical setting, several transformations that are fundamental to any optimising compiler for a functional language. These choices break down into 4 areas: CloSLang's design with both ML-style and C-style functions; CloSLang's formal semantics; syntactic relations between original and optimised values for verifying optimisations that do not affect the structure of closure values; and an untyped step-indexed logical relation for verifying optimisations that do affect closures.

To our knowledge, this is the first treatment of verified compilation for higher-order functions that addresses these important optimisations. The initial version of CakeML [Kumar et al. 2014] treated them naively, with applications feeding one argument in at a time, and always jumping to the code pointer loaded from the heap-allocated closure record. Pilsner [Neis et al. 2015] does the same, although it does support inlining, contification, and hoisting optimisations that we do not address here. The most recent work on CakeML [Tan et al. 2016] mentions CloSLang and the treatment of multi-argument functions, but it does not explain the semantics, or give any information on how the verification works. It also has no optimisations for calls to statically known functions.

Critically, our optimisations are verified in the context of an end-to-end verified compiler. By fitting into a flow that starts from a source-level semantics—against which programs can be verified [Guéneau et al. 2017; Myreen and Owens 2014]—and ends with machine code, we ensure our optimisations and their verifications work out in practice: we have not made any unrealistic assumptions, nor proved correctness theorems that are too weak to be used.

We start with a brief, informal explanation of CloSLang (§2), and then motivate its design with concrete examples of the verified optimisations (§2.3). Next we move on to the formal semantics of CloSLang (§3), and our proof techniques (§4). Lastly we turn to the verification of the optimisations (§5 and §6), and finally to the translation into a language that does not have higher-order functions (§7). Our proof scripts are available at code.cakeml.org.

Design considerations. Although fitting in with the existing CakeML infrastructure ensures that our verification techniques scale to real systems, and that the theorems we prove are not over-simplified, it does place some limits on the design decisions behind CloSLang and our optimisations.

First, CakeML is a direct-style compiler that implements function calls with a stack and calling convention, rather than using intermediate languages in continuation passing style [Appel 1992] or A-normal form [Flanagan et al. 1993]. This makes some of our optimisations more complicated, but it also means that it is easier to compile to our more performant C-style calls.

Second, CakeML encourages a curried programming style, similar to OCaml and Haskell, in contrast to the tupled style encouraged by Standard ML. This makes it important to optimise calls to statically unknown multi-argument functions, but less important to do optimisations that un-box tupled arguments.

Last, CakeML uses untyped ILs, and so we do not address verification techniques that rely on static types. However, our optimisations can rely on type soundness: all of the ILs' semantics get stuck when primitives are misapplied. The correctness theorems for each optimisation and compiler pass assume that the semantics cannot get stuck on the program being compiled, and they conclude that the resulting code is semantically equivalent to the source, and so cannot get stuck either. CakeML's verified type inferencer [Tan et al. 2015] allows these assumptions to be discharged at the top level.

In each of these cases, both CakeML's choices and our choices are mainstream and were informed by the design of the OCaml native code compiler. Most importantly, we hypothesise that our proof techniques would be applicable to many other functional language compilers, even if the details differ.

Notation. With the exception of §2.3, all of the definitions and theorems are typeset directly from our HOL4 sources. HOL4 syntax generally follows ML, with the exception that constructors in type and term definitions are curried (Haskell style). Definitions of boolean typed functions use $\Longleftrightarrow$, and logical implication (as well as pattern matching) uses $\Rightarrow$. Records use `.` for projection, and `with` for field update.

We use the following functions for lists: hd takes the first element; nth takes the nth element counting from 0; (@) appends them; all tests whether a predicate holds of all elements of a list; all2 takes a binary relation and two lists and checks that they are the same length and pairwise related; all2i is similar but passes the index of the elements to the relation too. map maps a function over a list and mapi is similar but also passes the index. foldr does a right fold and genlist f n builds the list $[f\ 0;\ \ldots;\ f\ (n - 1)]$. For sets we mostly use standard notation; f " s is the image of s under a function f.

2 CLOSLANG AND ITS OPTIMISATIONS

The ClosLang IL (Fig. 1) is based on an untyped lambda calculus with de Bruijn indices.[1] ClosLang extends this lambda calculus with extra features (§2.1, §2.2) that enable compilation of efficient function calls from the previous language's simple one-argument-at-a-time semantics.

In moving from CakeML source to ClosLang, earlier stages of compilation remove algebraic data types, pattern matching, modules, and top-level declarations. Datatypes are compiled to blocks created by primitive operator Cons and accessed with primitive operator El. Cons allocates a tagged block of a fixed size, and El projects a given index from a given block. Top-level declarations are compiled into expressions that write their values into write-once global reference cells. Top-level recursive (and mutually recursive) functions are supported via a knot-tying indirection through this state.[2] For example,

```
fun len x = case x of [] => 0 | h::t => 1 + len t
```

[1] The de Bruijn indices are inherited from earlier CakeML passes, but they are convenient here, so that our mechanised proofs do not need to explicitly deal with α-equivalence.

[2] Knot-tying avoids the array that is used in the implementation of expression-level `let recs`.

```
exp = Var num                                  (* de Bruijn variable *)
 | If exp exp exp                              (* Conditional *)
 | Let (exp list) exp                          (* Local bindings *)
 | Tick exp                                    (* No-op, decrement clock in semantics *)
 | Raise exp                                   (* Raise exception *)
 | Handle exp exp                              (* Catch exception *)
 | Call num num (exp list)                     (* C-style call *)
 | App (num option) exp (exp list)             (* ML-style call *)
 | Op op (exp list)                            (* Primitive operator *)
 | Fn (num option) (num list option) num exp   (* Anonymous function *)
 (* Local recursive functions *)
 | Letrec (num option) (num list option) ((num × exp) list) exp

v = Number int                                 (* Integer values *)
 | Word64 word64                               (* 64-bit unsigned word values *)
 | Block num (v list)                          (* Tagged, heap-allocated block of values*)
 | RefPtr num                                  (* Pointer to a mutable ref or array *)
 (* Closure with an optional code table pointer, partially applied arguments,
      environment, arity, and body *)
 | Closure (num option) (v list) (v list) num exp
 (* Bundle of mutually recursive closures *)
 | Recclosure (num option) (v list) (v list) ((num × exp) list) num
```

num is the type of natural numbers, and op is the type of CakeML primitive operations.

Fig. 1. CLOSLANG abstract syntax and semantic values

becomes the following in CLOSLANG:

```
Op (SetGlobal 0)
 [Fn None None 1
   (Let [Var 0]
     (If (Op Equal [Op (Cons 0) [ ]; Var 0]) (const 0)
       (Let [Op El [const 0; Var 0]]
         (Let [Op El [const 1; Var 1]]
           (Op Add [App None (Op (Global 0) [ ]) [Var 0]; const 1])))))]
```

The function is stored in global location 0, which the recursive application looks up (Global 0). Pattern matching compilation has generated several Lets, and an equality check between the empty list, represented as an empty block with tag 0, and a let-bound variable. Calls to the El operator extract the head and tail of the list, which itself is just a two-element block with tag 0.[3] We use const n as a shorthand for Op (Const n) [ ], which just evaluates to an integer constant Number n.

Var, If, Let, Raise, and Handle all do what one would expect. Op supports around 40 different primitive operations for reference cells, integers, words, arrays, vectors, equality, foreign functions, tagged heap allocation, etc. Tick does nothing observable (it just returns the value of its expression), but is important for semantic bookkeeping in proofs, because our semantics—similar to the other

[3]Blocks store how many elements they have, and so the tag only needs to be unique among blocks of the same size and from the same datatype.

parts of CakeML—are in the functional big-step style [Owens et al. 2016] (see §3 for more details). Now we turn to the function and application expressions.

2.1 Functions

A Fn expression creates a closure with arity specified in the third parameter to Fn. The first parameter optionally indicates where the function's code will eventually be placed, and the second parameter optionally indicates which environment variables should be placed in the closure's environment (if None, all variables are). Initially, each Fn with body e is of the form Fn None None 1 e. As a simple example, consider the following CakeML function.

```
fn x => fn y => (x + y) + z
```

If we assume that z is bound to the innermost binding where the function is defined, then it will be compiled to ClosLang as follows:

```
Fn None None 1
 (Fn None None 1 (Op Add [Op Add [Var 1; Var 0]; Var 2]))
```

Our function call optimisations use the extra annotations and flexibility to combine iterated Fns, choose a location l for the function body's code, and annotate its free variables. In this case, there is just one free variable, 0, corresponding to z.

```
Fn (Some l) (Some [0]) 2 (Op Add [Op Add [Var 1; Var 0]; Var 2])
```

Even though top-level recursive functions are compiled into non-recursive ones via indirection through the global definition state, ClosLang supports the definition of lexically nested recursive function definitions via Letrec. This form has the same optional annotations as Fn, and defines a list of functions with arity/body-expression pairs.

2.2 Calls

The Call and App forms both call functions, and their arguments are evaluated right-to-left. App expressions call closures created with Fn, and Call expressions call C-style functions kept in an immutable code store, called the *code table*. The first argument of App is an optional annotation indicating the called function's eventual address in the code table; therefore we actually distinguish three kinds of function call: App None for calls to statically unknown functions, App (Some l) for calls to statically known functions whose bodies can contain free variables, and Call for calls to statically known closed functions.

For both kinds of App expressions, their argument lists must not be empty, and function expression must evaluate to a closure or recursive closure. Suppose that the App has n arguments, and the closure has arity m. For an App (Some l), the closure must also be annotated with Some l, and m must equal n. In other words, the closure's code pointer must actually be the code pointer that the application expected, and the arities must match. An App None expression has neither restriction. If $m > n$, then a new closure wrapper with arity $m - n$ is created to stores the n arguments. If $m < n$, then the function body is executed, and the result is applied to the remaining $n - m$ arguments. This flexible semantics lets multi-argument App and Fn expressions simulate the curried one-argument-at-a-time semantics of ML-like languages.

For example, in CakeML source, we can partially apply the above function to a single argument, and we want to keep that ability, even after the compiler converts it to a function with arity 2. On

the other hand, the following function can be applied to three arguments, even though the compiler will not create a Fn expression with arity 3.

```
fn x => fn y => let
  val z = x + y
in fn a => a + z end
```

A Call *ticks l es* expression calls the function at location *l* in the code table. The function must have arity len *es*, and it must not contain any free variables. An additional *ticks* number of Tick instructions are performed after evaluating the arguments, but before making the call.

2.3 Example Optimisations

Two scenarios motivate our design: the application of statically known functions, and the application of unknown functions to more than one argument. In each case, we should be able to get better performance than a straightforward implementation of the semantics of CakeML. In the first case, the application should avoid the cost of extracting a function pointer from the closure record and jumping to it, and in the second case, the allocation of intermediate closures should be avoided as each argument is given to the unknown function.

Elsewhere, we use the abstract syntax of ClosLang (Fig. 1). However, to make larger examples readable, this section uses a notation more akin to concrete syntax of the source with variable names rather than de Bruijn indices.

Example: statically known function calls. The following example illustrates how the ClosLang optimisations compile applications of statically known functions into fast C-style function calls. We start with the concrete syntax (same as SML's) of an input CakeML program. This CakeML code defines a naive, quadratic list reversing function using an append function and applies the reverse function to an example.

```
fun reverse xs = let
    fun append xs ys = case xs of [] => ys
                        | x::xs => x :: append xs ys
    fun rev xs = case xs of [] => xs
                  | x::xs => append (rev xs) [x]
  in rev xs end;
val example = reverse [1,2,3];
```

The code above initially compiles to the following in ClosLang.

```
SetGlobal 0 (fn xs => let
    fun append xs = fn ys => if xs = [] then ys
      else el 0 xs :: (append (el 1 xs)) ys
    fun rev xs = if xs = [] then xs
      else (append (rev (el 1 xs))) [el 0 xs]
  in rev xs end);
SetGlobal 1 ((Global 0) [1,2,3]);
```

Top-level definition `reverse` has been allocated to global location 0, and `example` to location 1. Functions `append` and `rev` are not defined at the top-level, so they are defined in a `Letrec`. Pattern matching is compiled into `if`s and calls to `el` which extract elements of heap-allocated blocks.

The first optimisation we run, called Multi, turns single-argument closures and applications into multi-argument versions. For example `fun append xs = fn ys => ...` turns into a declaration that takes two arguments, `xs` and `ys`, simultaneously without tupling or currying. We use the

notation `fun append ⟨xs,ys⟩ = ...` for a function that takes simultaneous arguments. We use a similar notation for applications: `append ⟨xs,ys⟩`.

```
SetGlobal 0 (fn xs => let
    fun append ⟨xs,ys⟩ = if xs = [] then ys
      else el 0 xs :: append ⟨el 1 xs, ys⟩
    fun rev xs = if xs = [] then xs
      else append ⟨rev (el 1 xs), [el 0 xs]⟩
  in rev xs end);
SetGlobal 1 ((Global 0) [1,2,3]);
```

The next two phases annotate the program with information regarding closure values. The first annotation phase, Number, places a unique location in the first argument of each Fn and Letrec, written here as a subscript. This is where in the code table the function's code will eventually be placed. It uses only even numbers, reserving the odd numbers for optimised entry points to the functions. The second annotation phase, Known, performs a simple flow analysis that tracks which closure values flow to which function applications. It annotates each App that applies a statically known closure value with that closure's number from the previous phase, but only if the closure's arity statically matches the number of arguments. These annotations are placed in the first argument to App, and are written here as superscripts. Note that Known tracks value flow even through the globals, and adds an annotation (^{4}) to the application of Global 0.

```
SetGlobal 0 (fn₄ xs => let
    fun append₀ ⟨xs,ys⟩ =
      if xs = [] then ys else
        el 0 xs :: append⁰ ⟨el 1 xs, ys⟩
    fun rev₂ xs =
      if xs = [] then xs else
        append⁰ ⟨rev² (el 1 xs), [el 0 xs]⟩
  in rev² xs end);
SetGlobal 1 ((Global 0)⁴ [1,2,3]);
```

The Calls optimisation is next. It moves *closed* function bodies into a separate immutable code store, called the *code table*. Each application of a closure value that has a code table entry turns into a Call expression, which can then be compiled to an efficient C-style function application. In our example, we get entries corresponding to $\texttt{append}_0$, $\texttt{rev}_2$ and $\texttt{reverse}_4$, which are at code table locations 1, 3 and 5 respectively.

```
SetGlobal 0 (fn₄ xs => Call 5 xs);
SetGlobal 1 (Call 5 [1,2,3]);

Code Table:
1 ↦ ⟨xs,ys⟩ => if xs = [] then ys else
                 el 0 xs :: Call 1 ⟨el 1 xs, ys⟩
3 ↦ xs => if xs = [] then xs else
            Call 1 ⟨Call 3 (el 1 xs), [el 0 xs]⟩
5 ↦ xs => let
              fun append₀ ⟨xs,ys⟩ = Call 1 ⟨xs,ys⟩
              fun rev₂ xs = Call 3 xs
            in Call 3 xs end
```

A simple dead-code elimination pass, Remove, replaces unused local function definitions with zeros. In our example, it only affects entry 5, which becomes:

```
5 ↦ xs => let val append = 0 val rev = 0
           in Call 3 xs end
```

These pointless bindings are removed by later optimisations. We defer the removal because it would require shifting de Bruijn indices (not shown in this concrete syntax).

The translation from ClosLang to the next intermediate language, BVL[4], compiles away all Apps and closure creations. In our example, there is only one closure left and no general-purpose function applications. The main expression becomes the following, using `cons` to allocate a block tagged as a closure, containing the arity and a pointer into the code table, but no environment since the $\texttt{fn}_4$ expression was closed.

```
SetGlobal 0 (cons clos_tag ⟨1, CodePtr 2⟩);
SetGlobal 1 (Call 5 [1,2,3]);
```

and the code table gets a new entry for `reverse`'s body:

```
4 ↦ ⟨xs, closure⟩ => Call 5 xs
```

This code table entry consists of only a Call that will be optimised to an efficient tail-call lower in the compiler. The entries for 0 and 2 are not needed.

The final code for our running example uses only fast C-style function calls and creates only one closure value, which is only allocated once when the program runs.

Example: statically unknown multi-argument function. In the example above, Known and Calls turned all applications into fast Calls. Our next example illustrates how applications of *statically unknown* multi-argument functions get compiled. We will use `foldl` as an example.

```
fun foldl f e [] = e
  | foldl f e (x::xs) = foldl f (f e x) xs
```

When this program has made its way to ClosLang, and gone through Multi and Known, the application of `f` has been transformed into a multi-argument application but has no annotations. This is because the location-number of the applied closure is not statically known and may vary. Contrast the way the recursive application of `foldl` (in global cell 0), does get annotated.

```
SetGlobal 0 (fn0 ⟨f, e, xs⟩ =>
  if xs = [] then e else
    (Global 0)^0 ⟨f, f ⟨e, el 0 xs⟩, el 1 xs⟩)
```

[4]Short for Bytecode Value Language, for historical reasons: its values are the same as in CakeML version 1's bytecode.

This program translates to the following BVL program.

```
SetGlobal 0 (cons clos_tag ⟨3, CodePtr 0⟩)

Code Table:
0 ⟨f, e, xs, closure⟩ => Call 1 ⟨f, e, xs⟩
1 ⟨f, e, xs⟩ =>
  if xs = [] then e else
    Call 1 ⟨f,
            let val arg2 = el 0 xs
                val arg1 = e
                val clos = f
            in
              if el 0 clos = 2 then
                Call None ⟨arg1, arg2, clos, el 1 clos⟩
              else Call app2_lib ⟨arg1, arg2, clos⟩
            end,
            el 1 xs⟩
```

The general-purpose multi-argument application has become a `let` followed by an `if`. The `let` computes the values of the arguments in right-to-left order and then the value of the closure that is to be applied. The `if` checks whether the closure expects the number of arguments it is receiving in this application. If the numbers match, then a call is made. If the numbers do not match (which is allowed), then a call is made to a library function (also written in BVL) that implements the mismatch semantics, by either allocating a wrapper in the case of too few arguments, or doing repeated calls in the case of too many.

3 CLOSLANG SEMANTICS

ClosLang has a functional big-step semantics [Owens et al. 2016]. This means that its semantics is given as a pure function resembling an interpreter, with a clock to ensure that the interpreter always terminates.[5] If the clock runs out, then a timeout exception is raised; a program diverges iff it times out for all starting clocks. The top-level semantics is a function from programs to observable behaviours of (possibly infinite) sequences of foreign function calls that uses this interpreter and quantification.

Figure 1 above gives the abstract syntax and the type of semantic values. Values are either integers, 64-bit words, tagged blocks for data constructors, pointers to the heap for references and arrays, closures, or recursive closures. The first argument to Closure and Recclosure is the optional location from the Fn or Letrec expression that created the closure. The second contains the values that have already been passed into the function, when the closure requires still more arguments before executing its body. The third is its environment. For a Closure, the fourth is the arity, and the last is the body. For a Recclosure, all of the arities and bodies from the creating Letrec are listed, and the last argument indicates which list element the Recclosure represents.

Figure 2 gives the types of states and results that the semantics uses. The state is a record with the following components:

- globals: a list of write-once global variables where None indicates uninitialised variables,
- refs: a finite map from pointers to reference values which are either arrays of values (`ref` cells become single-element arrays) or byte arrays,
- max_app: the maximum permitted arity of a function or call site,

[5] We cannot use non-terminating functions for formal reasoning in the theorem prover.

```
α ref = ValueArray (α list) | ByteArray bool (word8 list)

φ state =
<| globals : v option list; refs : num ↦ v ref; max_app : num;
   ffi : φ ffi_state; clock : num; code : num ↦ num × exp |>

abort = Rtype_error | Rtimeout_error
α error_result = Rraise α | Rabort abort
(α, β) semanticPrimitives$result =
   Rval α
 | Rerr (β error_result)
TypeErr = Rerr (Rabort Rtype_error)
Timeout = Rerr (Rabort Rtimeout_error)
```

Fig. 2. CLOSLANG states

- ffi: the state of the foreign function interface,
- clock: the clock to ensure that the semantics is well-defined, and
- code: the code table that maps code pointers to arity/function-body pairs, where the bodies must be closed.

Neither max_app nor code change as the program executes. We place them in the state so that we can conveniently ignore them when they are not needed. It is trivial to prove the requisite lemmas that they do not change. A limit on arities is enforced because the translation to BVL (§7) must introduce library functions for each possible arity.

The results signal successful evaluation (Rval), or an exception reaching the top level (Rerr). The exception could be from a Raise expression (Rraise), or indicate abnormal termination (Rabort) due to the clock running out (Rtimeout_error), or the misapplication of a primitive or other type error (Rtype_error). Our correctness theorems assume that Rtype_error cannot happen, which we know from CakeML's type inferencer correctness theorem and its type soundness theorem, combined with the correctness of the compiler passes that translate from source to CLOSLANG.

The semantics are given with a function, evaluate, that takes a list of expressions, an environment, and a state and returns a result and updated state. It evaluates each expression in turn, and if none of the results is an exception, the overall result will be a list of values of the same length. Figure 3 gives a few indicative clauses. The dec_clock function decrements the clock in the state by a given amount; note that the clock is decremented on CLOSLANG function calls, but not on other recursive calls to evaluate.

The Var case looks up the variable in the environment, or aborts if the variable is not bound. The Let case appends the values of the bindings onto the environment. The Fn case builds a closure with the given optional location, number of arguments and body expression. If the Fn was annotated with free variables, lookup_vars constructs a new environment that contains only those. Otherwise, the entire environment is used. The Tick case times out if the clock has reached 0, and otherwise decrements the clock by 1. We need the Tick instruction in the language so that our optimisations can insert them to avoid changing the number of times the program decrements the clock. We found

```
lookup_vars [ ] env = Some [ ]
lookup_vars (v::vs) env =
  if v < len env then
    case lookup_vars vs env of
      None ⇒ None
    | Some xs ⇒ Some (nth v env::xs)
  else None

evaluate ([Var n],env,s) =
  if n < len env then (Rval [nth n env],s) else (TypeErr,s)

evaluate ([Let es e],env,s) =
  case evaluate (es,env,s) of
    (Rval vs,s1) ⇒ evaluate ([e],vs @ env,s1)
  | (Rerr v,s1) ⇒ (Rerr v,s1)

evaluate ([Tick e],env,s) =
  if s.clock = 0 then (Timeout,s)
  else evaluate ([e],env,dec_clock 1 s)

evaluate ([Fn loc vs_opt n exp],env,s) =
  if n ≤ s.max_app ∧ n ≠ 0 then
    case vs_opt of
      None ⇒ (Rval [Closure loc [ ] env n exp],s)
    | Some vs ⇒
        case lookup_vars vs env of
          None ⇒ (TypeErr,s)
        | Some env′ ⇒ (Rval [Closure loc [ ] env′ n exp],s)
  else (TypeErr,s)

evaluate ([Call ticks dest es],env,s1) =
  case evaluate (es,env,s1) of
    (Rval vs,s) ⇒
      (case find_code dest vs s.code of
         None ⇒ (TypeErr,s)
       | Some (args,exp) ⇒
           if s.clock < ticks + 1 then
             (Timeout,s with clock := 0)
           else evaluate ([exp],args,dec_clock (ticks + 1) s))
  | (Rerr v8,s) ⇒ (Rerr v8,s)
```

Fig. 3. Core semantics of CakeML (selected clauses)

that this is generally easy to arrange, and it simplifies the proof effort by making most reasoning about the clock happen automatically.[6]

[6]In fact, it happens that our optimisations do not introduce Ticks directly, but we need the instruction when compiling into CakeML, since prior ILs can have Ticks inserted into them. We do need to use the extra ticks inside the Call instruction when introducing them from Apps.

```
evaluate ([App loc_opt e es],env,s) =
  if len es > 0 then
    case evaluate (es,env,s) of
      (Rval args,s1) ⇒
        (case evaluate ([e],env,s1) of
           (Rval f,s2) ⇒ evaluate_app loc_opt (hd f) args s2
         | (Rerr v,s2) ⇒ (Rerr v,s2))
    | (Rerr v9,s1) ⇒ (Rerr v9,s1)
  else (TypeErr,s)

app_kind = Partial_app v | Full_app exp (v list) (v list)

evaluate_app loc_opt f [ ] s = (Rval [f],s)

args ≠ [ ] ⇒
evaluate_app loc_opt f args s =
  case dest_closure s.max_app loc_opt f args of
    None ⇒ (TypeErr,s)
  | Some (Partial_app v) ⇒
      if s.clock < len args then (Timeout,s with clock := 0)
      else (Rval [v],dec_clock (len args) s)
  | Some (Full_app exp env rest_args) ⇒
      let t = len args - len rest_args
      in
        if s.clock < t then (Timeout,s with clock := 0)
        else
          case evaluate ([exp],env,dec_clock t s) of
            (Rval [v],s1) ⇒ evaluate_app loc_opt v rest_args s1
          | res ⇒ res
```

Fig. 4. Semantics of CLOSLANG application

A Call evaluates the arguments, checks that the clock is large enough, finds the location in the code table, checks that the number of arguments is correct, and then evaluates the (closed) body, using the arguments as the environment.

For the App case (Fig. 4), the arguments and then the function are evaluated and passed to a helper function evaluate_app that actually does the application. The semantics of evaluate_app is complicated by the handling of multiple-argument functions where the numbers of arguments and arity differ. The evaluate_app function is recursive, so that the function can be applied multiple times in the case of too many arguments.

Suppose that evaluate_app is called with n arguments *args*, and function f with arity m. It first calls dest_closure which returns a value indicating what should happen next: either a Partial_app with a new closure when $n < m$, or a Full_app with the function body, environment to use, and remaining arguments when $n \geq m$. It then checks that there are at least n clock ticks remaining, decrements the clock, and either returns the partially applied function, or evaluates the body and then recurs.

Several invariants are enforced by dest_closure: the number of arguments given to the closure so far must be less than the arity, and for a location-unannotated App None, the arity must be less

than the compile-time constant *s*.max_app. For a location-annotated App (Some *l*), the closure must have that location, have arity *n*, and not have any previously applied partial arguments. Compilation out of ClosLang (§7) relies on these invariants, and the semantics is a convenient place to enforce them, since they are trivially satisfied by the compilation into ClosLang.

4 OUR LOGICAL TOOLBOX

Our attitude to the verification task is pragmatic: we aim to demonstrate that realistic functional programming language optimisations are verifiable in concert, not to establish that any particular approach is the best. Our focus is on verifying the CakeML compiler as an end-to-end system that produces machine-code executables.[7]

To connect to the outside world, CakeML programs call a foreign function interface (FFI), passing and returning byte array values. The stream of these FFI calls is the observable result of running a CakeML program, so the top-level correctness theorem for each optimisation requires it to be unchanged. Thus, the various relations that we use to verify an optimisation all require that the FFI components of states (which includes the trace of all executed FFI calls) be exactly equal.

4.1 A Step-indexed Logical Relation

Our first tool is a step-indexed logical relation [Appel and McAllester 2001]. This is now a well-explored technique for establishing a semantic correspondence between a lower-level and a higher-level language via a typing discipline [Benton and Hur 2009; Hur and Dreyer 2011; Perconti and Ahmed 2014], or for establishing contextual equivalences between different implementation of ADTs that encapsulate significant representation changes behind type abstractions [Dreyer et al. 2012]. In contrast to those approaches, we are not trying to establish a type discipline on ClosLang, nor do our optimisations affect the global state. Instead, our logical relation follows the spirit of our previous untyped logical relation for CBV-λ calculus [Owens et al. 2016]. We enhance it to support ClosLang and its multiple argument functions with Curried semantics, and we use it to verify the optimisations that introduce them.

In particular, our relation does not approximate contextual equivalence — and it is not symmetric — in the following case. For expressions on the left of the relation, those that result in a TypeErr are related to everything. We know that the compiler will never be given such a program (by type soundness), and so we do not want to verify the compiler in that impossible case. For example, the compiler can assume that all applications are of functions: no extra checks are required at run time, and we make no attempt to analyse the IL to prove that locally.

We require the step index to be the same for both sides of the relation: expressions that decrement the clock by different amounts are not related. This has two immediate benefits. Equi-termination of two related expressions is immediate, and it is easy to prove that the relation is transitive. With the usual approach to logical relations, one shows equi-termination by demonstrating that two expressions are in the relation, and also in its inverse, increasing the proof effort. In general, step-indexing and transitivity do not normally mesh easily in a direct-style logical relation. For example, Ahmed [2006] was only able to prove transitivity of the step-indexed relation of Appel and

[7] Thus, we are not forced to use sophisticated logical relations for separate compilation or multiple languages [Ahmed 2015; Devriese et al. 2016; Hur and Dreyer 2011; Neis et al. 2015; New et al. 2016]. Nor would they be of much help here, since none of CakeML's compilation steps from source to ClosLang, nor from ClosLang to machine code, have correctness theorems that support a compositional notion of separate compilation.

McAllester [2001] by adding static typing, but she did not find a counter-example to the original's transitivity.[8]

The logical relation intuitively says that two lists of expressions are related if, when evaluated starting from related states and environments, the results are related.

$$\begin{array}{l}\mathsf{exp\_rel}\ w\ es_1\ es_2\ \Longleftrightarrow\\ \quad \forall\, i\ env_1\ env_2\ s_1\ s_2\ j.\\ \quad\quad (\mathsf{state\_rel}\ i\ w\ s_1\ s_2 \wedge \mathsf{all2}\ (\mathsf{val\_rel}\ i\ w)\ env_1\ env_2) \wedge\\ \quad\quad j \leq i \Rightarrow\\ \quad\quad \mathsf{res\_rel}\ w\ (\mathsf{evaluate}\ (es_1,env_1,s_1\ \mathsf{with\ clock}\ :=\ j))\\ \quad\quad\quad (\mathsf{evaluate}\ (es_2,env_2,s_2\ \mathsf{with\ clock}\ :=\ j))\end{array}$$

Figure 5 shows selected cases of the relations for values and results. The state relation is straightforward and omitted, since our optimisations do not affect the state. The parameter w allows the state relation to constrain the max_app field of the state to be consistent with the value used by the compiler.

The results that res_rel relates contain the resulting state as well as the actual result of the computation. Rval results are related if the values and states are related, and also the two clocks must be the same. Timeout results are related if the states are related; the clocks are both 0, or else the timeouts could not have occurred. A TypeErr result on the left is related to anything on the right.

The cases for values are straightforward, except for the closure and recursive closure cases. Those essentially say that two closures are related if, when you apply them to related arguments in related states, you get related results. That implication must hold for all smaller step indices j. The complication arises from the many different cases: each of cl and cl' could be a Closure or a Recclosure. Additionally, the arguments could separately lead to over or under-application for cl and cl'. Since one of our goals is arity changing, we cannot require related functions to have the same arity.

Because we are applying the two closures to an arbitrary number of related arguments, the relation needs to make some of the same calculations as the semantics, using dest_closure to work out what the application is, and to then call evaluate_ev, which can evaluate either a partial application Val, or a full application Exp1, with additional logic to keep the clock and step index in sync.

We prove that the relation is reflexive and transitive, and that related expressions have the same semantics in terms of observable traces of FFI calls.

$$\begin{array}{l}\vdash \mathsf{exp\_rel}\ w\ es_1\ es_2 \wedge \mathsf{exp\_rel}\ w\ es_2\ es_3 \Rightarrow \mathsf{exp\_rel}\ w\ es_1\ es_3\\ \vdash \mathsf{exp\_rel}\ w\ es\ es\\ \vdash \mathsf{exp\_rel}\ w\ es_1\ es_2 \wedge (\forall\, i.\ \mathsf{state\_rel}\ i\ w\ s_1\ s_2) \wedge\\ \quad \mathsf{semantics}\ [\,]\ s_1\ es_1 \neq \mathsf{Fail} \Rightarrow\\ \quad \mathsf{semantics}\ [\,]\ s_1\ es_1 = \mathsf{semantics}\ [\,]\ s_2\ es_2\end{array}$$

4.2 Custom Syntactic Relations

The second approach used for verification of optimisation passes is to write custom, *syntactic* relations that relate states, values and expressions in ways appropriate to the optimisation at

[8] Bi-orthogonal logical relations are automatically complete wrt contextual equivalence, and hence automatically transitive. However, we prefer the direct formulation of the relation, since it more closely matches our semantics, and thus avoids introduction of reasoning about continuations.

```
res_rel w (Rval vs,s) x ⟺
 ∃ vs′ s′.
   x = (Rval vs′,s′) ∧ all2 (val_rel s.clock w) vs vs′ ∧
   state_rel s.clock w s s′ ∧ s.clock = s′.clock

res_rel w (Timeout,s) x ⟺
 ∃ s′. x = (Timeout,s′) ∧ state_rel s.clock w s s′

res_rel w (TypeErr,s) x ⟺ true

evaluate_ev i (Exp1 loc_opt e env vs dec) s =
 if dec - 1 ≤ i then
   case evaluate ([e],env,s with clock := i - (dec - 1)) of
     (Rval [f],s1) ⇒ evaluate_app loc_opt f vs s1 | res ⇒ res
 else (Timeout,s with clock := 0)

exec_rel i w (x,s) (x′,s′) ⟺
 ∀ i′.
   i′ ≤ i ⇒
   res_rel w (evaluate_ev i′ x s) (evaluate_ev i′ x′ s′)

is_closure cl ⇒
(val_rel i w cl cl′ ⟺
  is_closure cl′ ∧ check_closures cl cl′ ∧
  ∀ j vs1 vs2 s1 s2 loc dc1.
    j < i ∧ state_rel j w s1 s2 ∧ vs1 ≠ [ ] ∧
    dest_closure w loc cl vs1 = Some dc1 ∧
    all2 (val_rel j w) vs1 vs2 ⇒
    ∃ dc2.
      dest_closure w loc cl′ vs2 = Some dc2 ∧
      let E1 =
          case dc1 of
            Partial_app b1 ⇒ (Val b1 (len vs1),s1)
          | Full_app b1 env1 rem_vs1 ⇒
              (Exp1 loc b1 env1 rem_vs1
                (len vs1 - len rem_vs1),s1);
        E2 =
          case dc2 of
            Partial_app b2 ⇒ (Val b2 (len vs2),s2)
          | Full_app b2 env2 rem_vs2 ⇒
              (Exp1 loc b2 env2 rem_vs2
                (len vs2 - len rem_vs2),s2)
      in
        exec_rel j w E1 E2)
```

Fig. 5. Relating values (selected clauses)

hand. Moreover, these relations bottom out with references to the optimisation function itself. For example, the relation behind the KNOWN optimisation relates expressions thus:

$$\begin{aligned}&\mathsf{exp\_rel_k}\ as\ g\ e_1\ e_2 \iff \\ &\quad \exists\, g_1\ g_2\ apx.\ g_2 \preccurlyeq g \wedge \mathsf{known}\ [e_1]\ as\ g_1 = ([(e_2,apx)],g_2)\end{aligned}$$

This ties the relation to the particular optimisation (here, the function known), but allows for easy optimisation-specific flexibility. Here this flexibility allows for the addition of extra parameters (g and as here) that would be hard to handle within the general, but relatively simple, framework of the logical relation. Indeed, the flexibility is such that KNOWN's relation on expressions has two extra parameters, but the value relation can have just one: see Figure 7 and the associated discussion for more on this.

Pragmatically, these syntactic relations are usually easy to write (handling the extra parameters, if any, is where any intellectual challenge arises), and easy to work with when it comes to proof. Proofs that evaluation moves related arguments to related results are simply by induction over the recursion pattern induced by the definition of evaluate. Proofs at scale such as these are tedious because of the numerous cases, but intellectually straightforward.

4.3 Ticks and Clocks

The original CakeML [Kumar et al. 2014] paper advocated using clocked evaluation to prove that the compiler preserves diverging executions. More recently Owens et al. [2016] showed how to apply similar techniques when faced with possibly diverging computations that perform I/O (on a simple while loop language). Siek [2013], Owens et al. [2016] and Amin and Rompf [2017] all advocate a similar approach for type soundness proofs.[9] We adopt that technique here, and so all of our optimisations and proofs have to manage the clock, whether they are being verified with the logical relation or a syntactic relation. Our experience — especially with the logical relation, where requiring related expressions to use the same index is a significant simplification — is that the management of the clock is well worth it to keep the reasoning all in the direction of compilation, as those papers advocated.

The inserted Ticks must eventually be removed, since they are not part of machine code execution. This requires a theorem going against the direction of compilation, stating that any observable result of a program with Ticks removed is also observable on that program with those Tick instructions re-inserted. Since CakeML uses functional big-step semantics throughout the compiler, Tick instructions are included in the ILs that target CLOSLANG and the BVL IL that CLOSLANG targets. Thus, all Ticks introduced by various compilation steps are passed along until the final stage of compilation where the Tick-removal theorem is proved once in the final compilation step, where the IL has simpler semantics.

5 INTRODUCING MULTI-ARGUMENT FUNCTIONS

The MULTI pass itself is straightforward; it simply finds iterated Fns (including in Letrecs) and Apps and combines them, up to the static maximum given by max_app. It assumes that all of the annotations in Fns, Letrecs and Apps are None, which is guaranteed by the phases of compilation that generate the initial CLOSLANG code.

We designed CLOSLANG so that this simple transformation is correct. In particular, the semantics supports applications where the function's arity is not the same as the number of arguments, so the optimisation can apply to functions and applications separately without statically coordinating them. Furthermore, right-to-left evaluation ordering means that the optimisation can be applied to

[9] In the ACL2 community defining recursive functions with clocks is a standard technique [Boyer and Moore 1996; Young 1989].

an App even when the sub-expressions are not known to be pure. This is because all arguments are evaluated before attempting to apply the function, which is what will happen after Multi combines the arguments in a single App. In contrast, left-to-right evaluation would apply the function as soon as enough arguments had been evaluated, possibly executing the function's body in-between evaluating the arguments.

To verify Multi, we first prove that uncombined Fn and App expressions are related by exp_rel to combined versions. These proofs are slightly tedious, but boring: unpacking the definitions and properly instantiating various quantifiers. The proof of the compiler itself is a simple induction on expressions, using the following two lemmas and the transitivity of the relation.

$$
\begin{array}{l}
\vdash \mathsf{len}\ es_1 = \mathsf{len}\ es_1' \wedge \mathsf{len}\ es_2 = \mathsf{len}\ es_2' \wedge \\
\quad \mathsf{len}\ es_1' + \mathsf{len}\ es_2' \leq w \wedge \mathsf{exp\_rel}\ w\ [f]\ [f'] \wedge \\
\quad \mathsf{exp\_rel}\ w\ es_1\ es_1' \wedge \mathsf{exp\_rel}\ w\ es_2\ es_2' \Rightarrow \\
\quad \mathsf{exp\_rel}\ w\ [\mathsf{App\ None\ (App\ None}\ f\ es_1)\ es_2] \\
\qquad [\mathsf{App\ None}\ f'\ (es_2' \mathbin{@} es_1')] \\
\\
\vdash argc_1 + argc_2 \leq w \wedge \mathsf{exp\_rel}\ w\ [e]\ [e'] \Rightarrow \\
\quad \mathsf{exp\_rel}\ w\ [\mathsf{Fn\ None\ None}\ argc_1\ (\mathsf{Fn\ None\ None}\ argc_2\ e)] \\
\qquad [\mathsf{Fn\ None\ None}\ (argc_1 + argc_2)\ e']
\end{array}
$$

6 FAST CALLS TO KNOWN FUNCTIONS

As described in §2.3, after multi-argument function introduction (Multi), we proceed through a series of compiler passes—Number, Known, Calls, and Remove—which aim to detect and convert applications that can be converted into fast C-style calls. In this section, we describe these compiler passes and their verification in more detail. The overall aim is to convey what these passes do and, briefly, how they are verified.

6.1 Introducing Code Locations

The Number pass is implemented by $\mathsf{compile}_n$, which annotates Fn and Letrec expressions with unique code locations. The correctness proof uses a syntactic relation, $\mathsf{v\_rel}_n$, whose definition on closures is as follows:

$$
\begin{array}{l}
\mathsf{v\_rel}_n\ mxapp\ (\mathsf{Closure}\ loc\_opt\ args_1\ env_1\ n\ e_1)\ v_2 \iff \\
\quad \exists\, args_2\ env_2\ nxt\ loc\_opt'. \\
\qquad v_2 = \mathsf{Closure}\ loc\_opt'\ args_2\ env_2\ n\ (\mathsf{hd}\ (\mathsf{snd}\ (\mathsf{compile}_n\ nxt\ [e_1]))) \wedge \\
\qquad \mathsf{all2}\ (\mathsf{v\_rel}_n\ mxapp)\ env_1\ env_2 \wedge \\
\qquad \mathsf{all2}\ (\mathsf{v\_rel}_n\ mxapp)\ args_1\ args_2 \wedge \neg \mathsf{has\_App\_Some}\ mxapp\ [e_1]
\end{array}
$$

To be related to the original closure, the new value, v_2, must also be a closure with the same number of arguments and with recursively related environments, and its body must be exactly the result of compiling the original closure's body (with some initial location *nxt*). We also exclude input values that contain App (Some _) expressions. The two degrees of freedom are the *nxt* location passed to the compiler, and the code location, *loc_opt′*, on the new closure. Because *loc_opt′* need not be related to *loc_opt*, $\mathsf{compile}_n$ is free to introduce new locations.

Theorem 6.1. *Correctness of the* Number *pass.*

$$
\begin{array}{l}
\vdash \text{evaluate } (es,env_1,s_1) = (res_1,t_1) \wedge \\
\quad \neg\text{has\_App\_Some } s_1.\text{max\_app } es \wedge \\
\quad \text{all2 } (\text{v\_rel}_\text{n}\ s_1.\text{max\_app})\ env_1\ env_2 \wedge \text{state\_rel}_\text{n}\ s_1\ s_2 \Rightarrow \\
\quad \exists\, res_2\ t_2. \\
\qquad \text{evaluate } (\text{snd } (\text{compile}_\text{n}\ next\ es),env_2,s_2) = (res_2,t_2) \wedge \\
\qquad \text{res\_rel}_\text{n}\ s_1.\text{max\_app}\ res_1\ res_2 \wedge \text{state\_rel}_\text{n}\ t_1\ t_2
\end{array}
$$

At crucial points (e.g., for a Fn expression) where the transformed code produces a closure with a newly allocated location and a transformed body, the syntactic relation applies; in other cases the inductive hypotheses apply straightforwardly. Ultimately, such theorems can be used in the overall CakeML compiler correctness theorem, where observational equivalence is required, because our value relations (v_rel_n) require equality on first-order values—including those in the FFI state.

Our syntactic relation works well for Number, where the overall shape of values during execution changes very little between the original and the transformed code, and there is little context required to recreate appropriate calls to the compiler in the value relation.

6.2 Detecting Known Functions

The Known optimisation is responsible for detecting the application of closures with known locations and arities to the expected number of arguments. It uses a simple data-flow analysis that infers approximations of input expressions. The approximation values are drawn from the `val_approx` type (see Figure 6). The Clos l n approximation specifies that the associated expression must be a closure located at location l and expecting n arguments. The Tuple and Int forms allow the inference of useful approximations when tuples are indexed. Other and Impossible are at the bottom and top of the "definedness" lattice. When, for example, an If-expression is analysed, the approximation of the whole is the meet of the approximations for the then- and else-branches.

Because top-level functions are stored in global reference cells, Known must be able to to track values in this (write-once) store. The known function (Figure 6) is thus passed an approximation map for its expression-arguments' globals, as well as a list of approximations for the expressions' free variables. It then returns a triple for each expression: an optimised form of the expression; an approximation for that expression; and an updated approximation map for the globals.

When known analyses the application of built-in operations to arguments, those arguments are analysed first. This means that when analysing the `foldl` example from §2.3, the argument to the SetGlobal call is analysed before the effect of the call is. This then means that the first time known sees the Global operation in the argument, it cannot tell how to approximate that given cell. In fact, compile_k calls known twice: the first traversal of the ClosLang syntax tree is only used to calculate a good approximation map for the globals.

Proving Known's correctness. Although the known function performs the analysis and transforms the program together, it is convenient to prove its correctness in two steps, beginning with a proof of the soundness of the approximation flow-analysis:

```
val_approx = Clos num num | Tuple num (val_approx list)
           | Int int | Other | Impossible

known [Fn loc_opt _ n e] vs g0 =
 let (e1,g1) = known [e] (genlist (λ _. Other) n @ vs) g0;
     (e',a1) = hd e1
 in
  ([(Fn loc_opt None n e',
     case loc_opt of None ⇒ Other | Some loc ⇒ Clos loc n)],g1)

known [App loc_opt f args] vs g0 =
 let (args',g1) = known args vs g0;
     (fl,g2) = known [f] vs g1;
     (f',apx) = hd fl;
     loc_opt' =
      case (loc_opt,apx) of
        (None,Clos l aty) ⇒
            if len args = aty then Some l else None
      | (None,_) ⇒ None
      | _ ⇒ loc_opt
 in
  ([(App loc_opt' f' (map fst args'),Other)],g2)
```

Fig. 6. The known approximation function (selected clauses) and the approximation type it uses. The first argument to known is the list of expressions to approximate; the second is a list of approximations for the free variables of the expressions; the third is a map specifying approximations for the state's global cells.

Theorem 6.2. *The* known *function correctly approximates the expressions it is applied to.*

$$
\begin{aligned}
\vdash\ & \text{known } es\ as\ g_1 = (eas, g_2) \wedge \text{distinct\_globals } es \wedge \\
& \text{all2 val\_approx\_val } as\ env \wedge g_1 \preccurlyeq g_2 \wedge g_2 \preccurlyeq g \wedge \\
& \text{state\_globals\_approx } s_1\ g \wedge \text{ssgc\_free } s_1 \wedge \\
& \text{all vsgc\_free } env \wedge \text{all esgc\_free } es \wedge \\
& \text{evaluate (map fst } eas, env, s_1) = (res, s_2) \Rightarrow \\
& \text{state\_globals\_approx } s_2\ g \wedge \\
& \forall vs.\ res = \text{Rval } vs \Rightarrow \text{all2 val\_approx\_val (map snd } eas)\ vs
\end{aligned}
$$

In other words, if known*'s result (i.e.,* map fst *eas) is executed inside an environment (env) that is correctly approximated by as, and in a state correctly approximated by the map g then the resulting state s will be correctly approximated by g as well, and if the execution resulted in values vs (as opposed to an exception being thrown, say), then the values vs are approximated by the approximations computed by* known *(i.e.,* map snd *eas).*

The proof is by structural induction over es.

The other preconditions characterise the situations where known will work. In particular, it is important that the global approximation map increases monotonically (we write $g_1 \preccurlyeq g_2$ when, viewing the maps as sets of pairs, g_1 is a subset of g_2). In order to guarantee this, there must only ever be one SetGlobal call for a particular location (distinct_globals *es*), and there must be no SetGlobal calls within closures (the various *x*sgc_free conditions).

$$
\begin{array}{l}
\text{v\_rel}_\text{k}\ g\ (\text{Closure}\ \mathit{loc\_opt}\ \mathit{vs}_1\ \mathit{env}_1\ n\ \mathit{bod}_1)\ v \iff \\
\quad \text{unannotated\_fv}\ [\mathit{bod}_1] \wedge \\
\quad \exists\, \mathit{vs}_2\ \mathit{env}_2\ \mathit{bod}_2\ \mathit{eapx}. \\
\qquad \text{all2}\ (\text{v\_rel}_\text{k}\ g)\ \mathit{vs}_1\ \mathit{vs}_2 \wedge \text{all2}\ (\text{v\_rel}_\text{k}\ g)\ \mathit{env}_1\ \mathit{env}_2 \wedge \\
\qquad \text{all2}\ \text{val\_approx\_val}\ \mathit{eapx}\ \mathit{env}_2 \wedge \\
\qquad \text{exp\_rel}_\text{k}\ (\text{genlist}\ (\lambda\ \_.\ \text{Other})\ n\ @\ \mathit{eapx})\ g\ \mathit{bod}_1\ \mathit{bod}_2 \wedge \\
\qquad v = \text{Closure}\ \mathit{loc\_opt}\ \mathit{vs}_2\ \mathit{env}_2\ n\ \mathit{bod}_2
\end{array}
$$

Fig. 7. Syntactic relation behind the verification of the Known optimisation (closure clause; the clause for Recclosure is analogous).

The next stage of the proof uses a custom, syntactic relation. As hinted earlier, the only interesting question is what to do with the two other parameters to the known function (the approximation of the free variable environment, and the approximation of the initial state). It turns out (Figure 7) that the globals map needs to be a parameter of the relation, but the environmental approximation can be existentially quantified where necessary (see the *eapx* variable in the figure). Also, similar to the v_rel_n relation, but unlike the logical relation, the v_rel_k relation relates closure environments pointwise. This is a hallmark of situations when a syntactic relation will be effective.

This then leads to

Theorem 6.3. *The* known *and* compile_k *functions are correct*

$$
\begin{array}{l}
\vdash \text{compile}_\text{k}\ b\ e_1 = e_2 \wedge \text{evaluate}\ ([e_1],[\,],s_1) = (\mathit{res}_1,t_1) \wedge \\
\quad \text{esgc\_free}\ e_1 \wedge \text{unannotated\_fv}\ [e_1] \wedge \text{state\_rel}_\text{k}\ b\ \emptyset\ s_1\ s_2 \wedge \\
\quad \text{state\_globals\_approx}\ s_1\ \emptyset \wedge \text{distinct\_globals}\ [e_1] \wedge \\
\quad \text{ssgc\_free}\ s_1 \Rightarrow \\
\quad \exists\, \mathit{res}_2\ t_2\ g. \\
\qquad \text{evaluate}\ ([e_2],[\,],s_2) = (\mathit{res}_2,t_2) \wedge \\
\qquad \text{res\_rel}_\text{k}\ b\ g\ (\mathit{res}_1,t_1)\ (\mathit{res}_2,t_2)
\end{array}
$$

The requirement state_globals_approx $s_1\ \emptyset$ *is equivalent to there being no defined globals, which is indeed the case when* Known *is applied. The* res_rel_k *relation requires, among other things, that the FFI call streams of the states in its result arguments are equal.*

6.3 Introducing Fast Calls

The Calls optimisation moves closed function bodies into the code table, and replaces each correct-arity application of such a function with an efficient C-style function call. Figure 8 shows the App and Fn clauses of calls, the main function implementing Calls. The idea is that if the body of a known function does not refer to any variables in the closure environment, the closure is unnecessary and can be omitted. See the examples in §2.3.

The calls function tracks the locations, *locs*, of functions with closed bodies. On the App (Some *loc*) side (application of a known function to the correct number of arguments), we produce a C-style call if the body is closed (elem *loc locs'*), and only execute the closure expression f if it has effects (according to a simple under-approximation that excludes the common case of Vars). The call is to location *loc* + 1, where a new code table entry containing the closed body and not expecting a closure argument will be created.

On the function-creation side, the body is optimistically transformed first and then we check whether it is closed by the n arguments. It is important that the closedness check is done on the

```
calls [App (Some loc) f es] (locs,code) =
 let (es,g) = calls es (locs,code);
     (e1,locs',code') = calls [f] g
 in
  if elem loc locs' then
   if pure f then ([Call (len es) (loc + 1) es],locs',code')
   else
    ([Let (es @ [hd e1])
       (Call (len es) (loc + 1) (genlist Var (len es)))],
     locs',code')
  else ([App (Some loc) (hd e1) es],locs',code')

calls [Fn (Some loc) None n bod] (locs,code) =
 let (e1,locs',code') = calls [bod] (insert loc locs,code)
 in
  if closed (Fn (Some loc) None n (hd e1)) then
   ([Fn (Some loc) None n
      (Call 0 (loc + 1) (genlist Var n))],locs',
    (loc + 1,n,hd e1)::code')
  else
   let (e1',g') = calls [bod] (locs,code)
   in
    ([Fn (Some loc) None n (hd e1')],g')
```

Fig. 8. The Calls optimisation (selected clauses). The first argument is the list of expressions to optimise; and the second is a pair of the set of locations (*locs*) moved (or to be moved) into the code table, and the code table entries (*code*) collected so far.

transformed code, because Calls can remove free variables, namely, those that appear only in the function position of applications to known functions. If the check succeeds, any known applications to *loc* will be transformed to direct Calls as described above, and the Fn expression is transformed to one that immediately calls location *loc* + 1.

Proving Calls's correctness. We use a syntactic relation on values before and after the Calls optimisation. Figure 9 shows how our relation is defined on non-recursive closures. The compiler builds up state as it runs, which we pair together as g = (*locs*,*code*) (locations of known functions with closed bodies, and newly generated code-table entries). The interesting question is how the syntactic relation should handle this state when calling the compiler to relate closure bodies. It turns out we can leave the states passed to and returned by the compiler existentially quantified, but parameterise the relation by a state g representing some *extension* of the compiler's result state (we use $g_1 \preccurlyeq_c g_2$ to capture how the compiler extends its state). Additionally, we need a parameter to track the locations of non-closed functions that have been seen; we use l as a fixed *superset* of these locations. One trick in the correctness proof was to use these "final" values of g and l, because they stay fixed when applying inductive hypotheses.

Following the syntactic approach, we usually want to relate closure environments pointwise. For Calls, this works for the closures' held arguments ($args_1$ and $args_2$), but the closure environments (env_1 and env_2) require a custom relation $\text{env\_rel}_c\ g\ l$ that depends on the code (bod_2) in the second closure. The reason is that a closure can end up with a smaller environment after optimisation: the

$$
\begin{array}{l}
\mathsf{v\_rel_c}\ g\ l\ (\mathsf{Closure}\ loc\_opt\ args_1\ env_{v_1}\ n\ bod_1)\ v \iff \\
\quad \exists\, loc\ args_2\ env_{v_2}\ bod_2. \\
\qquad \mathsf{exp\_rel_c}\ g\ l\ loc\ [(n,bod_1)]\ [(n,bod_2)] \wedge \\
\qquad v = \mathsf{Closure}\ (\mathsf{Some}\ loc)\ args_2\ env_{v_2}\ n\ bod_2 \wedge \\
\qquad loc\_opt = \mathsf{Some}\ loc \wedge \mathsf{all2}\ (\mathsf{v\_rel_c}\ g\ l)\ args_1\ args_2 \wedge \\
\qquad \mathsf{env\_rel_c}\ g\ l\ env_{v_1}\ env_{v_2}\ 0\ [(n,bod_2)]
\end{array}
$$

Fig. 9. Syntactic relation on closures for Calls

Call that replaces the body of an optimised Fn expression restricts the environment of any closures created in the original function body. Our custom environment relation only requires closure environments to match on free variables (i.e., not counting the arguments and other mutually recursive closures) in the body.

The main inductive proof for Calls establishes this:

Theorem 6.4. *The* calls *function is correct.*

$$
\begin{array}{l}
\vdash \mathsf{evaluate}\ (es_1,env_{v_1},s_1) = (res_1,t_1) \wedge res_1 \neq \mathsf{TypeErr} \wedge \\
\quad \mathsf{calls}\ es_1\ g_1 = (es_2,g_2) \wedge \mathsf{annotated\_loc}\ es_1 \wedge \\
\quad \mathsf{unannotated\_fv}\ es_1 \wedge \mathsf{all}\ (\mathsf{wfv}\ g\ l)\ env_{v_1} \wedge \mathsf{wfv\_state}\ g\ l\ s_1 \wedge \\
\quad \mathsf{wfg}\ g_1 \wedge \mathsf{distinct\_locs}\ es_1 \wedge \\
\quad \mathsf{disjoint}\ ((\lambda\, l.\ l+1)\ \text{``}\ \mathsf{code\_locs}\ es_1) \\
\qquad (\mathsf{set}\ (\mathsf{map}\ \mathsf{fst}\ (\mathsf{snd}\ g_1))) \wedge g_2 \preccurlyeq_c g \wedge \\
\quad \mathsf{code\_locs}\ es_1 \setminus \mathsf{fst}\ g_2 \subseteq l \wedge \mathsf{disjoint\_locs}\ l\ (\mathsf{fst}\ g) \wedge \\
\quad \mathsf{wfg}\ g \Rightarrow \\
\quad \exists\, ck\ res_2\ t_2. \\
\qquad \mathsf{wfv\_res}\ g\ l\ res_1 \wedge \mathsf{wfv\_state}\ g\ l\ t_1 \wedge \\
\qquad (\mathsf{env\_rel_c}\ g\ l\ env_{v_1}\ env_{v_2}\ 0\ (\mathsf{map}\ (\lambda\, x.\ (0,x))\ es_2) \wedge \\
\qquad\ \mathsf{state\_rel_c}\ g\ l\ s_1\ s_2 \wedge \mathsf{code\_includes}\ (\mathsf{snd}\ g_2)\ s_2.\mathsf{code} \Rightarrow \\
\qquad\ \mathsf{evaluate}\ (es_2,env_{v_2},s_2\ \mathsf{with\ clock} :=\ s_2.\mathsf{clock} + ck) = \\
\qquad\quad (res_2,t_2) \wedge \mathsf{state\_rel_c}\ g\ l\ t_1\ t_2 \wedge \\
\qquad\ \mathsf{res\_rel_c}\ g\ l\ res_1\ res_2)
\end{array}
$$

Several of the hypotheses concern code locations: the input expressions (es_1) and the code table built up so far ($\mathsf{snd}\ g_1$) must all have distinct locations. The wfg predicate maintains an invariant on the compiler's state: the domain of the code table must be one plus each of the already-optimised locations ($\mathsf{fst}\ g_1$).

The wfv predicate is defined so that the following lemma holds:

$$\vdash \mathsf{wfv}\ g\ l\ v_1 \Rightarrow \exists\, v_2.\ \mathsf{v\_rel_c}\ g\ l\ v_1\ v_2$$

This is used in the App case of the induction, where calls does not recurse on all subexpressions (recall that pure expressions denoting known functions can be dropped) so the inductive hypothesis cannot be used directly, but we still need to know the correct function body is in the code table.

6.4 Removing Dead Code

The Remove pass replaces unused bindings (Let and Letrec) with dummy bindings to constants, and is aimed at cleaning up after the Calls pass which may remove references to known functions. The main function, remove, does a bottom-up pass of the expression collecting free variables, and replaces unreferenced bindings of pure expressions.

Remove's proof uses the logical relation. The key lemma is a reflexivity lemma under the weakened assumption that the environments are not necessarily related on elements that are not free in the expression to be related.

$$\begin{aligned}\vdash\ &\text{state\_rel}\ i\ w\ s_1\ s_2 \wedge j \leq i \wedge \text{fv}\ es \subseteq kis \wedge \text{unannotated\_fv}\ es \wedge \\ &\text{all2i}\ (\lambda\, k\ v_1\ v_2.\ k \in kis \Rightarrow \text{val\_rel}\ i\ w\ v_1\ v_2)\ env_1\ env_2 \Rightarrow \\ &\text{res\_rel}\ w\ (\text{evaluate}\ (es,env_1,s_1\ \text{with clock}\ :=\ j)) \\ &\quad (\text{evaluate}\ (es,env_2,s_2\ \text{with clock}\ :=\ j))\end{aligned}$$

7 COMPILING OUT OF CLOSLANG

ClosLang is compiled to BVL. The semantics of BVL is the same as ClosLang except that instead of App, Fn, or Letrec constructs, it uses code-pointer values and Calls. The overall compilation technique is to move all code into the code table, and implement closures as heap allocated closure records pairing a code pointer with an environment. We use a flat closure representation, so the environment should only include values for variables that occur free in the body.

For applications to unknown functions (i.e., App None), we use an Eval/Apply strategy [Marlow and Peyton Jones 2006], where each application site statically knows how many arguments it has, and dynamically checks the closure record's arity, and either simply calls it, branches to a library function to allocate a wrapper closure, or branches to a driver loop that performs multiple applications.

A syntactic relation between ClosLang and BVL values captures the encoding of closures, and partially applied closures, into BVL. The proof is straightforward, but tedious because it is impossible to ensure that the clocks tick the same amount. This is because the number of calls made by the Eval/Apply library functions in BVL can vary dynamically, depending on the closure values passed to them. Instead, the BVL side must be allowed to Tick an existentially quantified number of times more.

8 BENCHMARKS

Figure 10 shows the performance improvements of our optimisations on a series of micro-benchmarks. The benchmarks were compiled using a bootstrapped version of the CakeML compiler that includes the CakeML basis library. Note that the optimisations are cumulative: e.g., we do not apply Calls without Multi and Known first. Briefly, *reverse* and *foldl* apply the examples described in §2.3, *fib* computes an exponential version of Fibonacci, *btree* and *qsort* sort lists of numbers using a binary tree and quicksort respectively, *queue* runs a series of push and pop operations on a functional queue implementation. We also include benchmarks that use the imperative features of CakeML: *qsortimp* runs quicksort using in-place updates to an array, and *nqueens* uses exception backtracking to solve the N-queens problem.

The first chart shows that Multi is the optimisation that makes the biggest difference; Known is also significant. Note that even though the *fib* function itself is not affected by Multi, the benchmark still benefits from it because of knock-on effects from later passes, such as function in-lining. Additionally, the default value of max_app is set to 4, which prevents Multi from being applied on one of the core calls in the *qsortimp* benchmark. The *qsortimp'* benchmark has this parameter set to 5 instead.

The second chart shows that we are in the ball-park of other modern ML compilers, unlike CakeML version 1, which was slower than interpreted OCaml.[10] Our design choice of right-to-left evaluation order makes Multi easy to apply, as described in §5. OCaml benefits from the same due to its loose specification of evaluation order. In contrast, SML compilers must do more analysis

[10] Other optimisations in CakeML version 2, including register allocation, also contribute to the improvement over version 1.

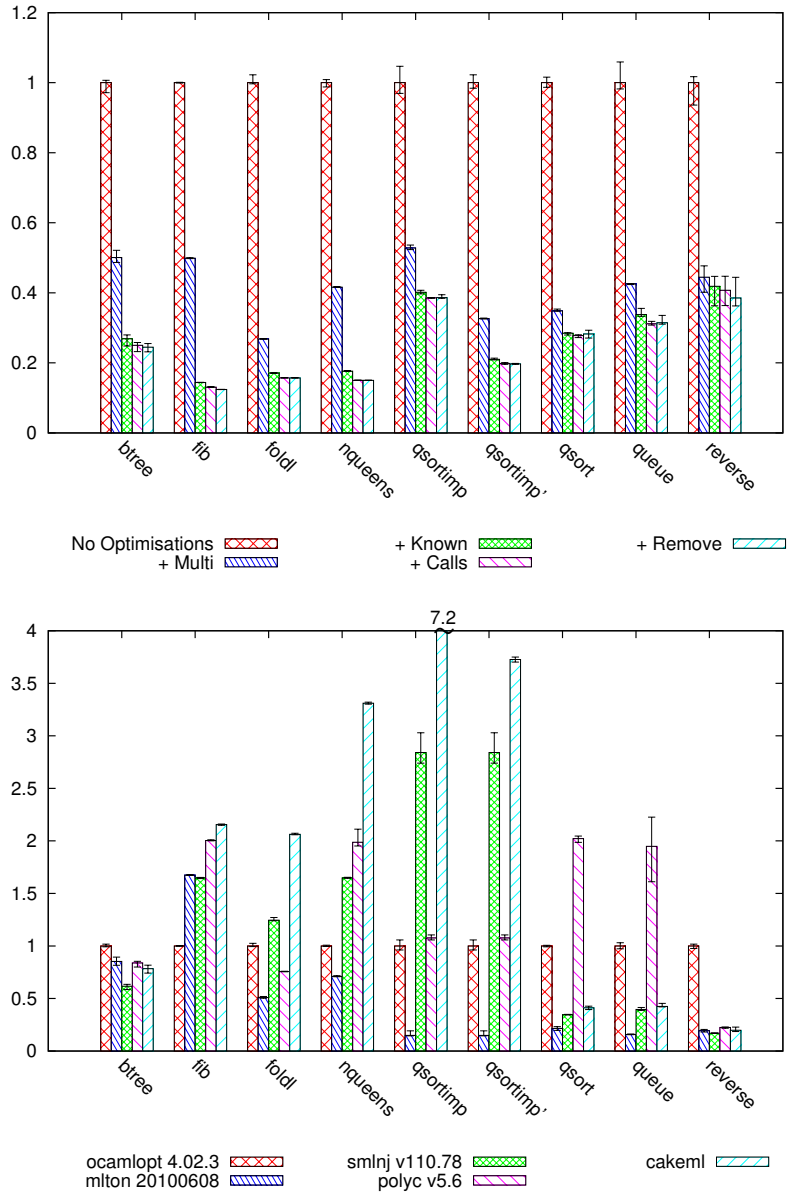

Fig. 10. (Top) Average execution time of the optimised benchmarks relative to the baseline (No Optimisations). The CLOSLANG optimisations are applied additively from left to right. (Bottom) Comparison of average execution times across ML implementations, relative to OCaml. The error bars show the maximum/minimum times measured over 10 executions.

of the code before they can apply MULTI-like optimisations, since the standard requires left-to-right evaluation order. Note also that the CakeML compiler uses bignum arithmetic for all of its computations, while most of the other compilers (except Poly/ML) default to fixed sized integers.

9 RELATED WORK

As mentioned in §1, neither previous versions of CakeML [Kumar et al. 2014], nor Pilsner [Neis et al. 2015] support special semantics for curried functions, nor do they optimise calls to known (but not inlined) functions. This is true for Lambda Tamer [Chlipala 2010] as well, although it uses higher-order abstract syntax instead of de Bruijn indices and shifting. As far as we are aware, no other verified compiler supports closure conversion for mutually recursive functions via heap allocation. Note that these projects' goals differ from ours: Pilsner's was to demonstrate compositional verification using parametric simulations; Lambda Tamer emphasised proof technology.

Dargaye and Leroy [2009] verify higher-order uncurrying using a step-indexed logical relation. Our approach appears to be slightly more flexible in that we always uncurry applications to unknown functions (e.g., the function passed into `map2`), and our semantics supports mismatched applications, whereas they require `curry` and `uncurry` coercions to match everything up (static uncurrying). Their logical relation differs from ours by using soft types to track arities, which meshes well with coercion insertion.

Breitner [2015] verifies an arity analysis that attempts to η-expand definitions to the number of arguments that they will be called with. This differs from and is complementary to our optimisations: it is trying to recognise multiple argument functions that are not simple iterations of `fn`. For example, transforming the function at the end of §2.2 into the following:

```
fn x => fn y => fn new_param => let
  val z = x + y
in (fn a => a + z) new_param end
```

Breitner [2015] is working in the context of Haskell, and verifies a sophisticated cardinality analysis to ensure that the eta expanded definition is not called more than once, lest extra thunks be allocated. This example of verifying a performance rather than soundness property in Haskell would become a matter of semantic correctness in our setting, where the definition to be expanded might have side effects.

Breitner [2015] notes that because they are working on a model of a single optimisation for a simple calculus (and necessarily so given the complexity of GHC), the real-world counterpart of their verified optimisation still had a bug due to complex interactions with the rest of the compiler that was not modelled by their calculus. However, their modelling efforts did lead them to another bug in the real optimiser. Our approach of working in the context of a fully verified compiler ensures that we do not have any semantic mismatches, but it also makes it more difficult to verify more complex analyses: none of the ILs or their semantics can be easily tailored to the optimisation in question. The approaches are complementary.

Dargaye and Leroy [2009] also claim that no practical compiler does higher-order uncurrying due to the difficulty of choosing where to insert the coercions. We believe from reading the OCaml native code compiler's sources, that our solution is similar to OCaml's. In particular, OCaml wraps closures that are partially applied and generates library functions to handle various application/arity mismatches.

The Jitawa verified LISP system [Myreen and Davis 2011] supports first-order functions that take multiple simultaneous arguments. Its calls are similar to those in CakeML's BVL language.

Untyped logical relations appear to be rare, but they are used on occasion. Pitts [1994] uses similar techniques to prove adequacy of a denotational semantics. Acar et al. [2008] defined an untyped logical relation to reason about cyclic structures in the heap, which requires a more sophisticated treatment of references than we support.

10 CONCLUSION

We have added state-of-the-art compilation for curried, higher-order functions to CakeML. Our contribution is based on three things: designing an IL and its semantics to provide the right invariants for the optimisations to be verified, splitting the optimisations themselves into several passes that each perform a single task, and defining relations on values that are either flexible (the logical relation), or tailored to the proof at hand (each syntactic relation). We propose the following two rules-of-thumb: first, keep the logical clocks the same between the optimised and unoptimised versions, inserting clock-decrementing Tick instructions where necessary; second, consider a syntactic relation when closure environments are related pointwise or in other simple ways, and a more sophisticated logical relation otherwise.

Building on this basis, our contribution is a demonstration that realistic and performance-critical optimisations for functional programming can be verified in the setting of a complete end-to-end compiler. Lastly, even though CakeML's handling of calls and closures is considerably improved, there is room for further improvement in other parts of the compiler. In particular, adding lambda lifting or inlining would make the current optimisations apply in more situations.

ACKNOWLEDGMENTS

The first author was partly supported by EPSRC Grant EP/N028759/1, UK; the fourth author was partly supported by the Swedish Foundation for Strategic Research and the Swedish Research Council; and the fifth author was supported by an A*STAR National Science Scholarship (PhD), Singapore.

REFERENCES

Umut A. Acar, Amal Ahmed, and Matthias Blume. 2008. Imperative self-adjusting computation. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008*. ACM, 309–322. https://doi.org/10.1145/1328438.1328476

Amal Ahmed. 2015. Verified Compilers for a Multi-Language World. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 15–31. https://doi.org/10.4230/LIPIcs.SNAPL.2015.15

Amal J. Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006 (Lecture Notes in Computer Science)*, Peter Sestoft (Ed.), Vol. 3924. Springer, 69–83. https://doi.org/10.1007/11693024_6

Nada Amin and Tiark Rompf. 2017. Type Soundness Proofs with Definitional Interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, 666–679. https://doi.org/10.1145/3009837.3009866

Andrew W. Appel. 1992. *Compiling with Continuations*. Cambridge University Press.

Andrew W. Appel and David A. McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.* 23, 5 (2001), 657–683. https://doi.org/10.1145/504709.504712

Nick Benton and Chung-Kil Hur. 2009. Biorthogonality, step-indexing and compiler correctness. In *Proceeding of the 14th ACM SIGPLAN International Conference on Functional Programming, ICFP 2009*. ACM, 97–108. https://doi.org/10.1145/1596550.1596567

R.S. Boyer and J S. Moore. 1996. Mechanized Formal Reasoning about Programs and Computing Machines.. In *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*. MIT Press.

Joachim Breitner. 2015. Formally Proving a Compiler Transformation Safe. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15)*. ACM, 35–46. https://doi.org/10.1145/2804302.2804312

Adam Chlipala. 2010. A Verified Compiler for an Impure Functional Language. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*. ACM, 93–106. https://doi.org/10.1145/1707801.1706312

Zaynah Dargaye and Xavier Leroy. 2009. A verified framework for higher-order uncurrying optimizations. *Higher-Order and Symbolic Computation* 22, 3 (2009), 199–231. https://doi.org/10.1007/s10990-010-9050-z

Dominique Devriese, Marco Patrignani, and Frank Piessens. 2016. Fully-abstract compilation by approximate back-translation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016.*

ACM, 164–177. https://doi.org/10.1145/2837614.2837618

Derek Dreyer, Georg Neis, and Lars Birkedal. 2012. The impact of higher-order state and control effects on local relational reasoning. *J. Funct. Program.* 22, 4-5 (2012), 477–528. https://doi.org/10.1017/S095679681200024X

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI '93)*. ACM, 237–247. https://doi.org/10.1145/155090.155113

Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. 2017. Verified Characteristic Formulae for CakeML. In *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017*, Hongseok Yang (Ed.). Springer, 584–610. https://doi.org/10.1007/978-3-662-54434-1_22

Chung-Kil Hur and Derek Dreyer. 2011. A Kripke logical relation between ML and assembly. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011*. ACM, 133–146. https://doi.org/10.1145/1926385.1926402

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *POPL '14: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 179–191. https://doi.org/10.1145/2535838.2535841

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

Simon Marlow and Simon Peyton Jones. 2006. Making a fast curry: Push/Enter vs. Eval/Apply for higher-order languages. *J. Funct. Program.* 16, 4-5 (2006), 415–449. https://doi.org/10.1017/S0956796806005995

Magnus O. Myreen and Jared Davis. 2011. A Verified Runtime for a Verified Theorem Prover. In *Interactive Theorem Proving - Second International Conference, ITP 2011 (Lecture Notes in Computer Science)*, Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk (Eds.), Vol. 6898. 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* 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 *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015*. 166–178. https://doi.org/10.1145/2784731.2784764

Max S. New, William J. Bowman, and Amal Ahmed. 2016. Fully abstract compilation via universal embedding. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016*. ACM, 103–116. https://doi.org/10.1145/2951913.2951941

Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-step Semantics. In *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016 (Lecture Notes in Computer Science)*, Peter Thiemann (Ed.), Vol. 9632. Springer, 589–615. https://doi.org/10.1007/978-3-662-49498-1_23

James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-language Semantics. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014 (Lecture Notes in Computer Science)*, Zhong Shao (Ed.), Vol. 8410. Springer, 128–148. https://doi.org/10.1007/978-3-642-54833-8_8

Andrew M. Pitts. 1994. *Computational adequacy via 'mixed' inductive definitions.* Springer Berlin Heidelberg, 72–82. https://doi.org/10.1007/3-540-58027-1_3

Jeremy Siek. 2013. Type Safety in Three Easy Lemmas. (2013). http://siek.blogspot.com/2013/05/type-safety-in-three-easy-lemmas.html.

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2016. A New Verified Compiler Backend for CakeML. In *ICFP '16: Proceedings of the 21th ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 60–73. https://doi.org/10.1145/2951913.2951924

Yong Kiam Tan, Scott Owens, and Ramana Kumar. 2015. A Verified Type System for CakeML. In *27th Symposium on the Implementation and Application of Functional Programming Languages, IFL 2015*. ACM Press. https://doi.org/10.1145/2897336.2897344

William D. Young. 1989. A Mechanically Verified Code Generator. *J. Autom. Reasoning* 5, 4 (1989), 493–518. https://doi.org/10.1007/BF00243134