ZU064-05-FPR paper 19 November 2018 11:55

The Verified CakeML Compiler Backend

Yong Kiam Tan[1], Magnus O. Myreen[2], Ramana Kumar[3] Anthony Fox[4], Scott Owens[5], Michael Norrish[6]
[1]Carnegie Mellon University, USA
[2]Chalmers University of Technology, Sweden
[3]Data61, CSIRO / UNSW, Australia
[4]University of Cambridge, UK
[5]University of Kent, UK
[6]Data61, CSIRO / ANU, Australia

Abstract

The CakeML compiler is, to the best of our knowledge, the most realistic verified compiler for a functional programming language to date. The architecture of the compiler, a sequence of intermediate languages through which high-level features are compiled away incrementally, enables verification of each compilation pass at an appropriate level of semantic detail. Parts of the compiler's implementation resemble mainstream (unverified) compilers for strict functional languages, and it supports several important features and optimisations. These include efficient curried multi-argument functions, configurable data representations, efficient exceptions, register allocation, and more. The compiler produces machine code for five architectures: x86-64, ARMv6, ARMv8, MIPS-64, and RISC-V. The generated machine code contains the verified runtime system which includes a verified generational copying garbage collector and a verified arbitrary precision arithmetic (bignum) library.

In this paper we present the overall design of the compiler backend, including its 12 intermediate languages. We explain how the semantics and proofs fit together, and provide detail on how the compiler has been bootstrapped inside the logic of a theorem prover. The entire development has been carried out within the HOL4 theorem prover.

1 Introduction

Optimising compilers are complex pieces of software and, as such, errors are almost inevitable in their implementations, as Yang *et al.* (2011) showed with systematic experiments. The only compiler Yang *et al.* did not find flaws in was the verified part of the CompCert C compiler (Leroy, 2009). The CompCert project has shown that it is possible to formally verify a realistic, optimising compiler, and thereby encouraged significant interest in compiler verification. In fact, much of this interest has gone into extending or building on CompCert itself (Stewart *et al.*, 2015; Sevcík *et al.*, 2013; Mullen *et al.*, 2016).

Verified compilers for functional languages have not previously reached the same level of realism, even though there have been many successful projects in this space, e.g. the compositional Pilsner compiler (Neis *et al.*, 2015) and the previous CakeML compiler which is able to bootstrap itself (Kumar *et al.*, 2014).

This paper is the extended version of our earlier conference paper (Tan *et al.*, 2016). We present the most realistic, to our knowledge, verified compiler for a functional programming language to date:

- The backend starts from a fully featured source language, namely CakeML, which includes user-defined modules, signatures, user-defined exceptions and datatypes, mutually recursive functions, pattern matching, references, mutable arrays, immutable vectors, strings, and a foreign function interface.
- Compilation passes through the usual compilation phases, including register allocation via Iterated Register Coalescing. The backend uses 12 intermediate languages that together allow implementation of optimisations at convenient levels of abstraction. It uses efficient, configurable data representations and properly compiles the call stack into memory, including the ML-style exception mechanism. The final memory model does not distinguish between pointers and machine words.
- The output is concrete machine code for five real machine languages, including both 32-bit and 64-bit architectures, and both big- and little-endian architectures.
- We bootstrap the compiler within the logic so that it can be used both inside and outside of the logic. Inside the logic, the compiler backend takes source language abstract syntax trees (ASTs) as input. Outside of the logic, it takes concrete syntax as input, which it parses, performs type inference on and then compiles using the compiler backend.

The compilation strategy is not the novelty here, and we freely take inspiration from existing compilers, including CompCert and OCaml. Our contribution here is the verification effort, and explaining how it affects the compiler's structure and vice versa.

Traditional compiler design is motivated by generated-code quality, compiler-user experience (especially compile times), and compiler-writer convenience. Designing a *verified* compiler is not simply a matter of taking an existing compiler and proving it correct while simultaneously fixing all its bugs. To start with, it is probably not written in the input language of a theorem proving system, but even if it could be translated into such a form, we would not expect to get very far in the verification effort. Although theoretically possible, verifying a compiler that is not designed for verification would be a prohibitive amount of work in practice.

To make the verification tractable, the compiler's design must also consider the compiler verifier. This means that the compiler's intermediate languages, including their semantics, need to be carefully constructed to support precise specification of tractable invariants to be maintained at each step of compilation. Of course, we cannot forgo the other design motivations completely, and our main contribution here is a design that allowed us to complete the verification while supporting both good quality code (for multiple targets) and the implementation of further optimisations in the future.

This paper aims to be a comprehensive presentation of how the design, implementation and verification of the CakeML compiler fit together to produce an end-to-end verified compiler with acceptable performance. This paper presents:

- the top-level design of the compiler and the structure of its implementation;

- the proof methodology used throughout the verification of the compiler backend;
- the top-level correctness theorem for the compiler;
- a run through of all of the main phases of the compiler, providing more detail on the phases that we suspect are interesting from a verification point of view;
- how the new compiler has been bootstrapped inside the logic of a theorem prover;
- benchmarking numbers that show how different optimisations impact the performance and also compare the CakeML compiler against other ML compilers.

All of our definitions and proofs are carried out in the HOL4 system (Slind & Norrish, 2008). The code is available at: `https://code.cakeml.org`.

2 Approach

In this section, we start with a brief overview of the compiler implementation including a summary of the main design decisions. We then describe the semantics of its intermediate languages, and the correctness proofs. Subsequent sections will expand on the details.

2.1 Compiler Implementation and Major Design Decisions

This paper describes what we call version 2 of the CakeML compiler, which is the latest version, at the time of writing. Version 2 uses 12 intermediate languages (ILs), as illustrated in Figure 1. The compiler backend starts from full CakeML, which includes modules, nested pattern matching, data types, references, arrays, and an I/O interface, and targets five real machine architectures. Each important step is separated into its own compiler pass, including closure conversion, concretisation of data representations, register allocation, concretisation of the stack, etc. The compiler uses a configuration record which specifies the data representation to use and details of the target architecture.

We note that the CakeML compiler continues to evolve and many details will change as the compiler improves. However, there are certain deep-rooted design decisions that are likely to persist. In what follows, we describe those major design decisions, while the rest of the paper describes the compiler as it is now.

The top-most design decision is about the purpose of the compiler. The purpose of the CakeML compiler is to be a verified compiler that is (P1:) as realistic as is practically possible, and (P2:) simple and tidy enough to be used as a platform for future research and student projects. In situations where P1 and P2 have been in conflict, e.g. when a modification was about to make the compiler more complicated in order to produce better code, we have tended to emphasise P1 over P2.

The following high-level design decisions have been made during the development of the compiler. Note that for some of these, namely D1, D3, D4, D5, we could have taken the decisions differently and the resulting compiler and its verification could possibly have been equally good overall — it would just have ended up with the implementation and verification complexities distributed differently. We motivate our choices below.

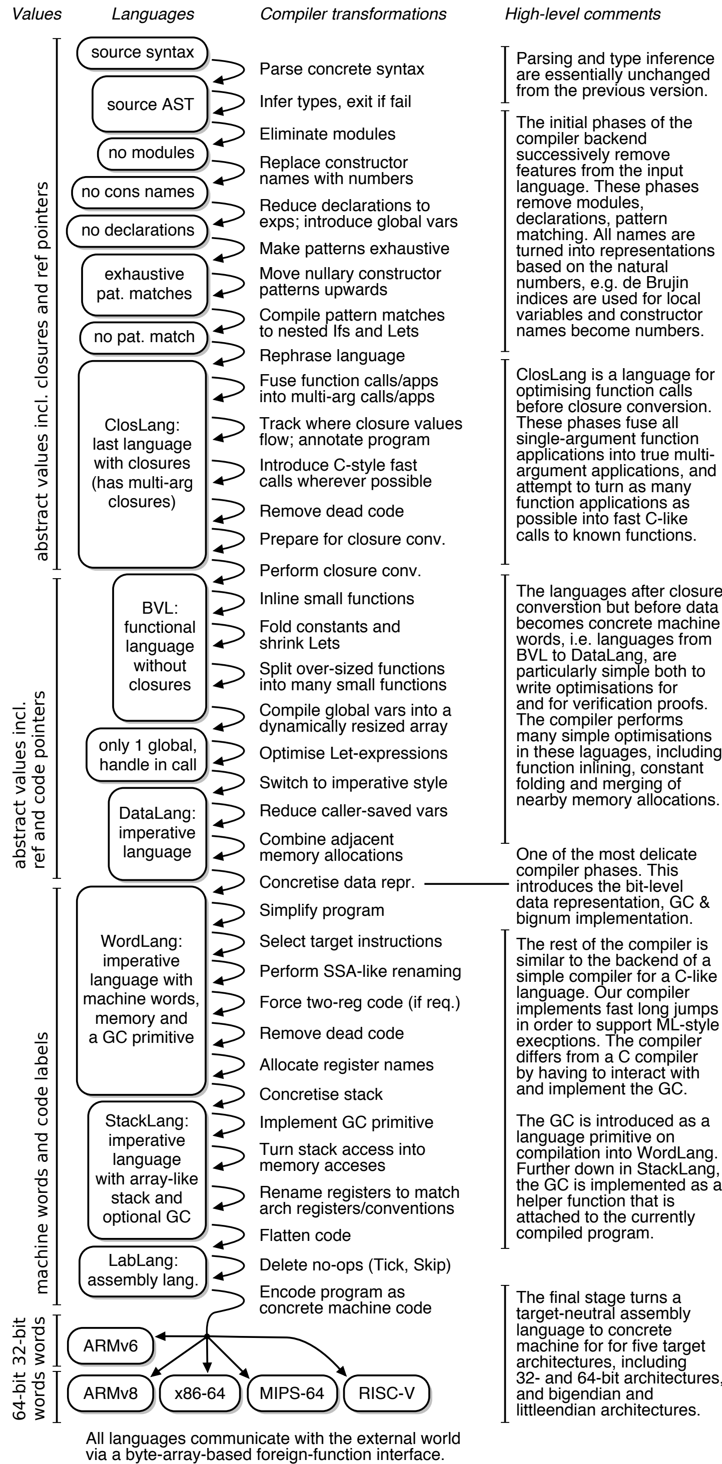


Fig. 1. The structure of the new CakeML compiler.

D0: *Sufficiently many intermediate languages for ease of compiler implementation and verification.* When designing the structure of a compiler there is a question: should there be many intermediate languages or just a few? In the context of a verified compiler, having fewer ILs supports the re-use of infrastructure, including utility lemmas, that is specific to each IL's semantics. However, ILs whose semantics support both higher- and lower-level features can complicate the invariants needed to verify transformations, especially when no program will contain both at the same time. Thus, our CakeML compiler introduces a new intermediate language whenever a significant higher-level language feature has been compiled away, or when a new lower-level one is introduced. Although this leads us to 12 ILs, many transitions work within a single IL, as can be seen from Figure 1. We note that this design choice is not unique to our compiler; CompCert (Leroy, 2009) similarly uses several intermediate languages in order to capture syntactic and semantic guarantees at each stage of compilation.

D1: *Direct-style compilation (as opposed to continuation-passing style), and the call stack separate from the heap.* Compiling with continuation-passing style (CPS) (Appel, 1992) has been investigated by the research community for a while and it is tempting to try to write an ML compiler in that style, as SML of New Jersey has done. It is tempting since all function calls become tail calls which means that there is no need for a conventional call stack in the generated code. The downside is that CPS makes the compiler harder to write (against our P2) and effectively moves the call stack into the heap, which means that stack frame deallocation (which is simple and fast with a direct style) is usually left to the garbage collector in CPS (against our P1). We further note that the leading ML and Haskell compilers (MLton, Poly/ML, OCaml, GHC) opt for direct-style compilation.

D2: *Curried functions, compilation to multi-argument functions, and good closure representations (but without interior pointers).* In ML (and Haskell), each function takes only one argument, and programmers write multi-argument functions at the source level either with currying or taking input as a tuple. Any realistic ML compiler has to optimise for at least one of these patterns, and pass multiple arguments to functions without any intermediate allocation of closures or tuples. The CakeML compiler emphasises curried functions, and compiles them into true multi-argument closure values (emphasis on P1 rather than P2). It uses a flat representation of the closure environments with constant-time lookups. Some compilers use internal pointers, i.e. pointers into the inside of heap objects rather than the heads of heap objects, in the representation of closures for mutually recursive functions. We do not do this partly due to D3 below and partly because it would cause more complexity in the garbage collector and probably slow it down.

D3: *A first-order functional intermediate language,* BVL*, that is as clean and simple as possible.* Motivated by P2, the middle part of the compiler grew outwards from a first-order functional language called BVL, which we have tried to keep as neat and tidy as possible. For purposes of modularity, we did not want to build a specific closure format into the language definition. In the current compiler, the only compiler phase that needs to know the closure format is the compilation pass from

CLOSLANG into BVL, at which point closures are compiled into a format that uses BVL's code pointers. The semantics of BVL treats these code pointers like any other semantic value (constructor, integer, reference pointer, etc.) — a design decision that affects the rest of the compiler, because, from this point onwards, any value can potentially be a code pointer. This has an impact on the garbage collector and lower-level data representations. In particular, the implementation of the garbage collector must be able to efficiently spot the difference between code pointers and data pointers. Efficiency is achieved here by ensuring that all code pointers have the least-significant bit set to zero, while all data pointers have it set to one. We make our verified assembler two-byte align[1] all code labels so that every code pointer has zero in the least significant bit. By representing small data (e.g. small integers) also with a zero in the least significant bit, the garbage collector can treat all code pointers as data. An alternative would have been to do a range check for the code addresses in the garbage collector, but that would have cost us performance and gone against P1.

D4: *Structured control-flow (e.g. if-statements as opposed to gotos) for nearly all of the intermediate languages.* Motivated by P2, a decision was made early on that every intermediate language, except the assembly language LABLANG, should have structured control flow based on if- and case-statements that are familiar from while-languages. An alternative, at least for the lower-level more C-like intermediate languages, is to represent code with a control flow graph consisting of basic blocks along with control flow edges between blocks. For example, the CompCert compiler uses such a representation internally. Our representation loses the flexibility of control flow graph-based representations, but in return makes certain important compilation passes more straightforward to implement and verify, e.g. the liveness analysis described in Section 7.2.

D5: *One place for concretisation of data representations into finite memory.* When structured data (tuples, arrays, arbitrary sized integers, etc.) is compiled into machine words and memory, we do not want to be obstructed by intermediate abstraction levels that hinder low-level trickery with bit-level representations and creative use of memory. By intermediate abstractions, we mean CompCert-style abstractions that make pointers distinct from machine words, and abstractions that give memory more structure than a flat array of machine words. There is a trade-off here: CompCert-like abstractions would have allowed us to avoid the garbage-collection oracle of Section 7.1, but such abstractions would have prevented — or at least complicated — several low-level optimisations and tricks (emphasis on P1 instead of P2 here). One example of such low-level trickery is how the implementation of integer addition performs a word-addition on machine words and then checks with only one test whether there was overflow in the word-addition or whether one of the inputs was a pointer to a bignum.

D6: *Concrete machine code targets for several different architectures.* This is motivated by P1 and means that every part of the compiler must be able to use data repre-

[1] On most architectures, all code labels are naturally aligned to two- or four-bytes because of the instruction format; the x86-64 architecture is an exception.

sentations that work for both 32-bit and 64-bit architectures, and big-endian and little-endian architectures. Concretely, the compiler and its verification proof are parametrised by a configuration record describing the target architecture.

The end-to-end compiler. The complete compiler function takes a string and configuration as input, and produces, on a successful execution, a tuple consisting of a list of bytes, a list of machine words (i.e., either 32 or 64-bit words), and a configuration record. The bytes are machine code for the specific target architecture implementing the input program. The machine words are assumed by the compiler to be loaded into the data section on startup; they contain read-only liveness information that is used by the GC at runtime. The configuration record contains book-keeping information about the compilation. In particular, it contains a list of strings corresponding to the foreign function interface (FFI) entry points the executable assumes it has access to.

Compiler failure. The compiler function is allowed to fail, i.e., return an error. It can return an error due to parsing failure, type inference failure, or instruction encoding failure. The parser and type inferencer have been proved sound and complete, which means that an error there indicates a fault by the user. An instruction encoding error can happen when a jump instruction or similar cannot be encoded at the very last step of compilation, where we perform verified assembly. The related paper (Fox *et al.*, 2017) discusses this problem in detail. In brief, such an encoding error can occur when the compiler attempts to encode, e.g., a jump of l bytes where l is too large to fit within the offset field of the jump instruction's encoding. In practice, these encoding errors are rare: all of the benchmark programs in Section 12 can be successfully compiled for each of our compiler's target architectures. One could potentially prove that instruction encoding errors are impossible for compiler configurations that sacrifice one or two registers as temporary registers in the encoding of unusually long jumps. At the time of writing, we have not proved such a theorem nor constructed such defensive compiler configurations.

2.2 Semantics of Intermediate Languages

The compiler's intermediate languages can be divided into three groups based on the values they operate over. The first group uses abstract values that include closures; the second group uses abstract values without closures; and the third uses machine words and memory. See the annotations on the left in Figure 1.

Every language has a semantics at two levels: there is the detailed expression- or program-level evaluation semantics (called evaluate), and an observational semantics for the whole program (called semantics).

We define our semantics in *functional big-step* style (Owens *et al.*, 2016). This style of semantics means that the evaluate functions are interpreters for the abstract syntax. These interpreter functions use a clock, which acts as fuel for the computation, to ensure that they terminate for all inputs. A special uncatchable *timeout* exception is raised if the clock runs out. An example of an evaluate function is shown in Figure 2 in Section 4.

The semantics functions return the set of terminating/diverging observable behaviours, including FFI calls, that a program can exhibit. Below φ `ffi_state` is the type of the FFI

state which models how the environment responds to FFI calls. We use postfix notation for type constructors and φ is a type variable for the type of the state of the external world.

$$\mathsf{semantics} : \varphi\ \mathtt{ffi_state} \rightarrow \mathtt{program} \rightarrow \mathtt{behaviour\ set}$$

A behaviour is either divergence, termination, or failure. The first two carry a possibly infinite stream of FFI I/O events, representing a trace of all the I/O actions that the program has performed given the initial FFI state.

$$\mathtt{behaviour} = \mathsf{Diverge}\ (\mathtt{io_event\ stream}) \mid \mathsf{Terminate}\ \mathtt{outcome}\ (\mathtt{io_event\ list}) \mid \mathsf{Fail}$$

$$\mathtt{outcome} = \mathsf{Success} \mid \mathsf{Resource_limit_hit} \mid \mathsf{FFI_outcome}\ \mathtt{final_event}$$

If the evaluate function reaches a result for some clock value, then the program has terminating behaviour with the resulting trace of FFI communications. If the evaluate function hits a *timeout* for all clock values, then it has diverging behaviour and returns the least upper bound of the resulting partial FFI traces in the complete partial ordering of lazy lists with prefix ordering. We know this uniquely exists, because we require our evaluate function to be monotone as a function from the clock to partial FFI traces. Finally, if the evaluate function hits an error, then the program is said to Fail.

2.3 Compiler Proofs

The objective of the compiler proofs is to show that the semantics functions of the source and target produce compatible results. The semantics function is overloaded: there is a version for each IL (including source and target). We annotate the functions with A and B for compilation from ILA to ILB. The FFI state, *ffi*, includes an oracle specifying how foreign function calls behave. For most compiler transitions, we can prove that the semantics functions produce identical behaviours, if the original program does not Fail[2]. These theorems have the form:

$$\begin{array}{l}\vdash \mathsf{compile}\ \mathit{config}\ \mathit{prog} = \mathit{new_prog}\ \wedge \\ \quad \mathsf{syntactic_condition}\ \mathit{prog}\ \wedge \\ \quad \mathsf{Fail} \notin \mathsf{semantics}_{\mathrm{A}}\ \mathit{ffi}\ \mathit{prog} \Rightarrow \\ \qquad \mathsf{semantics}_{\mathrm{B}}\ \mathit{ffi}\ \mathit{new_prog} = \mathsf{semantics}_{\mathrm{A}}\ \mathit{ffi}\ \mathit{prog}\end{array}$$

However, for some compiler transformations (e.g., concretisation of data representations and stack concretisation), the output programs are allowed to bail out with an *out-of-memory error*. In such cases, we prove a weaker conclusion of the form:

$$\mathsf{semantics}_{\mathrm{B}}\ \mathit{ffi}\ \mathit{new_prog} \subseteq \mathsf{extend_with_resource_limit}\ (\mathsf{semantics}_{\mathrm{A}}\ \mathit{ffi}\ \mathit{prog})$$

where

$$\begin{array}{l}\mathsf{extend_with_resource_limit}\ \mathit{behaviours} = \\ \ \mathit{behaviours}\ \cup \\ \ \{\ \mathsf{Terminate}\ \mathsf{Resource_limit_hit}\ \mathit{io_list} \mid \exists t\ l.\ \mathsf{Terminate}\ t\ l \in \mathit{behaviours} \wedge \mathit{io_list} \preccurlyeq l\ \}\ \cup \\ \ \{\ \mathsf{Terminate}\ \mathsf{Resource_limit_hit}\ \mathit{io_list} \mid \exists ll.\ \mathsf{Diverge}\ ll \in \mathit{behaviours} \wedge \mathsf{fromList}\ \mathit{io_list} \preccurlyeq_\infty ll\ \}\end{array}$$

[2] For each compiler phase, we assume that the source IL's semantics does not include Fail. At the top level, we have a theorem which states that a type-correct CakeML program never includes Fail in its semantics.

Note that this weaker form of correctness allows for a trivial compiler implementation that always outputs a program that immediately cause an *out-of-memory error*. Users can, with simple experiments, convince themselves that the bootstrapped compiler implementation does not default to such a trivial implementation.

We prove the semantics theorems using simulation theorems relating the respective evaluate functions. At the level of evaluate functions, we prove correctness theorems of the following form, where evaluation in the source IL is assumed and evaluation in the target IL is proved.

$$\begin{array}{l}\vdash \mathsf{compile}\ \mathit{config}\ \mathit{exp} = \mathit{exp}_1\ \wedge \\ \quad \mathsf{evaluate}_{\mathrm{A}}\ \mathit{exp}\ \mathit{state} = (\mathit{new_state},\mathit{res})\ \wedge \\ \quad \mathsf{state_rel}\ \mathit{state}\ \mathit{state}_1 \wedge \mathit{res} \neq \mathsf{Error} \Rightarrow \\ \quad\ \exists \mathit{new_state}_1\ \mathit{res}_1. \\ \quad\quad \mathsf{evaluate}_{\mathrm{B}}\ \mathit{exp}_1\ \mathit{state}_1 = (\mathit{new_state}_1,\mathit{res}_1)\ \wedge \\ \quad\quad \mathsf{state_rel}\ \mathit{new_state}\ \mathit{new_state}_1 \wedge \mathsf{res_rel}\ \mathit{res}\ \mathit{res}_1\end{array}$$

The evaluate functions are also overloaded at each IL. They return a new state, *new_state*, which includes the FFI state, and a result, *res*, indicating a normally returned value or an exception or an error. The state_rel and res_rel relations specify how values are related from one IL to the next. For many of our proofs, *exp* and the compiled code exp_1 will consume the same amount of fuel and so the state_rel relation asserts that the clock values for its two input states are equal. In some proofs, however, extra fuel needs to be added to the clock in state_1 because the compiled code uses (a finite number of) extra ticks. This results in a correctness theorem of the following form:

$$\begin{array}{l}\vdash \mathsf{compile}\ \mathit{config}\ \mathit{exp} = \mathit{exp}_1\ \wedge \\ \quad \mathsf{evaluate}_{\mathrm{A}}\ \mathit{exp}\ \mathit{state} = (\mathit{new_state},\mathit{res})\ \wedge \\ \quad \mathsf{state_rel}\ \mathit{state}\ \mathit{state}_1 \wedge \mathit{res} \neq \mathsf{Error} \Rightarrow \\ \quad\ \exists \mathit{new_state}_1\ \mathit{res}_1\ \mathit{ck}. \\ \quad\quad \mathsf{evaluate}_{\mathrm{B}}\ \mathit{exp}_1 \\ \quad\quad\ (\mathit{state}_1\ \mathsf{with}\ \mathsf{clock} := \mathit{state}.\mathsf{clock} + \mathit{ck}) = \\ \quad\quad\ (\mathit{new_state}_1,\mathit{res}_1) \wedge \mathsf{state_rel}\ \mathit{new_state}\ \mathit{new_state}_1\ \wedge \\ \quad\quad \mathsf{res_rel}\ \mathit{res}\ \mathit{res}_1\end{array}$$

The extra fuel *ck* required in the evaluation of exp_1 is existentially quantified along with $\mathit{new_state}_1$ and res_1.[3] After exp_1 consumes the extra *ck* fuel, the result states *new_state* and $\mathit{new_state}_1$ have the same final clock (as asserted by state_rel). Manipulation of this existentially quantified fuel typically leads to additional complications in our proofs. Thus, our compilation passes are designed to avoid needing extra fuel where possible.

The evaluate theorems are *forward simulation* theorems because the source semantics is on the left hand side of the implication and the target semantics is on the right. Such theorems follow the intuition of the compiler writer. They are proved by induction on the structure of evaluate for the source intermediate language.

Since our ILs are fully deterministic, these forward simulation theorems for evaluate are sufficient[4] for proving the equivalence (or correspondence) of the observational semantics

[3] Fuel is used for evaluation of the compiled code; the compiler itself is proved to always terminate.

[4] See the extensive discussion in Leroy (2009) regarding forward and backward simulations. In his terminology, we generally use forward simulation for safe programs.

at the semantics level, which includes the proof of divergence preservation. Our divergence preservation proofs follow the style of Kumar *et al.* (2014) and Owens *et al.* (2016).

It should be noted that the entire compiler verification could be done at the level of evaluate functions, letting us only at the very end relate the semantics functions for the source and target semantics. We opted for the approach where we relate semantics functions for each major step in the compiler since the equations between semantics functions are easier to compose.

Our overall correctness theorem (stated in Section 10) inherits the weaker conclusion involving extend_with_resource_limit. A trivial compiler that always produces code that runs out of memory would satisfy this kind of theorem. However, it is hard to prove anything stronger when compiling to a machine with finite memory, without exposing unwanted implementation details in the source semantics. Our correctness theorem is to be complemented by observations that our compiler produces code that does not in fact terminate prematurely.

2.4 Top-level Correctness Theorem

The top-level correctness theorem is formally stated in Section 10. Informally, this theorem can be read as follows:

> *Any binary produced by a successful evaluation of the compiler function will either*
>
> - *behave exactly according to the observable behaviour of the source semantics, or*
> - *behave the same as the source up to some point at which it terminates with an out-of-memory error.*
>
> *This theorem assumes that*
>
> - *the compiler and machine configurations are well-formed,*
> - *the initial machine state is set up according to CakeML's assumptions,*
> - *the generated program runs in an environment where the external world only modifies memory outside CakeML's memory region, and*
> - *the behaviour of the FFI in the machine semantics matches the behaviour of the abstract FFI parameter provided to the source semantics.*

The details of the formal statement are made complicated by our support for multiple architectures and the interaction with the FFI.

Structure The rest of the paper gives more details on how the compiler operates, in particular how it removes abstractions as it makes its way towards the concrete machine code of the target architectures. Along the way, we provide commentary on our invariants and proofs. The description of the interaction between the verification of the register allocator and our garbage collector, in Section 7, is given most space.

3 Early Phases

The compiler starts by parsing the concrete syntax and by running type inference — two verified phases that we re-use from the previous CakeML compiler (Kumar *et al.*, 2014; Tan *et al.*, 2015).

The first few transformations of the input program focus solely on reducing the features of the source language. Modules are removed, algebraic datatypes are converted to numerically tagged tuples, top-level declarations are compiled to updates and lookups in a global store, and pattern matches are made exhaustive and then compiled into nested combinations of if- and let-expressions (which get optimised further down). Later on, after BVL, a pass compiles the global store into an array in the heap.

The early stages of the compiler end in a language called CLOSLANG. This language is the last language with explicit closure values, and is designed as a place where functional-programming specific optimisations (e.g., lambda lifting) can prepare the input programs for closure conversion. We give a brief overview of CLOSLANG and its optimisations here, and refer readers to Owens *et al.* (2017) for a complete discussion, including the in-depth details of the techniques used to verify its various optimisations.

CLOSLANG is the first language to add a feature: it adds support for multi-argument functions, i.e., function applications that can apply a function to multiple arguments at once and construct closures that expect multiple simultaneous arguments. All previous languages required either currying or tupled inputs in order to simulate multi-argument functions. A naive implementation of curried functions causes heap-allocation overhead which we reduce by optimising them to (true) multi-argument functions.

A value in CLOSLANG's semantics is either a mathematical integer, a 64-bit word[5], an immutable block of values (constructor or vector), a pointer to a reference or array, or a closure. Through the paper, we use HOL4's type definition syntax: each constructor name is followed by the types of its arguments in Haskell style, but type constructors use postfix application following ML style.

```
v =
   Number int
 | Word64 (64 word)
 | Block num (v list)
 | ByteVector (8 word list)
 | RefPtr num
 | Closure (num option) (v list) (v list) num exp
 | Recclosure (num option) (v list) (v list) ((num × exp) list) num
```

A closure can either be a non-recursive closure (Closure) or a closure for a mutually recursive function (Recclosure). The arguments for the Closure constructor are an optional location for where the code for the body will be placed[6], an evaluation environment (values

[5] We made 64-bit words a primitive value type through the top-half of the compiler in order to aid compiler bootstrapping, as described in Section 11. The lower half of the compiler uses the machine words, which are either 32-bit or 64-bit words, as the value type. Note that the compiler implements CLOSLANG's 64-bit words regardless of the size of the target architecture.

[6] The placement name acts as a name for the origin of the closure. Each closure creating expression can optionally be annotated with a numeric place name. These annotations cause the expression

for the free variables in exp), the values of the already-applied arguments, the number of arguments this closure expects, and finally the body of the closure.

The arguments for Recclosure are similar but with a list of expressions and number of arguments rather than just one number and expression. The final argument to Recclosure is a list index, which indicates which body from the list should be used when this mutually recursive closure is applied. The most significant difference between the Recclosure and Closure values is that the evaluation of a Recclosure body has access to itself, for purposes of recursion. Strictly speaking, each Closure can be represented as a Recclosure value, but we decided to keep the Closure values for presentation and prototyping[7] purposes.

Having (recursive) closure values as part of the language adds a layer of complication to the compiler proofs, since program expressions (exp above) are affected by the compiler's transformations. There are different ways to tackle this complication in proofs.

For pragmatic reasons, most of our proofs use a simple syntactic approach. Our proofs relate the values before a transformation with the values that will be produced by the code after the transformation. Concretely, for a compiler function compile, we define a syntactic relation v_rel which recursively relates each syntactic form to the equivalent form after the transformation. For example,

$$\begin{array}{l}\mathsf{v_rel}\ (\mathsf{Closure}\ loc_1\ env_1\ args_1\ arg_count_1\ exp_1)\\ \quad(\mathsf{Closure}\ loc_2\ env_2\ args_2\ arg_count_2\ exp_2)\end{array}$$

is true if the environment and arguments are related by v_rel and the expressions are related by the compiler function compile, i.e., $exp_2 = \mathsf{compile}\ exp_1$. This style of value relation is very simple to write, but causes some dull repetition in proofs.

An alternative strategy is to use logical relations to relate the values via the semantics: two values are related if they are semantically equivalent. We use an untyped logical relation for CLOSLANG in some proofs (e.g., multi-argument introduction and dead-code elimination), see Owens *et al.* (2017) for more details.

4 Closure Conversion

Closures are implemented in the translation from CLOSLANG into a language called bytecode-value language (BVL). We use this name because BVL uses almost the same value type as the semantics for the bytecode language of the original CakeML compiler. BVL's value type is also almost identical to CLOSLANG's value type; the difference being that BVL does not have closure values, instead it has code pointers that can be used as part of closure representations. CLOSLANG's ByteVector values are transformed into byte arrays when

to tag each created closure value with the name in the annotation. The name annotations are used in an optimisation that attempts to figure out which closures flow where. The compiler optimises function applications that only apply closures with one known name.

[7] Often if a new optimisation can be proved correct for the simpler Closure values, then it can be made to work for the more complicated Recclosure values.

compiling into BVL.

```
v =
  Number int
 | Word64 (64 word)
 | Block num (v list)
 | CodePtr num
 | RefPtr num
```

BVL is an important language for the new CakeML compiler, and is perhaps the simplest language in the compiler. One can view CLOSLANG, which comes before, as an extension of BVL with closures; and one can view the languages after BVL as reformulations of BVL that successively reduce BVL to machine code.

BVL is a first-order functional language with a code store, sometimes called a code table. It uses de Bruijn indices for local variables. The abstract syntax for BVL is given below. Tick decrements the clock in BVL's functional big-step semantics. Call also decrements the clock: its first argument indicates the number of ticks the call consumes; the ticks are consumed after the arguments to the call have been evaluated, but before evaluation enters the body of the called code. Call's second argument is the optional destination of the call, where None means the call is to jump to a CodePtr provided as the last argument in the argument list.

```
exp =
  Var num
 | If exp exp exp
 | Let (exp list) exp
 | Raise exp
 | Handle exp exp
 | Tick exp
 | Call num (num option) (exp list)
 | Op op (exp list)
```

Figure 2 shows an extract of BVL's functional big-step semantics, i.e., functions in HOL that define BVL's big-step semantics. The evaluate function takes a list of BVL expressions exp and returns a list of values v corresponding to the expressions. We could have defined evaluate using mutual recursion and avoided the need for single list applications of evaluate. We chose this style of definition for convenience of proofs: by avoiding mutual recursion we have one statement of the inductive hypothesis instead of two in each proof.

The exception mechanism shapes the look[8] of the BVL semantics. Each evaluation returns either a return-value Rval or raises an exception Rerr. An exception Rerr (Rraise ...) is produced by the Raise program expression. Running out of clock ticks during evaluation results in Rerr (Rabort Rtimeout_error), while hitting a type error in the program results in Rerr (Rabort Rtype_error). Most expressions pass on exceptions that occur inside sub-expressions, with Handle being the only construct that can catch exceptions. Handle can only catch Rraise exceptions, i.e. both Rabort exceptions cannot be caught and will always

[8] The semantics could be made more reader friendly using monad syntax. However, in Figure 2 we opted for a raw style in order to avoid too many auxiliary definitions: monad bind, raising an exception, exception handle, and update and access functions for individual state components.

```
evaluate ([],env,s) = (Rval [],s)
evaluate (x::y::xs,env,s) =
 case evaluate ([x],env,s) of
  (Rval v1,s1) ⇒
   (case evaluate (y::xs,env,s1) of
     (Rval vs,s2) ⇒ (Rval (v1 ++ vs),s2)
    | (Rerr e,s2) ⇒ (Rerr e,s2))
 | (Rerr v10,s1) ⇒ (Rerr v10,s1)
evaluate ([Var n],env,s) =
 if n < len env then (Rval [nth n env],s)
 else (Rerr (Rabort Rtype_error),s)
evaluate ([Let xs x],env,s) =
 case evaluate (xs,env,s) of
  (Rval vs,s1) ⇒ evaluate ([x],vs ++ env,s1)
 | (Rerr e,s1) ⇒ (Rerr e,s1)
evaluate ([Op op xs],env,s) =
 case evaluate (xs,env,s) of
  (Rval vs,s1) ⇒
   (case do_app op (rev vs) s1 of
     Rval (v,s2) ⇒ (Rval [v],s2)
    | Rerr err ⇒ (Rerr err,s1))
 | (Rerr v9,s1) ⇒ (Rerr v9,s1)
evaluate ([Raise x],env,s) =
 case evaluate ([x],env,s) of
  (Rval vs,s1) ⇒ (Rerr (Rraise (hd vs)),s1)
 | (Rerr e,s1) ⇒ (Rerr e,s1)
evaluate ([Handle x1 x2],env,s) =
 case evaluate ([x1],env,s) of
  (Rval v,s1) ⇒ (Rval v,s1)
 | (Rerr (Rraise v),s1) ⇒ evaluate ([x2],v::env,s1)
 | (Rerr (Rabort e),s1) ⇒ (Rerr (Rabort e),s1)
evaluate ([Call ticks dest xs],env,s) =
 case evaluate (xs,env,s) of
  (Rval vs,s1) ⇒
   (case find_code dest vs s1.code of
     None ⇒ (Rerr (Rabort Rtype_error),s1)
    | Some (args,exp') ⇒
      if s1.clock < ticks + 1 then
        (Rerr (Rabort Rtimeout_error),s1 with clock := 0)
      else
        evaluate ([exp'],args,dec_clock (ticks + 1) s1))
 | (Rerr v8,s1) ⇒ (Rerr v8,s1)
  ...

do_app (Const i) [] s = Rval (Number i,s)
do_app (Cons tag) xs s = Rval (Block tag xs,s)
  ...
```

Fig. 2. Extracts of BVL's semantics.

bubble up to the top-level. We prove that well-typed CakeML programs cannot produce Rtype_error.

The semantics of Call is the most interesting part of BVL. Call starts by evaluating the argument expressions. It then finds the code for the called function from the code field of the state. If the name of the called function is given explicitly in *dest* then the values *vs* are used as arguments, otherwise the last value in *vs* must be a CodePtr and all but the last element of *vs* is returned as *args*. The value of the clock is checked before evaluation continues into the code of the called function; a too small clock value causes a Rtimeout_error. The values of the passed arguments *args* are the initial environment for the evaluation of the called function.

4.1 Closure Representation

We use BVL's Blocks and value arrays to represent closures in BVL. Non-recursive and singly recursive closures are represented as Blocks with a code pointer and the argument count followed by the values of the free variables of the body of the closure.

Block closure_tag
([CodePtr *ptr*; Number *arg_count*] ++ *free_var_vals*)

Mutually recursive closures are represented as Blocks, where the free-variable part is a reference pointer to a value array.

Block closure_tag
[CodePtr *ptr*; Number *arg_count*; RefPtr *ref_ptr*]

Such arrays contain the closures for each of the functions in the mutual recursion and the values of all their free variables. Arrays are used for the representation of mutually recursive closures since such closures need to contain their own closure values. Arrays are the only way to construct the required cyclic structures in BVL. The arrays used for closures are only mutated as part of the closure-creation process.

The compilation of closure construction relies on a preliminary pass within CLOSLANG that annotates each closure creation with the free variables of the closure bodies. The same transformation shifts the de Bruijn indices to match the updated evaluation environment.

4.2 Multi-argument functions

The compilation into BVL needs to implement CLOSLANG's function application expression. The semantics of CLOSLANG's function application expression is far from simple, since CLOSLANG allows multi-argument closures and multi-argument function applications. In particular, the semantics deals with the case where the argument numbers do not match. Owens *et al.* (2017) explain our compilation strategy for and verification of multi-argument function compilation; here we give an overview.

Each *n*-argument function application is compiled to code which first evaluates the arguments and then the closure; it then checks if the closure happens to expect exactly the given number of arguments; if it does, then the code calls the code pointer in the closure (or makes a direct jump if the CLOSLANG function application expression is annotated with a known jump target, which is the case for known functions). In all other cases, i.e., if

there is any mismatch between the number of arguments, the code makes a call to a library function (also written in BVL), which implements CLOSLANG's mismatch semantics. The semantics dictates that partial applications result in new closure values with additional already-provided arguments. Applications that are given too many arguments — a valid case — are split into a call to the expected number of arguments, followed by a call for the remaining arguments. Jump-table-like structures are used to quickly find the right case amongst all the combinations of possible cases. The BVL code for these library functions is generated from verified generator functions; given a maximum number of arguments that can be passed in one go, these generator functions will generate all of the required library functions.

Our support for this kind of multi-argument semantics is similar to OCaml's, and relies on the adoption of right-to-left evaluation order for application expressions. We expect most well-written CakeML programs to use mutable state sparingly, and that applied arguments will usually be pure. Therefore, the evaluation order should not matter. This change was necessary to keep the BVL code that implements multi-argument function applications short and fast.

5 Going Fully Stateful

BVL programs are compiled via an IL to an imperative version of BVL called DATALANG. DATALANG is the last language with functional-style abstract data. In DATALANG, local variables are held in the state as opposed to in an environment, and there is a call stack. On function calls, a subset of the local variables are stored onto the stack, and on return they are restored from the stack. Subsequent languages, WORDLANG and STACKLANG, mimic DATALANG in style and structure. The abstract syntax of DATALANG programs is as follows:

```
prog =
  Skip
 | Move num num
 | Call ((num × num_set) option) (num option)
     (num list) ((num × prog) option)
 | Assign num op (num list) (num_set option)
 | Seq prog prog
 | If num prog prog
 | MakeSpace num num_set
 | Raise num
 | Return num
 | Tick
```

In the abstract syntax, all numbers (of type num) are variable names with the exception of the second argument to Call which is an optional target location for the call. As in BVL, None indicates that a code pointer from the argument list is to be used as the target. The first argument to Call is a return variable name, where None indicates that this is a tail call. The last argument to Call is an optional exception handler. The exception handlers are fused into Calls so that raising an exception can jump directly to the handler's stack frame and restore the values of the local variables from that stack frame. The finite sets of numbers (of type num_set) are cut-sets that keep track of the local variables that survive past calls. These cut-sets specify which variables must be included in the stack frame at this point

and specifies that these variable names need to be marked as live for the garbage collector, which is introduced later. Besides Call, the allocation primitive MakeSpace, which is descrbed in more detail below, also takes a cut-set argument since its implementation may make a call to the GC. Similarly, Assign needs an (optional) cut-set argument because the implementations of some operations, e.g. bignum arithmetic, may internally require subroutine calls in their implementation in WORDLANG.

DATALANG's semantics uses the same value type as BVL and operates over a state that is defined as a record type as shown in Figure 3. The state has a field for the local variables which is a finite mapping[9] from `num` to `v`. The stack is a list of frames where each frame is either a normal value environment (Env) or an environment with an exception handler value (Exc). When an exception is raised, the stack is reset to have the length of the state's handler field, the locals are restored, and the state's handler field is set to be the number stored in the Exc frame. There is a special optional global reference which in practice points at an array containing all the global variables of the program. Furthermore, there are fields for the references, the functional big-step semantics' clock, the code, and the state of the foreign function interface. Finally, there is space field which is described below.

φ `state` = ⟨
  locals : v `num_map`;
  stack : `frame list`;
  global : `num option`;
  handler : `num`;
  refs : `num` $\mapsto$ v `ref`;
  clock : `num`;
  code : (`num` $\times$ `prog`) `num_map`;
  ffi : φ `ffi_state`;
  space : `num`
⟩

`frame` = Env (v `num_map`) | Exc (v `num_map`) `num`

α `ref` = ValueArray (α `list`) | ByteArray `bool` (8 `word list`)

Fig. 3. The definition of the DATALANG state.

DATALANG is designed to support optimisation of memory allocation: a compiler pass within DATALANG combines memory allocations (MakeSpace) in straight-line code so that simple source-level expressions, such as `([x,4,5],true,[a])`, call allocation only once. The semantics of an allocation, MakeSpace *n names*, is to guarantee that there are at least *n* units of space available: the semantics simply assigns *n* to the space field. Correspondingly, operations such as Cons consume space equal to the length of the Block that is produced. The MakeSpace optimisation in DATALANG collects and combines instances of MakeSpace in straight-line code. Some operations, such as e.g. bignum addition, consume a statically

[9] This paper uses three forms of mappings where `num` is type of the keys: `num` $\rightarrow$ α (a normal total function in HOL), `num` $\mapsto$ α (a partial function with a finite domain) and α `num_map` (a different partial function with a finite domain). For the purposes of this paper, one can read α `num_map` and `num` $\mapsto$ α as the same.

unknown amount of space, which resets the available space to zero and obstructs the MakeSpace optimisation.

In DATALANG, the space measure is an abstract notion since there is no memory. However, memory becomes very concrete when DATALANG is compiled into WORDLANG. This compilation step is described next.

6 Concretisation of data representations

DATALANG sets the stage for the concretisation of data representations. DATALANG is compiled into a language called WORDLANG, which has an abstract syntax that at a glance looks similar to DATALANG's. The major difference is in the values of the semantics: DATALANG values are of type `v`, whereas WORDLANG values are of type α `word_loc`, which are machine words and labels. The type variable α indicates the length of the machine word.

$$\alpha\,\texttt{word_loc} \;=\; \mathsf{Word}\;(\alpha\,\texttt{word}) \mid \mathsf{Loc}\;\texttt{num}\;\texttt{num}$$

We keep the type parametric throughout most of the compiler in order to support architectures with different word sizes, and maximise code re-use. Some of our proofs assume that α is either 32 or 64. Labels, i.e. Loc n_1 n_2 values, have two parts: n_1 is the name of the function, and n_2 is the label name within function n_1.

Compared with DATALANG, WORDLANG operates over a more complex state. The WORDLANG state, a record as shown below, includes (in order) a local variable store, floating-point registers (for forthcoming support for floating-point arithmetic), a global variable store, a stack, a word-addressed memory and the memory domain.

$$
\begin{array}{l}
(\alpha,\,\varphi)\;\texttt{state} = \langle \\
\quad \mathsf{locals} \;:\; \alpha\,\texttt{word_loc}\;\texttt{num_map}; \\
\quad \mathsf{fp_regs} \;:\; \texttt{num} \mapsto 64\,\texttt{word}; \\
\quad \mathsf{store} \;:\; \texttt{store_name} \mapsto \alpha\,\texttt{word_loc}; \\
\quad \mathsf{stack} \;:\; \alpha\,\texttt{stack_frame}\;\texttt{list}; \\
\quad \mathsf{memory} \;:\; \alpha\,\texttt{word} \to \alpha\,\texttt{word_loc}; \\
\quad \mathsf{mdomain} \;:\; \alpha\,\texttt{word}\;\texttt{set}; \\
\quad \mathsf{permute} \;:\; \texttt{num} \to \texttt{num} \to \texttt{num}; \\
\quad \mathsf{gc_fun} \;:\; \alpha\,\texttt{gc_fun_type}; \\
\quad \mathsf{handler} \;:\; \texttt{num}; \\
\quad \mathsf{clock} \;:\; \texttt{num}; \\
\quad \mathsf{termdep} \;:\; \texttt{num}; \\
\quad \mathsf{code} \;:\; (\texttt{num} \times \alpha\,\texttt{prog})\;\texttt{num_map}; \\
\quad \mathsf{be} \;:\; \texttt{bool}; \\
\quad \mathsf{ffi} \;:\; \varphi\,\texttt{ffi_state} \\
\rangle
\end{array}
$$

We defer discussion of the GC primitive (gc_fun) and the permute oracle (permute) to Section 6.1 and Section 7.1 respectively. The remaining fields are, in order: an exception handler pointer, a clock (used in functional big-step semantics), a depth counter used for book-keeping (its details are unimportant for this paper), a code table (looked up by function calls), a flag controlling big-endianness, and a state for the FFI. Note that

some state components, namely mdomain, code, be and gc_fun, stay constant throughout execution but are carried around by the semantics as part of the state for convenience[10].

The stack is a list of stack frames of the type shown below. The first argument is an association list mapping variable names to values. The second argument, i.e., the optional triple of numbers, is a Some only if the corresponding frame in DATALANG is an Exc. The first number in the triple is the handler value from DATALANG's Exc and the two other numbers are the components of a Loc pointing to the code for the handler.

```
α stack_frame =
  StackFrame ((num × α word_loc) list) ((num × num × num) option)
```

WORDLANG is set up to allow easy manipulation of local variables, including renaming of variables, introduction of new variables, and parallel copying/movement of variables. These kinds of manipulations are required for instruction selection and register allocation.

6.1 Garbage Collection Primitive

After concretising the data representation, WORDLANG programs operate over a *finite*, word-addressed memory. Thus, it becomes necessary for us to model how stale memory can be reclaimed and re-used along a WORDLANG program's execution. In other words, this is the first IL where we need to introduce a notion of garbage collection (and where we implement arbitrary precision arithmetic).

The garbage collector is present in the WORDLANG state as a black-box primitive (gc_-fun) since it cannot be implemented as a WORDLANG program. There are two reasons for this: functions in WORDLANG cannot inspect or update their callers' stack frames, and compilation of WORDLANG programs passes through register allocation which can lead to spilling of local variables onto the stack. Having part of the GC's state spilled onto the stack would cause complications, since the GC walks the stack as part of its execution. Our solution is to avoid these complications by making the GC a special semantic subroutine that one can call through execution of a WORDLANG command called Alloc. The GC primitive is abstractly characterised in the correctness theorem from DATA-to-WORD, and it is removed, i.e., implemented as a call to verified library code, in STACKLANG.

The semantics of executing WORDLANG's GC primitive is shown in Figure 4. The GC function is passed the roots extracted from the stack. It updates the stack with the new root values when it completes.

We ensure that a stack swapping property holds of the WORDLANG semantics: for any WORDLANG program, its local execution behaviour is unchanged when we swap the stack component – as long as the new stack's roots after enc_stack look the same. Intuitively, this holds because the only semantic primitive that directly inspects the stack is the GC, and its behaviour is unchanged when it sees the same roots.

Why is the GC primitive a component of the WORDLANG state? The semantics needs to know what garbage collector to use as the CakeML compiler supports both a generational

[10] In retrospect, these should probably have been carried around in a separate configuration record instead of the state record.

```
gc s =
  case s.gc_fun (enc_stack s.stack,s.memory,s.mdomain,s.store) of
    None ⇒ None
  | Some (roots′,m′,st′) ⇒
      case dec_stack roots′ s.stack of
        None ⇒ None
      | Some stack′ ⇒
          Some (s with ⟨stack := stack′; store := st′; memory := m′⟩)

enc_stack [] = []
enc_stack (StackFrame l _::st) = map snd l ++ enc_stack st
```

Fig. 4. The semantics of invoking WordLang's GC.

and non-generational versions; and each collector is parametrised by the compiler configuration which determines how data is represented. One could bring in all the necessary compiler configurations into the semantics of the WordLang intermediate language. However, it seemed cleaner to just carry around one function that is the garbage collector and let the proofs constrain the choice of this function based on the compiler configurations.

6.2 Value Representation and Heap Invariant

The compiler from DataLang to WordLang is set up so that the only interesting thing it does is change the value representation. Even so, it is non-trivial to verify. The proofs about the value representation are made complicated by the big leap in abstraction level, the fact that WordLang can run a garbage collector, must implement arbitrary precision arithmetic (bignums), and the flexibility we want in the data representation.

The details of the value representation's definition are dictated by the presence of the garbage-collector verification. We adapted the original CakeML compiler's verified copying collector (Myreen, 2010), which is defined at an abstract level in order to maximise potential for reuse. The invariant relating DataLang's values with WordLang's machine words and memory is phrased as an instantiation of the garbage collector's abstract datatypes.

Unfortunately, the layered structure of this definition, which makes the proofs manageable, also makes the definitions too long to reasonably fit into this paper. The lengthy definition is the topic of a forthcoming long version of our paper on CakeML's latest garbage collector (Ericsson *et al.*, 2017). However, an important abstraction in the invariant used for the Data-to-Word compiler proof is memory_rel, which relates the reference store and space of a DataLang state *s*, with the global store, memory and big-endianness (be) of a WordLang state *t*, as well as a list, *value_pairs*, of pairs (v,w) with the first component a value from DataLang and the second a value that fits into a local variable in WordLang (i.e., a α `word_loc`).

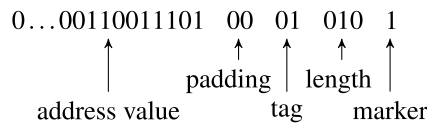
```
memory_rel config s.refs s.space t.store t.memory
      t.mdomain t.be value_pairs
```

Here *s*.space is a lower bound: memory_rel is true for space *n* if the heap has space for at least *n* α `word_loc` values.

We opt for an informal description of the specifics of the representation of DATALANG's values in WORDLANG. Our convention is to use word values with a least significant bit of zero for values that the GC is not to treat as pointers (i.e., small numbers, empty Blocks and code pointers). We chose zero because it allows addition and subtraction of small numbers to be performed directly on the word values. Similarly, we arrange the assembler to two-byte align all labels so that Loc values are all represented (further down) with zero as the least significant bit. This way the garbage collector treats code pointers as small integer values and the collector naturally does not confuse[11] code pointers with data pointers.

Pointers can carry information. Each pointer has a least significant bit of 1, followed by a length field, a tag field, some zero padding and finally the actual address of the pointer.

Example pointer value:



The lengths of the padding-, length-, and tag-fields are configurable and can be set to zero, i.e., removing them from the representation. The padding helps remove extra shift instructions. Each pointer dereference uses shifts to remove the extra information around the pointer value. One (logical) right shift deletes the extra information. An additional left shift is required to word align the address value in case there are not enough zero padding bits (3 for 64-bit and 2 for 32-bit) for the first shift to leave behind.

The length and tag fields are used for storing information about the object pointed to. These fields are used in the implementation of DATALANG primitives used by pattern matching: if the tag and length values to be checked are small enough to fit in these fields, then no pointer dereference is needed. Values that exceed the capacity of the small length and tag fields of pointers are represented as a bit pattern of all ones. For example, if the tag field is 2 bits long, then tag value 1 is represented as binary 01, tag value 2 is represented as binary 10, and all tag values greater than 2 are represented in binary as 11. A tag field of all ones indicates that the tag value did not fit in the field.

Currently, elements on the heap are represented by a header word followed by the payload of the heap element. The header contains information indicating what kind of heap element the payload is. For example, the header of a Block element:

- tells the GC whether the payload is garbage collectable values, and
- contains the tag and length of the Block.

At the time of writing, we are considering dropping the header from the memory representation in cases where the tag and length fields of pointers carry all the necessary information. Such an optimisation would save space for many common constructors, e.g.,

[11] In our set up, the GC must be able to tell the code pointers and data pointers apart because it can encounter them in the same situations. This can be seen from the BVL's value type v where the CodePtr and the Block are both present and can be mixed. At the level of WORDLANG, CodePtr values become Loc values and each non-empty Block is accessed using a data pointer.

list-cons is likely to be represented as two words in memory as opposed to the current three. However, the more space efficient representation might cause certain operations to compile to slightly slower code.

6.3 Implementing bignum arithmetic

Arbitrary precision arithmetic is implemented in the DATA-to-WORD compilation pass. On the DATALANG side, we have mathematical integer values, while in WORDLANG, all values must fit into machine words or memory. For efficiency, we distinguish between *small* integer values that fit inside a machine word, and *large* integer values that do not. In our implementation, small integers are kept directly as machine words in local program variables, while large integers (bignums) are represented as pointers to a memory location which stores the “digits” of the large integer (in this context, each “digit” is a machine word).

The fact that a value can either be a pointer or a direct value forces the compiler to generate code which works for any mix of these representations. Here, it is crucial that we can make use of bit-level trickery in the implementations in order to produce code with acceptable performance. Arithmetic on small integer values is compiled directly to word arithmetic, except if an overflow occurs, in which case additional code produces the corresponding in-memory bignum representation. For the cases where large integers are involved, the generated code jumps to library code that implements general bignum arithmetic. This library code, which has been proved correct, is attached as a WORDLANG program to the top of the program being compiled. Subsequent compilation passes in WORDLANG and beyond treat the library code identically to any other part of the program. Our verified bignum library has its roots in our previous work on verified bignum arithmetic (Myreen & Curello, 2013).

7 Low-level Optimisation Steps

After removing the data abstraction, we land in WORDLANG. Many of our low-level optimisations and target-specific compilation steps are performed here. In this section, we discuss three important WORDLANG compilation steps: instruction selection, an SSA-like variable transformation, and register allocation. Other optimisations performed in WORDLANG can be seen in Figure 1.

In our context—a functional language with a copying garbage collector—verifying register allocation is more complicated than usual, mostly due to our design decision D5 as described in Section 2.1. The GC affects the situation via a combination of circumstances:

- The GC looks for roots in the stack as part of its operation.
- The order in which these roots are processed affects the output of our copying GC. A new order can result in a different output.
- The exact order of the roots on the stack is determined by the register allocator when it gives names to spilled variables.
- The verification proof for the register allocator does not have direct access to invariants from the DATA-to-WORD proof, which imply that any order will do.

On the other hand, we also only want to perform register allocation after making several low-level code optimisation steps and certainly after SSA form and its related optimisations. We start by explaining a semantic device, which we call a *permute oracle*, to communicate that any order picked by the register allocator will do for the overall proof.

7.1 Permute Oracle

The WordLang semantics has a component called the *permute oracle* which allows us to influence the order in which the GC primitive sees its roots. Briefly speaking, we use this oracle to control variable orderings on the stack in WordLang so that we can decouple reasoning about an abstract GC function from its concrete implementation in StackLang (the language after WordLang). Formally, a permute oracle is an infinite sequence of bijections on the natural numbers (i.e., WordLang variable names). The HOL type of the permute oracle is `num` → `num` → `num`, as can be seen under permute in Section 6.

Stack frames in WordLang are created when a caller function needs to give up control to its callee: it saves the local variables it needs onto the stack and pops them off when control is returned. Calls to the GC are treated similarly – this means that the GC only needs to look at the current stack for the root set when it is called rather than all the live registers.

It is important to note that these these stack frames are still relatively abstract: they *do not* yet contain the spilled variables. In WordLang (and DataLang), they are simply used to store the local variables that need to be saved across a call. The more conventional notion of stack frames is introduced later when we enter StackLang.

The semantics uses the following functions to push a stack frame onto the stack.

```
env_to_list env permute =
 let mover = permute 0;
   permute = (λ n. permute (n + 1));
   l = toAList env;
   l = sort key_val_compare l;
   l = list_rearrange mover l
 in
  (l,permute)

push_env env handler s =
 let (l,p) = env_to_list env s.permute
 in
  case handler of
    None ⇒ s with ⟨permute := p; stack := StackFrame l None::s.stack⟩
  | Some (_,_,h) ⇒
     s with
     ⟨permute := p; stack := StackFrame l (Some (s.handler,h))::s.stack;
      handler := len s.stack⟩
```

To create a stack frame, the locals are first reduced down to the set of variables that need to be saved, then they are sorted by variable name to get a list of pairs of variable names and their values. The head of the permute oracle is popped and used to permute this sorted list by index (using the list_rearrange function), and the resulting list is added to the

WORDLANG stack as a new stack frame. (The with keyword above denotes record update.) If there is an exception handler, this is pushed via the second argument of StackFrame.

The presence of this oracle component in WORDLANG is best motivated by considering the adjacent correctness theorems. For brevity, we only show the general shape of these theorems. We also annotate each of the evaluation and compilation functions with the first letter of the associated languages e.g., D for DATALANG.

For compilation from DATALANG into WORDLANG (DATA-to-WORD), we want to show that it is correct regardless of the order in which the GC visits the roots. This is controlled by the order in which values appear on the stack, and therefore, by how we permute the values when creating stack frames. Hence, in the theorem below, we prove that DATA-to-WORD is correct *for all* choices of permute oracles *perm*.

$$\begin{array}{l}
\vdash \mathsf{evaluate}_{\mathrm{D}}\ (prog,s) = (res,s_1) \wedge \\
\quad res \neq \mathsf{Some}\ (\mathsf{Rerr}\ (\mathsf{Rabort}\ \mathsf{Rtype_error})) \wedge \\
\quad \mathsf{state_rel}\ c\ l_1\ l_2\ s\ t\ []\ locs \wedge t.\mathsf{termdep} > 1 \Rightarrow \\
\quad\ \mathsf{let}\ (res_1,t_1)\ = \\
\quad\quad\quad \mathsf{evaluate}_{\mathrm{W}} \\
\quad\quad\quad\ (\mathsf{compile}_{\mathrm{DTW}}\ c\ n\ l\ prog,t\ \mathsf{with}\ \mathsf{permute}\ :=\ perm) \\
\quad\ \mathsf{in} \\
\quad\quad \ldots
\end{array}$$

On the other hand, when we compile from WORDLANG into STACKLANG (WORD-to-STACK), we need to concretely implement the stack. One critical step of this concretisation is to give a fixed ordering to variables on the stack; this allows us to generate fixed lookups into the stack and fixed code for the GC implementation later. Hence, we have to pick *some* fixed permute oracle and prove that WORD-to-STACK is correct with respect to it. In the following theorem, we choose the oracle to be the infinite sequence of identity functions, i.e. K I, where K and I are the usual combinators.

$$\begin{array}{l}
\vdash \mathsf{evaluate}_{\mathrm{W}}\ (prog,s) = (res,s_1) \wedge res \neq \mathsf{Some}\ \mathsf{Error} \wedge \\
\quad \mathsf{state_rel}\ k\ f\ f'\ s\ t\ lens \wedge s.\mathsf{permute} = \mathsf{K}\ \mathsf{I} \wedge \ldots \Rightarrow \\
\quad\ \exists ck. \\
\quad\quad \mathsf{let}\ (res_1,t_1)\ = \\
\quad\quad\quad\quad \mathsf{evaluate}_{\mathrm{S}} \\
\quad\quad\quad\quad\ (\mathsf{compile}_{\mathrm{WTS}}\ prog\ bs\ (k,f,f'), \\
\quad\quad\quad\quad\quad t\ \mathsf{with}\ \mathsf{clock}\ :=\ t.\mathsf{clock} + ck) \\
\quad\quad \mathsf{in} \\
\quad\quad\quad \ldots
\end{array}$$

Finally, the oracle allows us to reason about WORDLANG to WORDLANG (WORD-to-WORD) code transformations where variables are renamed. Without the oracle, renamed variables may not be sorted in the same order when creating stack frames. In that case, the GC will not see the roots in the same order, and its behaviour will be altered. By choosing the oracle so that the values of stack frames always line up, we can avoid explicit reasoning

about the GC in our proofs for these kinds of transformations.

$$
\begin{array}{l}
\vdash \ldots \Rightarrow \\
\quad \exists perm'. \\
\quad\quad \mathsf{let}\ (res,rst) = \\
\quad\quad\quad\quad \mathsf{evaluate}_{\mathrm{W}}\ (prog,st\ \mathsf{with\ permute} := perm'); \\
\quad\quad\quad (res',rcst) = \\
\quad\quad\quad\quad \mathsf{evaluate}_{\mathrm{W}} \\
\quad\quad\quad\quad\quad (\mathsf{compile}_{\mathrm{WtW}}\ t\ k\ a\ c\ ((name_num,n,prog),col), \\
\quad\quad\quad\quad\quad\ \ st\ \mathsf{with\ permute} := perm) \\
\quad\quad \mathsf{in} \\
\quad\quad\quad \ldots
\end{array}
$$

Given this intuition, and considering the adjacent correctness theorems, we arrive at a slightly surprising form for correctness theorems for WORDLANG transformations that change variable names (shown above). We shall prove that for *any* oracle *perm* used to evaluate the program after the transformation, there exists *some* oracle $perm'$ such that the program semantics were preserved with respect to the untransformed program. This is useful in two ways: (1) multiple transformations that have correctness theorems of this form can be composed to give a correctness theorem with the same form, and (2) it connects with the correctness theorems for the adjacent languages.

The reasoning for the second point is as follows: for the oracle *perm* we picked in WORD-to-STACK compilation, the WORD-to-WORD correctness theorem gives us some oracle $perm'$ such that the WORD-to-WORD transformations preserve program semantics with respect *perm*. Since the DATA-to-WORD compilation works for any oracle, this $perm'$ can be used to instantiate its correctness theorem when we compose all of three correctness theorems. Note that the permute oracle is a local mechanism to connect the correctness theorems of these passes; after composing the theorems, the permute oracle does not appear in our top-level correctness theorem.

At first glance, the permute oracle seems rather cumbersome to work with. Indeed, it would be a lot of verification effort to prove the aforementioned theorem with respect to the oracle for every WORDLANG compilation step. Fortunately, most of the optimisations that we are interested in can be proved without the generality of the existentially quantified oracle provides. In those cases, picking $perm' = perm$ in the oracle-style theorem presented above effectively reduces it to a forward-style compilation theorem. However, we do need the oracle for some proofs, in particular, for our register allocator whose permute oracle proof we describe next.

7.2 Register Allocation

The *register allocator* compiles from an infinite set of temporary variables down to the finite set of registers available in the target machine. At a high-level, this proceeds in two steps: we first perform liveness analysis to find variables that cannot be assigned to the same registers, then we allocate variables to registers following those constraints. The latter step is done heuristically using a graph colouring algorithm, with the aim of minimising the number of spilled variables and maximising the number of coalesced moves.

Since the semantics of WORDLANG does not distinguish registers from temporaries, the allocator implicitly adopts special naming conventions for variables and we separately

prove that it generates syntactically correct outputs for the next phase of compilation. For example, even variable names of the form $2n$ where n is less than the number of registers refer to the nth target register. This syntactic separation also lets us easily force the allocator to set up syntactic calling conventions. We use it to ensure that all caller save variables are appropriately assigned to stack positions when making function calls, and also that callee arguments are passed inside the appropriate registers (some may also be passed on the stack if there are too many of them). To prevent these conventions from degrading the performance of the allocator, we also introduce extra temporaries and moves between them and the appropriate registers / stack positions so that the register allocator can potentially perform some coalescing.

There are two important simplifications in our register allocator:

1. The control flow graph of its input programs always forms a directed acyclic graph (DAG). Control flow graphs form DAGs in WORDLANG because all loops in CakeML are written using recursion and control-flow within functions can only flow forward[12].
2. We assume that two registers are reserved for compiling stack accesses later in the compiler.

The first simplification allows us to use a bottom-up liveness analysis rather than more complex fixed-point iterations, e.g., Kildall's algorithm as used by CompCert (Leroy, 2009). A similar simplification was also used by Chlipala (2010). The second simplification means that the compiler is not required to rewrite code to handle spills, e.g., by inserting memory loads/stores, in the middle of register allocation. This allows us to decouple register allocation from stack allocation later in the compiler, and greatly simplifies the verification of both phases. However, it means that our allocation is sometimes less efficient because some unnecessary spills might occur due to having fewer registers available.

Since we use a graph-colouring based allocator, we refer to the mapping from temporaries to registers as a *colouring function* and the output after register allocation as a *coloured program*. The key insight is that we can first abstractly characterise colouring functions that satisfy the constraints generated by the liveness analysis, and then separately prove that the graph colouring algorithm generates a colouring function satisfying this property. The main advantage here is that we only need to reason about the permute oracle in the proof of the former.

7.2.1 Correctness of Liveness Analysis

Liveness analysis generates a set of live/clash sets at each program point. These sets are exactly the variables that cannot be assigned to the same register (or stack location) since their values might be simultaneously live. The abstract characterisation of suitable colouring functions, f, thus checks that it is injective over all the live sets generated in this way. Given such a function, we can apply it to a program by recursively colouring the

[12] All tail-recursive calls are optimised to direct jumps; either to a fixed-offset when the target is known, or to a register value otherwise. However, the compiler currently does not go as far as inlining tail-recursive functions as while loops or similar.

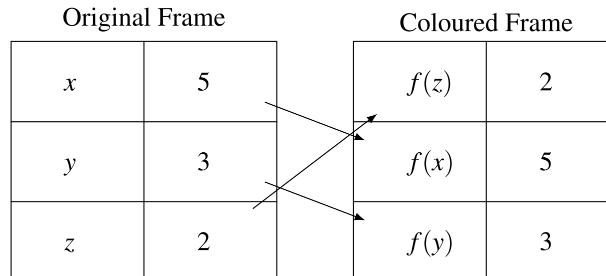
program's temporaries with it. The full correctness theorem including the permute oracle is shown below.

$\vdash$ colouring_ok f *prog live* $\wedge$ state_rel *st cst* $\wedge$
  loc_rel f (dom (get_live *prog live*)) *st*.locals *cst*.locals $\Rightarrow$
   $\exists$*perm*$'$.
    let (*res*,*rst*) = evaluate$_W$ (*prog*,*st* with permute := *perm*$'$)
    in
     if *res* = Some Error then true
     else
      let (*res*$'$,*rcst*) = evaluate$_W$ (apply_colour f *prog*,*cst*)
      in
       res = *res*$'$ $\wedge$ state_rel *rst rcst* $\wedge$
       case *res* of
        None $\Rightarrow$ loc_rel f (dom *live*) *rst*.locals *rcst*.locals
       | Some v $\Rightarrow$ *rst*.locals = *rcst*.locals

One can immediately see several features of a standard forward-style compiler correctness proof. The predicate colouring_ok abstractly characterises f being injective over the livesets of *prog*. The predicate loc_rel checks that the local variables of the two states are related under the colouring function, but only on the variables that are currently live. Finally, state_rel checks that *st*, *cst* are equal on all other state components that are not affected by the colouring. We prove, inductively, that these two state invariants are maintained across the program in the conclusion of the theorem, except when the original source program had a type error (*res* = Some Error). Importantly, *res* = *res*$'$ in the conclusion shows that the two programs return the same result before and after colouring by the colouring function.

The only thing different from a standard inductive proof is that we need to choose a permute oracle, i.e., we need to show that whenever we create a stack frame during the evaluation of the coloured program, there exists a permutation of the original stack frame so that its values are ordered in the same way.

The crucial argument is as follows: we assumed that f is injective, and in particular, that it is an injection between the variable names in the coloured and original stack frames. The state invariants imply that the value associated with each variable matches up across the colouring. Moreover, the variables kept in stack frames are determined by cut-sets, whose cardinality is the same in both coloured and original programs. Therefore, f implies the existence of a bijection between the positions of the two stack frames that forces their values to line up. An illustration of such a bijection is shown by the arrows in the example below. This bijection is precisely the permute oracle that we need to construct for the stack frame.



7.2.2 Extracting Clash Sets

Having abstractly characterised appropriate colouring functions, we next extract from the input WORDLANG program a simple tree-like control flow structure, where each instruction is reduced to the list of variables that it reads and writes. The tree structure is a convenient interface for the upcoming graph colouring step. It also helps decouple our graph colouring allocator from the rest of the CakeML development: the allocator can perform allocation on any program specified by only its read/write and control flow structure.

Correspondingly, we define and verify a colouring function checker that checks the aforementioned injectivity property over this tree. Unlike the abstract characterisation, this intermediary checker is designed to evaluate efficiently in the logic. We will explain how this intermediary is used later in Section 11. The verification of this checker shows that any colouring function that passes this checker also satisfies colouring_ok.

7.2.3 Graph Colouring Register Allocation

Finally, we implement a graph colouring register allocator that produces the actual colouring function from a clash graph. The clash graph is extracted from the aforementioned tree-like program structure.

A simple trick is used here to reduce the verification effort. Many classical graph colouring algorithms operate in two steps. First, the vertices of the input graph are ordered in some way, and then a second phase colours the vertices in that order. The trick is to observe that only the second phase needs to be verified, i.e., given any input order, the colour selector always picks colours such that none of the vertices have clashing colours[13].

Using this trick, we verified three increasingly complex variants of graph colouring algorithms and heuristics, including a version of Iterated Register Coalescing (IRC)-based allocation (George & Appel, 1996). Since the IRC allocator is relatively slow, a flag controls which of these allocators is used during compilation.

The correctness theorem here is connected back to our semantics theorem by showing that all the vertices in any clique of the clash graph are given distinct colours by the colouring function produced, and then showing that this implies the required injectivity property because all clash sets correspond to cliques in the graph.

7.3 SSA Form and Instruction Selection

Register allocation performance can be further improved by reducing the live ranges of the input program's variables. We achieve this by performing an *Static Single Assignment (SSA)*-like pass before register allocation. The resulting program is not strictly in SSA form because our semantics do not have ϕ-functions. Instead, we implicitly perform ϕ-elimination (replacing ϕ-functions with variable movement) directly inside the SSA pass. This retains the intended benefit of live range splitting for our variables, without the full verification cost of specifying a non-deterministic IL with ϕ-functions.

[13] We also verify that it generates the syntactic properties we need.

Since this transformation renames variables (like register allocation), we again have to provide permute oracles. The insight here is, similarly, to show that the SSA mapping defines an injective function from original variable names to variable names after SSA. This gives us a way to construct the bijection as required in the oracle style proof.

Instruction selection is another important pass within WordLang. It flattens arbitrary depth expression trees down to a sequence of instructions implementing that tree. The instructions need to make use of extra temporaries, but since we have an SSA pass, it uses the same temporaries throughout and relies on the SSA pass to appropriately rename the temporaries. We use a greedy maximal munch instruction selector that is parametrised by the target architecture's constraints, e.g., whether it only allows 2-register instructions, and the bounds on allowed memory operation constants. Additional expression-based optimisations are also performed within the phase, e.g., constant folding. Unlike the two aforementioned passes, this pass does not perform any variable renaming. Instead, it just introduces an extra temporary, and so its correctness theorem does not need to mention the permute oracle. The correctness theorem shows that the sequence of instructions picked for each expression correctly implements that expression, and that WordLang programs are invariant to extra temporaries not mentioned in the program. As we have seen, this forward simulation-style proof is just a special case of the permute oracle-style theorem, and so we can easily compose it with the other two correctness theorems.

7.4 Compiling Multiple Code Table Entries

All of the aforementioned WordLang compilation passes are performed in an intra-function manner without crossing function call boundaries, i.e., we can perform instruction selection, SSA and register allocation for each code table entry separately. Therefore, the complete compilation pass in the CakeML compiler simply maps the aforementioned passes over the entire input code table.

Here, we run into another difficulty with the permute oracle: the theorems that we have proved show that an oracle exists when executing an individual program with respect to a fixed code table. This oracle might well be different between different code table entries since, e.g., each entry may have a completely different colouring function. Thus, the local theorem does not immediately tell us that a single global oracle exists when the entire code table is transformed. An additional proof is required to construct a global permute oracle for any program after we have transformed the code table.

The essential correctness argument is as follows. Suppose we are currently evaluating a compiled WordLang program *prog* with its own permute oracle *perm*. At the point where another compiled code table entry $prog'$ is called, we have used some finite prefix p of $perm = p + q$. We use $+$ here for append on infinite sequences.

Furthermore, $prog'$ has its own permute oracle $perm'$ guaranteed by its local compilation correctness theorem. Now, if $prog'$ diverges, we simply use $p + perm'$ as the oracle. Otherwise, $prog'$ converges, and it has used some finite prefix p' of $perm' = p' + q'$. Then, we can construct $p + p' + p''$, where p'' is the oracle that we obtain from inductively repeating this argument for the rest of the evaluation of *prog*.

Note that an alternative way to verify this would have been to generalise all of our permute oracle-style theorem to account for compiling the entire code table. However,

such an approach would essentially have to repeat the above reasoning each time, and so we prefer to prove it only once globally after all the local theorems have been proved.

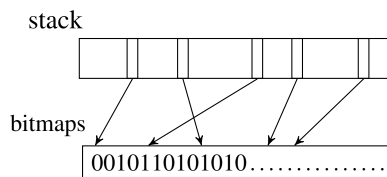
8 Compilation of Stack and Exceptions

The overall aim of STACKLANG, as its name suggests, is to support a concrete implementation of the stack. The STACKLANG transformations also implement the GC primitive as STACKLANG code.

8.1 An Array-like Stack

The translation from WORDLANG into STACKLANG compiles the abstract stack of WORDLANG into an array-like stack. Here, we implement the naming conventions used by the register allocator: WORDLANG names corresponding to stack variables are compiled into element lookups in stack frames, and those corresponding to registers are compiled into registers. In addition, we compile the parallel moves generated within WORDLANG down to single move instructions in STACKLANG[14].

Unlike stack frames in WORDLANG, stack frames in STACKLANG allocate enough space for all of the stack variables that may be used inside a function body. However, not all of these stack positions will be live at every call from the body and, in particular, it would be inefficient to sanitise all of the non-live positions in stack frames on every function call. Therefore, caller functions always write a number, represented as a single word, into the top entry of their stack frames[15]. This number is used to index into a bitmap table, which is separate from the stack, to obtain a bitmap that describes which positions in the stack frame are live at this point in time. When the GC is called, it looks up and decodes the bitmap for each stack frame, and then uses bitmap to consider only the variables that are live in the stack frame. These live positions correspond exactly to the original entries of the WORDLANG stack frames; recall that these entries were saved onto the stack so that the caller can restore them after its callee returns control.



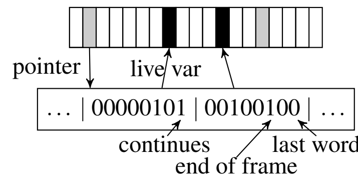
These bitmaps are designed to be as compact as possible. A bitmap can consist of multiple words. Each word except the last has its most significant bit set to one; in the last word, the most significant one bit represents the end of the frame being described.

[14] Our implementation and proof of the parallel moves compilation step is a HOL formalisation of Rideau *et al.* (2008).

[15] Our compiler writes such a bitmap-index number to the stack at every non-tail call. In contrast, GCs for conventional implementations tend to use return addresses stored in the stack to find the relevant bitmaps. At the time of writing, we are looking into switching to the conventional return-address-based indexing because that would make function calls a bit faster.

The payload of the bitmap, consisting of the remaining bits, has the same length as the length of the stack frame it describes. Each position in the bitmap tells the GC whether the corresponding index in the stack frame contains a live variable that the GC needs to process. Bitmaps are shared between call sites that happen to have the same bitmap layout.

The following diagram illustrates how the details of bitmaps are set up. Note that this illustration shows the most significant bit furthest to the right. The GC walks these bitmaps from left-to-right, from least-significant bit to most-signifiant bit. This illustration pretends that words are 8 bits. In reality they are 32 or 64 bits.



The StackLang semantics represents the bitmaps as a state component separate from the array-like stack which is also separate from the data heap. The bitmaps are moved into the state's memory component by a later transformation (Section 8.3).

In addition to concretising stack variables, the Word-to-Stack compiler also concretises the exception mechanisms. Stack frames with exception-handling information are converted to two stack frames: one for the variables part and one small frame for the handler information. The code for raising an exception rewinds the stack by simply assigning a stored value to the stack pointer and jumping to a stored code pointer. Installing exception handlers involves storing information about the current handler onto the stack before making a normal call to a function that holds the body of the handler expression.

The main verification difficulty in this step is to set up the appropriate state invariant between the abstract and concrete stacks. Our technique reconstructs an abstract stack (and local variables) from the concrete stack, and then defines a stack invariant between the two abstract stacks.

8.2 Implementation of the GC Primitive

StackLang's array-like stack and separate bitmap store provide a convenient level of abstraction for implementation and verification of the GC primitive. A simple compiler phase replaces every call to Alloc with a call to a library function, which we prove implements the GC. The GC implementation is parametrised by the data configuration specified in the compiler configuration.

We equip the state of the semantics with a switch which determines whether calls to the GC primitive in StackLang's state are allowed. We prove that the GC implementing transformation allows us to turn the switch off, forbidding calls to Alloc thereafter. We refer the interested reader to Ericsson *et al.* (2017), which describes the CakeML generational copying garbage collector and its proof in detail.

8.3 Moving the Stack and Bitmaps into Memory

The next transformation moves STACKLANG's stack and global variable store into memory. At the same time, the bitmaps are moved into a read-only data section. We assume that the bitmaps are correctly loaded into the data section when the program is executed.

Operations that interact with each of these primitive state components are implemented by one or two straightforward assembly instructions. This transformation turns off semantic switches, like the one for the GC primitive mentioned above. The result of the STACKLANG transformations is a structured program where only machine-instruction-like operations are permitted.

9 Compiling to Multiple Targets

Our compiler targets concrete machine code for multiple targets and supports a foreign-function interface (FFI). This section explains the final phases of the compiler and how the target specific details are factored in. For more details concerning our target models, e.g., how each assembly instruction is encoded down to bytes for a specific target, we refer the reader to Fox *et al.* (2017).

9.1 Abstract Machine Instructions

The compilation from DATALANG to WORDLANG is the first phase that reveals details specific to the target. This phase introduces the size of the machine words (either 64 or 32 bit), but is otherwise target independent.

The instruction selector, which runs right before register allocation, is the next phase to be affected by the target architecture. The instruction selector compiles WORDLANG's expressions into instructions of the datatype shown in Figure 5. The FP constructor is part of an ongoing effort to add floating-point support in CakeML.

The instructions that each target supports is a subset of these, e.g., no real target allows arbitrary sized constants in the immediate operands on arithmetic instructions. It is the job of the instruction selector to pick instructions that are acceptable for the target architecture. Each target architecture is described by a record with information about the target. This information is included in the compiler configuration.

After instruction selection, the register allocator picks register names and stack positions that fit within the number of registers allowed by the target. We chose to use our own naming schemes and calling conventions for most of the compiler in order to maintain uniformity throughout the interesting parts of the compiler.

The target-specific renaming of registers is performed as a STACKLANG-to-STACKLANG transformation, which occurs just before the compiler translates STACKLANG programs into a flat labelled assembly language. This renaming is no more than an application of a bijective renaming function to the names of the STACKLANG registers. The mapping ensures, for example, that CakeML's return address register (zero) gets mapped to the corresponding register of the target, e.g. register 14 on ARM. By the x86-64 calling convention, the return address is passed on the stack. The CakeML compiler ignores this convention internally, but adheres to it when calling external functions through the FFI.

```
α inst =
    Skip
  | Const num (α word)
  | Arith (α arith)
  | Mem memop num (α addr)
  | FP fp

memop = Load | Load8 | Store | Store8

α addr = Addr num (α word)

α arith =
    Binop binop num num (α reg_imm)
  | Shift shift num num num
  | Div num num num
  | LongMul num num num num
  | LongDiv num num num num num
  | AddCarry num num num num
  | AddOverflow num num num num
  | SubOverflow num num num num

cmp = Equal | Lower | Less | Test | NotEqual
    | NotLower | NotLess | NotTest

binop = Add | Sub | And | Or | Xor

α reg_imm = Reg num | Imm (α word)
```

Fig. 5. The instruction datatype.

9.2 Labelled Assembly Language

We use a flat labelled assembly language, called LabLang, as a stepping stone between reduced StackLang and concrete machine code. This assembly language has abstract syntax shown in Figure 6. A LabLang program consists of a list of sections (α `sec`). Each section has a name and contains a list of assembly lines (α `line`). Each line is either a label (Label), a simple assembly instruction (Asm), or a labelled assembly instruction (LabAsm).

Each line includes fields that can hold information about the byte encoding of the line. Label lines Label l_1 l_2 l mention the name of the label (l_1, l_2) and have a length l. This length field is non-zero if padding is required to align the value of the label to an even machine address. Certain labels need to be placed at even machine addresses in order for all code pointers to have to have their least significant bit set to zero, so that the garbage collector does not mistake code pointers for pointers to heap data. The simple (Asm) and labelled assembly lines (LabAsm) have a length field that simply records the length of their concrete byte encoding (8 `word list`).

9.3 Removal of Tick Instructions

Before LabLang programs are converted to concrete machine code, they go through a simple transformation that removes all skip instructions. Why are there skip instructions in the code at this stage of the compiler? The answer is that skip instructions are the result of compiling StackLang's Tick instructions into LabLang. Tick instructions are

```
α sec = Section num (α line list)
α line =
   Label num num num
  | Asm (α asm_or_cbw) (8 word list) num
  | LabAsm (α asm_with_lab) (α word) (8 word list) num
α asm_with_lab =
   Jump lab
  | JumpCmp cmp num (α reg_imm) lab
  | Call lab
  | LocValue num lab
  | CallFFI string
  | Install
  | Halt
lab = Lab num num
```

Fig. 6. LabLang abstract syntax.

a side effect of using a functional big-step semantics (Owens *et al.*, 2016). All compiler transformations thus far have produced code that ticks as much or more than the code before. Some transformations, such as function-inlining, introduce Tick expressions that artificially ensure generated programs tick more than the programs they were generated from. By removing the skip instructions in LabLang, we remove the artificial ticks.

The most interesting aspect of this proof is that it is the only proof we have that *goes against the direction of compilation*: we prove that adding back the removed skip instructions cannot change the observational semantics of the transformed program.

9.4 Concrete Machine Code

The compiler ends with a translation of LabLang programs into concrete machine code. The transformation starts by encoding all instructions in the LabLang program using an encoding function from the target configuration. It then enters an assembly loop which computes the location of all labels and re-encodes all jumps and other label-dependent instructions.

The encoders may emit jump instructions of different lengths, depending on the precise jump distances required. Increasing the length of a jump encoding can, in turn, invalidate the jump distance for other jumps and so we perform the label computation and jump encoding steps in a loop. In this way, our assembler effectively performs a limited form of branch relaxation, where it optimistically starts with shorter jump encodings before moving to longer ones if required. We note that this loop can only increase the length of jump encodings, and thus it terminates because every jump has a maximum encoding length.

The instruction encodings are stored in the syntax of the LabLang program. On exit from the loop above, the compiler checks that all instructions (jumps in particular) are encodable with the assigned arguments (e.g. jump lengths). If they are not encodable, then the compiler returns an error. Otherwise, the compiler returns a list of bytes that is the concatenation of all byte-list annotations in the LabLang program.

9.5 Target Semantics

The correctness of the LabLang-to-target compiler is proved with respect to the target semantics. The target is given a functional big-step semantics with evaluate and semantics functions similar to the languages above. The evaluate function for the target is split into two layers. First, we have the target instruction-set-architecture's next-state function and state type. On top of this, we define a second layer which is the evaluate function that executes the next-state function in the presence of an interference oracle and the FFI interface. The definition is too long to be shown here, so we explain it informally. The evaluate function operates as follows:

- If the clock has hit zero, exit with a *timeout*.
- Decrement the clock.
- Read the program counter's value *pc* from the machine state.
- If *pc* is a memory address within the region for the generated machine code, then execute the target's next-state function followed by an environment interference function (which is allowed to change any state outside of the CakeML processes registers and memory).
- If *pc* is the exit address, then stop; return *success* if the return value is 0, otherwise raise *resource-bound-hit*.
- If *pc* is an FFI entry point, then execute the FFI semantics according to the current FFI state, followed by an application of an FFI interference oracle which can arbitrarily change the state of the caller-saved registers, etc.
- In all other cases, *fail*.

The environment interference oracle is run in between every target machine instruction; it can arbitrarily update parts of memory that are irrelevant to the CakeML process. We have such an oracle to model the interference of an operating system, which can interrupt and later restore the CakeML process's execution at any time.

9.6 Correctness of the Assembler Function

We prove that all well-annotated LabLang programs (i.e., ones that have passed the exit condition for the loop described above, Section 9.4) will flatten to a byte list that executes on the target machine with an equivalent observable semantics.

In order to make this proof manageable, with support for multiple targets, we decoupled the target-specific proof from LabLang by having another abstraction layer. We define the following abstract syntax for non-labelled assembly instructions, and prove for each target that any target-specific encoding of these will produce a simulation of the abstract instruction using the target machine's next-state function and environment interference oracle. The environment oracle comes into play here because some abstract instructions are encoded using multiple instructions in the target architecture. For example, loading a large constant requires some target- and constant-dependent number of instructions: 1–6 for MIPS; 1–4 for RISC-V; 1–4 for ARMv8; 1–2 for ARMv6; and just one for x86-64. The

environment interference oracle is allowed to alter the state midway through this execution.

$$
\begin{array}{l}
\alpha\ \texttt{asm} = \\
\quad \mathsf{Inst}\ (\alpha\ \texttt{inst}) \\
\quad |\ \mathsf{Jump}\ (\alpha\ \texttt{word}) \\
\quad |\ \mathsf{JumpCmp}\ \texttt{cmp}\ \texttt{num}\ (\alpha\ \texttt{reg_imm})\ (\alpha\ \texttt{word}) \\
\quad |\ \mathsf{Call}\ (\alpha\ \texttt{word}) \\
\quad |\ \mathsf{JumpReg}\ \texttt{num} \\
\quad |\ \mathsf{Loc}\ \texttt{num}\ (\alpha\ \texttt{word})
\end{array}
$$

The LABLANG-to-target compiler's proof lifts per instruction simulations to a simulation result for the entire LABLANG program.

10 Top-level Correctness Theorem

In this section, we present two compiler correctness theorems for the CakeML compiler. The first concerns the compiler backend, i.e., the composition of all the compilation passes, including all those we have discussed previously, that operate on abstract syntax. The second theorem — our main result about the compiler function — concerns, in addition, our verified frontend (lexing, parsing, and type inference), and proves semantics preservation for the whole compiler. We end by defining the user interface for the compiler, which will be used in the next section on bootstrapping.

10.1 Backend Correctness

The compiler backend, which composes all compilation passes, is a function called compile. Its correctness theorem is shown in Figure 7.

$$
\begin{array}{l}
\vdash \mathsf{compile}\ cc\ prog = \mathsf{Some}\ (code,data,cc') \wedge \neg\mathsf{semantics}\ ffi\ prog\ \mathsf{Fail} \wedge \\
\quad \mathsf{backend_config_ok}\ cc \wedge \mathsf{mc_conf_ok}\ mc \wedge \mathsf{mc_init_ok}\ cc\ mc \wedge \\
\quad \mathsf{installed}\ code\ data\ cc'.\mathsf{ffi_names}\ ffi\ (\mathsf{heap_regs}\ cc.\mathsf{stack_conf.reg_names})\ mc\ ms \Rightarrow \\
\quad\quad \mathsf{machine_sem}\ mc\ ffi\ ms \subseteq \mathsf{extend_with_resource_limit}\ (\mathsf{semantics}\ ffi\ prog)
\end{array}
$$

Fig. 7. Backend compiler correctness theorem.

On success, the compiler backend returns a 3-tuple, $(code,data,cc')$. The first component, *code*, is a list of bytes: the binary-level implementation of the input program, *prog*, for the chosen architecture. The second component, *data*, is a list of machine words to be loaded into the data section (see Section 8.3). The final component, cc', consists of additional book-keeping information about the compilation. The most important piece is the cc'.ffi_names field, which lists the names of FFI ports that the code assumes it has access to.

There are several assumptions made in the backend correctness theorem. After the assumption that the input program does not Fail, which will be discharged in the top-level theorem, the next three assumptions are essentially well-formedness conditions on the input configurations:

- backend_config_ok checks that the initial compiler configuration *cc* is sound.

- mc_conf_ok and mc_init_ok check that the machine configuration is well-formed, and that the compiler configuration matches the machine configuration. For example, it would not make sense to have an ARMv8 compiler configuration producing code to run on an x64 machine.

All three of these assumptions can be discharged by instantiating the theorem with a concrete choice of target and configuration. For each compilation target, we provide a default configuration that satisfies the well-formedness conditions. One such instantiation for the x64 backend is shown in Figure 8.

$$\begin{array}{l}\vdash \mathsf{compile}\ \mathsf{x64_backend_config}\ \mathit{prog} = \mathsf{Some}\ (\mathit{code},\mathit{data},\mathit{cc}') \wedge \\ \quad \neg\mathsf{semantics}\ \mathit{ffi}\ \mathit{prog}\ \mathsf{Fail} \wedge \mathsf{is_x64_machine_config}\ \mathit{mc} \wedge \\ \quad \mathsf{installed}\ \mathit{code}\ \mathit{data}\ \mathit{cc}'.\mathsf{ffi_names}\ \mathit{ffi}\ (6,1)\ \mathit{mc}\ \mathit{ms} \Rightarrow \\ \quad\quad \mathsf{machine_sem}\ \mathit{mc}\ \mathit{ffi}\ \mathit{ms} \subseteq \mathsf{extend_with_resource_limit}\ (\mathsf{semantics}\ \mathit{ffi}\ \mathit{prog})\end{array}$$

Fig. 8. Backend compiler correctness theorem instantiated for the x64 backend.

The final assumption is the installed predicate. It checks that the machine state *ms*, and machine configuration *mc* are set up correctly with respect to the output code, the data section, and the FFI. As an example, for the theorem in Figure 8, installed assumes that register 6 corresponds to the first heap address, and that the address of the start of the data section is stored in the first heap address; register 1 corresponds to the first address past the end of the stack. Each of these assumptions turns into a corresponding part of the (unverified) wrapper assembly around our generated bytes. A code snippet for our wrapper assembly is shown in Figure 9. Note that in the register encoding scheme for x64, `rcx` and `rsi` correspond to registers 1 and 6 respectively. In the conclusion of the theorem, extend_with_resource_limit adjusts the behaviours set to allow early exit on the `outcome` which signals a *resource-limit-hit*.

10.2 Top-level Compiler Correctness

Using the backend correctness theorem, we prove a top-level correctness theorem relating the source semantics, the CakeML compiler, and the target semantics.

The top-level semantics of CakeML, cakeml_semantics, is defined as follows based on the specification of the parser, the specification of what is well-typed, and the observable semantics of executing a CakeML program. The *prelude* argument is for the CakeML basis library, which is included at the start of every compiled program.

$$\begin{array}{l}\mathsf{cakeml_semantics}\ \mathit{ffi}\ \mathit{prelude}\ \mathit{input} = \\ \quad \mathsf{case}\ \mathsf{parse}\ (\mathsf{lex}\ \mathit{input})\ \mathsf{of} \\ \quad\quad \mathsf{None} \Rightarrow \mathsf{CannotParse} \\ \quad | \ \mathsf{Some}\ \mathit{prog} \Rightarrow \\ \quad\quad \mathsf{if}\ \mathsf{can_type_prog}\ (\mathit{prelude} +\!\!+ \mathit{prog})\ \mathsf{then} \\ \quad\quad\quad \mathsf{Execute}\ (\mathsf{semantics}\ \mathit{ffi}\ (\mathit{prelude} +\!\!+ \mathit{prog})) \\ \quad\quad \mathsf{else}\ \mathsf{IllTyped}\end{array}$$

We define the CakeML compiler, cakeml_compile, correspondingly using our implementations of the parser, the type inferencer, and the compiler backend.

```
cake_heap:
  ...                             # Space allocated for CakeML's heap
cake_stack:
  ...                             # Space allocated for CakeML's stack
cake_bitmaps:
  ...                             # CakeML's data section (containing bitmaps)

main:
   ...                  # save command line arguments
   pushq   %rbp         # push base pointer
   movq    %rsp, %rbp   # save stack pointer
   movabs  $cake_main, %rdi        # arg1: entry address
   movabs  $cake_heap, %rsi        # arg2: first address of heap
   movabs  $cake_bitmaps, %rdx
   movq    %rdx, 0(%rsi)           # store bitmap pointer
   movabs  $cake_stack, %rdx       # arg3: first address of stack
   movabs  $cake_end, %rcx         # arg4: first address past the stack
   jmp     cake_main

  .p2align 4

cake_ffi...:
  ...                             # FFI entry points

cake_main:
  ...                             # The generated bytes are printed below
```

Fig. 9. Snippet of x64 wrapper assembly. The wrapper sets up the initial CakeML environment according to the assumptions arising from installed.

The final top-level correctness theorem for the compiler is shown in Figure 10. The type system guarantees that well-typed programs do not have runtime errors, so the associated assumption in Figure 7 is discharged.

10.3 Top-level Compiler with User Interface

There is still a gap between the top-level compiler, as defined above, and a usable compiler that runs natively on a target machine. The first step towards closing this gap is to define the user interface of the compiler. We define a function, compiler_ui, that wraps around cakeml_compile and adds the user interface. Here we describe its definition informally:

1. In addition to the input program, compiler_ui reads and parses a configuration string for the compiler. The configuration string allows end users to, e.g., choose the compilation target.
2. If the configuration string parses successfully, then compiler_ui calls cakeml_compile with the parsed configuration and the input program.
3. On successful compilation, compiler_ui returns a formatted string containing the generated bytes and data section appropriately wrapped in the unverified assembly for the chosen target, e.g., as shown in Figure 9 for the x64 target.
4. On failure in any of the steps above, compiler_ui returns a formatted error string instead.

```
⊢ inf_conf_ok cc.inferencer_config ∧ ¬cc.input_is_sexp ∧ ¬cc.exclude_prelude ∧
  ¬cc.skip_type_inference ∧ backend_config_ok cc.backend_config ∧
  mc_conf_ok mc ∧ mc_init_ok cc.backend_config mc ⇒
   case cakeml_compile cc prelude input of
    Success (code,data,cc′) ⇒
     ∃behaviours.
       cakeml_semantics ffi prelude input = Execute behaviours ∧
       ∀ms.
         installed code data cc′.ffi_names ffi
           (heap_regs cc.backend_config.stack_conf.reg_names) mc ms ⇒
           machine_sem mc ffi ms ⊆ extend_with_resource_limit behaviours
   | Failure ParseError ⇒ cakeml_semantics ffi prelude input = CannotParse
   | Failure (TypeError _) ⇒ cakeml_semantics ffi prelude input = IllTyped
   | Failure CompileError ⇒ true
   | Failure (ConfigError _) ⇒ true
```

Fig. 10. Top-level compiler correctness theorem.

We emphasise that compiler_ui is *not verified* in the same sense as the backend and top-level compilers discussed above, and it does not have a corresponding correctness theorem. Verifying it would require verifying the small assembly wrapper that is included in the formatted output string. This would require formalising the linking system, which we have not done. Instead, we view compiler_ui as a functional specification of the top-level user interface of the compiler. In the next section, we explain how we generate *machine code* that is verified to implement this functional specification.

11 Compiler Bootstrapping

A unique feature of the CakeML compiler is that it is bootstrapped "in the logic" – essentially, an application of the compiler function with the compiler's source implementation as argument is evaluated via logical inference. This bootstrapping method produces a machine code implementation of the compiler and also proves a theorem stating functional correctness of that machine code. Bootstrapping removes the need to trust an unverified code generation process. By contrast, CompCert first relies on Coq's extraction mechanism to produce OCaml, and then relies on the OCaml compiler to produce machine code.

The original bootstrapping process described in Kumar *et al.* (2014) and Kumar (2016) takes most of the CakeML compiler from its definition in HOL down to a bytecode implementation. Then the verified bytecode-to-x64 compilation pass is evaluated in the logic on the bytecode implementation to produce machine code for most of the compiler. However, the bytecode-to-x64 pass itself is not bootstrapped along with the rest of the compiler. Instead, its x64 implementation is manually verified via decompilation into logic (Myreen *et al.*, 2012).

For the new compiler, we have extended the bootstrapping process down to the machine code-level, i.e., the full compiler now compiles and assembles itself, in the logic, down to machine bytes. We have also improved the description of the I/O interface using our mechanisation of characteristic formulae (CF) for CakeML (Guéneau *et al.*, 2017). The CF framework provides a separation logic that supports reasoning about (among other

things) FFI models. This allows us to verify CakeML code that makes FFI calls against the specification of the relevant FFI models. The bootstrapped CakeML compiler, for example, uses FFI calls to access its command line arguments, and to read/write to the standard input, output, and error streams.

In this section, we review the verified bootstrapping technique before describing the challenges faced in bootstrapping the new compiler.

11.1 Verified Bootstrapping

Our bootstrapping technique consists of two steps: proof-producing translation, and symbolic evaluation. The first involves creating a CakeML program whose semantics is proved to implement the compiler function. The second involves applying the compiler function to that CakeML program to produce a verified machine-code implementation.

Proof-producing Translation

We use proof-producing translation (Myreen & Owens, 2014) to convert the compiler definition in HOL into an implementation in CakeML source code. In particular, we translate the compiler_ui function from Section 10.3 (including all its dependencies). As we will see in Section 11.2, the generated CakeML program differs depending on whether we are translating the compiler for 64-bit targets or for 32-bit targets, so we translate two versions of compile_ui. We shall focus on the 64-bit version, which we label as compile_ui_64. The translation is mostly automatic, and produces a certificate theorem that relates the semantics of the generated CakeML implementation with the input HOL function (compile_ui_64).

HOL functions are mathematical functions, so it makes no sense for them to obtain their arguments from I/O operations or to print their return values. But we want an I/O interface for the compiler, so we write an effectful wrapper directly in CakeML code: `main_64`, shown below (the version for 32-bit targets is similar). It reads command line arguments using our `CommandLine` library and then calls the translated implementation of the compiler user interface (`compiler_ui_64`) with the input program read from standard input. It prints the formatted bytes (`print_app_list ls`) to standard output, and any compiler errors to standard error (`print_err e`).

```
fun main_64 u =
  let
    val args = CommandLine.arguments ()
  in
    case compiler_ui_64 args
           (String.explode (TextIO.inputAll TextIO.stdIn)) of
    (ls, e) => (print_app_list ls; TextIO.output TextIO.stdErr e)
  end
```

We can now define a whole CakeML program, main_64_prog, that consists of declarations up to the `main_64` function above followed by a call to `main_64` (applied to unit). Using characteristic formulae (CF) for CakeML for results about the I/O wrapper, and the

results generated by the proof-producing translation process for the rest, we can prove a theorem about this whole CakeML program:

$$
\begin{array}{l}
\vdash \mathsf{wfcl}\ cl \wedge \mathsf{stdFS}\ fs \Rightarrow \\
\quad \exists io_events. \\
\qquad \mathsf{semantics_prog}\ (\mathsf{init_state}\ (\mathsf{basis_ffi}\ cl\ fs))\ \mathsf{init_env}\ \mathsf{main_64_prog} \\
\qquad\ \ (\mathsf{Terminate}\ \mathsf{Success}\ io_events) \wedge \\
\qquad \mathsf{extract_fs}\ fs\ io_events = \\
\qquad\ \ \mathsf{Some}\ (\mathsf{compiler_ui_64_spec}\ (\mathsf{tl}\ cl)\ (\mathsf{get_stdin}\ fs)\ fs)
\end{array}
$$

The theorem states that running main_64_prog in CakeML's source semantics from an appropriate initial state always terminates successfully with a list of I/O events. Moreover, examining these I/O events gives us exactly the output of compiler_ui_64_spec called with the command line arguments and input from standard input. This compiler_ui_64_spec function simply describes the effect of printing the return values of compiler_ui_64 on standard output and standard error. Thus, we now have a CakeML program that is verified to have the desired I/O behaviour of the CakeML compiler.

Symbolic Evaluation

For the second step, we make use of HOL4's symbolic evaluation mechanism to evaluate the CakeML compiler backend, i.e., the HOL function compile, on the input program main_64_prog. Importantly, this mechanism for evaluation in the logic produces a theorem, which is of the following form:

$$
\begin{array}{l}
\vdash \mathsf{compile}\ \mathsf{x64_bootstrap_config}\ \mathsf{main_64_prog} = \\
\quad \mathsf{Some}\ (\mathsf{cake_code},\mathsf{cake_data},\mathsf{cake_config})
\end{array}
$$

For ease of presentation, we have defined constants cake_code, cake_data, and cake_config for each component of the result of symbolic evaluation. For example, the cake_code constant abbreviates the list of bytes comprising the machine-code implementation of the compiler. The input compiler configuration, x64_bootstrap_config, is a slight modification of our default configuration, x64_backend_config, with some options (such as the inlining size limit) tweaked. Observe that for bootstrapping we evaluate the compiler backend (compile), but not the top-level compiler (cakeml_compile) that has parsing and type inference. Parsing is unnecessary because the input, main_64_prog, is already CakeML abstract syntax produced by translation. Type inference is unnecessary because the CF theorem shown earlier guarantees that main_64_prog does not have runtime errors.

The Bootstrapping Result

Composing the two aforementioned theorems together with our backend compiler correctness theorem from Figure 7 yields the overall correctness theorem, shown in Figure 11, about the machine-code implementation of the CakeML compiler. The composition is straightforward, except we need to slightly tweak the input compiler configuration to account for our bootstrapping x64 compiler configuration. Note, also, that the installed predicate now has concrete instantiations for its *code*, *data*, cc' and *ffi* arguments.

At a high level, the theorem in Figure 11 states that running the machine code implementation of the compiler produces the same output as would a run of the compiler in

```
⊢ wfcl cl ∧ stdFS fs ∧ is_x64_machine_config mc ∧
  installed cake_code cake_data cake_config.ffi_names (basis_ffi cl fs) (6,1)
   mc ms ⇒
   ∃io_events.
     machine_sem mc (basis_ffi cl fs) ms ⊆
     extend_with_resource_limit { Terminate Success io_events } ∧
     extract_fs fs io_events =
      Some (compiler_ui_64_spec (tl cl) (get_stdin fs) fs)
```

Fig. 11. Bootstrapped compiler correctness theorem.

the logic. We have used our compiler correctness theorem once to obtain this result. One might consider using the compiler correctness theorem *again*, to prove that the machine semantics of the *output* of the bootstrapped compiler is constrained by CakeML's source semantics. Specifically, this would involve reading the filled-in assembly wrapper from standard output, processing it so it can be fed back into installed and considering the machine_sem of that machine configuration. We have not proved such a result yet, mainly because the gains do not seem high without a formalisation of linking and loading. The original CakeML compiler involved this kind of result, however, because its read-eval-print loop ran the compiler at runtime.

11.2 Extending the Translator

The first step of bootstrapping involves proof-producing translation of HOL definitions to CakeML implementations. In this section, we describe an extension to the tool that does this job, the translator, that was motivated by the new multi-target compiler backend. We updated the translator to include primitive support for machine words (of all sizes up to 64 bits, implemented by CakeML's 64-bit words). This change was necessary for efficient translation of the compiler because all of our ILs from WordLang onwards require a word type. Furthermore, many of the compilation steps, e.g., Data-to-Word and the target encoding phase, utilise word operations.

The difficulties we faced here mostly revolved around sufficiently extending the translator automation to cover all the required cases. For example, to reduce the complexity of implementation, we only added a subset of standard word operations to the CakeML source language[16]. As a result, some of the word operations in our HOL implementation do not have a natural target in CakeML itself. We supplied manual rewriting theorems in order to turn these operations into ones that are actually in the source language.

As an example, consider the following definition of e_imm8. It conditionally extracts the last 8 bits from a 64-bit input, and it is used in the x64 encoder.

```
⊢ e_imm8 imm =
    if 0xFFFFFFFFFFFFFF80w ≤ imm ∧ imm ≤ 127w then
     [(7 >< 0) imm]
    else []
```

[16] Any new word operations would have to be propagated and supported throughout the compiler.

However, the extraction operation (><) does not have a natural target in CakeML. Instead, the rewritten theorem below only targets standard word operations.

$$\begin{array}{l} \vdash \mathsf{e_imm8}\ imm = \\ \quad \mathsf{if}\ 18446744073709551488w \leq imm \wedge imm \leq 127w\ \mathsf{then} \\ \quad\ [\mathsf{w2w}\ (imm\ \&\&\ 255w)] \\ \quad \mathsf{else}\ [\,] \end{array}$$

A more insidious example arises because we kept the word size parametric – we sometimes used definitions that are dependent in the word size type parameter such as the following in the DATA-to-WORD compiler:

$$\vdash \mathsf{shift}\ (:\alpha) = \mathsf{if}\ \mathsf{dimindex}\ (:\alpha) = 32\ \mathsf{then}\ 2\ \mathsf{else}\ 3$$

Neither HOL4 nor CakeML are dependently typed so these do not fit naturally into the translation procedure. Fortunately, these definitions are mostly for convenience when writing the definition and proofs of the compiler in HOL4. We in-lined them wherever they occurred after instantiating the type parameter appropriately.

11.3 Evaluation in the Logic

The new compiler has more optimisations and phases than the first CakeML compiler and it has become much harder for us to achieve a reasonable time for compiling the compiler in the logic. The increased difficulty comes from two directions: any new pass that we add to the compiler increases the number of translated definitions that have to be compiled, and the pass itself needs to be run during in-logic compilation. Yet, having a reasonable bootstrapping mechanism is necessary – we do not want bootstrap times to be a major stumbling block when we implement new compiler optimisations in the future.

A trivial option is to simply turn off or reduce the aggressiveness of some optimisations during the bootstrapping process. Of course, turning everything off would lead to an extremely slow bootstrapped compiler. We currently do this rather judiciously, e.g., we in-line slightly less aggressively during the bootstrap process.

We used three other techniques, and an improvement to the compiler, in order to reduce bootstrap compilation times.

11.3.1 Translation Validation

Following the terminology of Leroy (2009), translation validation is a process where the output of an untrusted compilation step is verified by a validator. This can reduce the verification effort because one only has to verify that the validator is sound, i.e., that it only accepts correctly compiled programs. But translation validation can also be used for runtime efficiency: untrusted steps can be run separately using a more efficient implementation. The CompCert compiler, for example, performs register allocation with unverified OCaml code before checking the result.

For bootstrapping the CakeML compiler, we found constructing the clash graph and running the full IRC algorithm during the symbolic evaluation part of bootstrapping to be infeasible. So we set up our register allocator and its correctness theorems to support a validation procedure.

More precisely, our register allocator takes an optional list of oracle colouring functions. Whenever it is called with such a list of oracles, it validates that they correctly colour the program being allocated. If validation succeeds, then the allocator uses those colourings. Otherwise, it calls the graph colouring algorithm that we have proved always produces a correct colouring.

A pleasing consequence of our bootstrapping procedure is that the translation step produces CakeML code that can be used independently of its verification. In particular, our register allocator is translated into CakeML during bootstrapping. We pretty printed this implementation to SML concrete syntax[17] for use as the external efficient implementation to be validated in the evaluation step of bootstrapping.

We only use translation validation for efficiency when evaluating the compiler in the logic – the bootstrapped compiler calls the allocator with no oracle functions, so it runs the verified graph colouring algorithm directly. Furthermore, the use of the external register allocator during bootstrapping has no impact, apart from the time required, on the result of symbolically evaluating the compiler. In other words, the result is a proven theorem with no extra assumptions.

11.3.2 Parallel Compilation

Following the compilation into BVL and closure conversion, the program is represented as a list of code table entries. Many of the subsequent compilation phases work independently on each code table entry – their definition simply maps some inner auxiliary function over the code table.

For all such phases, we run the in-logic compilation of code table entries in parallel. The compilation theorems for each code table entry are then lifted back up to a compilation theorem for the entire phase. To be able to do this, we need a result such as

$$\vdash f\ x_1 = v_1 \land \ldots \land f\ x_n = v_n \Rightarrow \mathsf{map}\, f\ [x_1;\ \ldots;\ x_n] = [v_1;\ \ldots;\ v_n]$$

We cannot apply this technique when there are dependencies between sub-computations since we cannot join their results using a theorem like the one above.

11.3.3 Encoder Memoisation

The most expensive phase in the entire bootstrap process is the assembly phase of the compiler. Each assembly instruction has to be encoded down into a list of bytes, and the encoding steps are directly evaluated in the logic. Here, a speed up was achieved by memoising the encodings of common instructions and the use of specialised evaluation theorems.

Note also that our assembler is inherently sequential because it needs to look at the entire code table in order to properly adjust jump labels. This makes it difficult to properly parallelise the process.

[17] This is convenient since SML is the implementation language for HOL4.

11.3.4 Improved Assembly Loop

In the conference version of this publication (Tan *et al.*, 2016), we stated that the assembler makes several syntactic checks on its output assembly. These syntactic conditions are necessary for the assembler's correctness, for example, they might check that none of the instructions mention a reserved register, or that the immediate constants are within bounds for each target.

We have since proved that many of these conditions are guaranteed by our compiler e.g., the instruction selector never introduces illegal immediate constants, and so they do not need to be explicitly checked by the assembler. The remaining checks are done after assembly, where we check that the generated jump offsets are within range for the target architecture. This check is difficult to avoid, because the jump offsets are only known at assembly time.

12 Benchmarks

We evaluated our new compiler backend's performance with a series of performance tests based on a subset of MLton's benchmark suite obtained from the MLton repository[18]. We excluded the benchmarks that:

- use records since they are not currently supported in CakeML, or
- perform system calls, since CakeML uses its own FFI implementations for these system calls, or
- use the basis libraries for machine words and floating point.

The remaining 16 benchmarks were slightly modified to fit CakeML's requirements. The changes are summarized as follows:

- Minor syntactic changes, e.g., CakeML does not allow capitalised variable names.
- Curried instead of tupled functions, since CakeML currently optimizes heavily for curried functions, but not for tupled functions.

We roughly split the benchmarks into 4 categories, depending on the size of the benchmark, and whether it uses any imperative features (e.g., references, arrays and vectors):

- (Small, Pure) `even-odd`, `fib`, `merge`, `tailfib`, `tak`
- (Small, Imperative) `flat-array`, `imp-for`, `vector-concat`, `vector-rev`
- (Large, Pure) `knuth-bendix`, `life`, `pidigits`
- (Large, Imperative) `logic`, `mpuz`, `ratio-regions`, `smith-normal-form`

All tests were executed on a laptop running Ubuntu 16.04, with an Intel® Core™ i7-6820HQ @ 2.70GHz CPU, and 16GBs of RAM. Additional details relevant to each set of tests are provided in their respective subsections. Our versions of the benchmarks can be found in the CakeML repository.[19]

The compiler bootstrapping process discussed in Section 11 requires roughly 9 hours for the translation step, and between 24 to 30 hours for evaluation in the logic. The resulting bootstrapped compiler is used to compile all of the benchmarks in this section.

[18] `https://github.com/MLton/mlton/tree/master/benchmark`
[19] `https://code.cakeml.org/tree/version2/compiler/benchmarks/mlton_benchmarks`

12.1 Comparison with Other ML Implementations

We benchmarked the CakeML compiler (with all optimisations enabled) against other ML compilers. The compilers and their corresponding versions are:

- (MLton) Version 20171211 (MLton developers, 2017).
- (Poly/ML) Version 5.7 (Matthews, 2017).
- (SML/NJ) Version 110.78 (SML/NJ developers, 2017).
- (Moscow ML) Version 2.10 (Romanenko *et al.*, 2013).
- (CakeML) Our compiler with all optimisations enabled.

CakeML does not currently support a distinguished small integer type, i.e., all of CakeML's integers are unbounded. The implementation switches to bignum arithmetic when the integers involved no longer fit within a single word. We could only configure the Poly/ML and MLton compilers to use a default unbounded integer type. In Figure 12, we compare the performance of CakeML against these two compilers configured to use unbounded integers. We *emphasize* that Figure 12 provides a fairer comparison against CakeML's default integer type. For completeness, we have also included the benchmark performance against all the ML compilers (without unbounded integers) in Figure 13.

For the small, purely functional benchmarks, our new compiler's performance is relatively close to the other compilers. It is, in fact, the fastest on the `fib` and `tailfib` benchmarks with unbounded integers. These two benchmarks both compute Fibonacci numbers, however, the integers involved all fit within CakeML's small integer width, and so our compiler uses efficient small integer arithmetic instead of bignum arithmetic.

In contrast, the `pidigits` and `smith-normal-form` benchmarks make essential use of the unbounded integer type[20]. In this case, MLton performs far better than any other compiler because it uses GMP (Granlund *et al.* , 2017) directly for its bignum computations. Note that Moscow ML failed to compile these benchmarks, while SML/NJ was killed after executing for $>$200s. Our compiler successfully compiles and runs these benchmarks, but its performance can be improved by connecting to verified bignum library implementations directly at the assembly level for a specific architecture.

On the other hand, there is clearly room for improvement on both small and large imperative benchmarks. Our compiler does not currently optimise heavily in these cases. For example, we do not currently have any optimisations to remove bounds checks on arrays and vectors.

Finally, all 3 of the large, purely functional benchmarks make use of tupling and tupled constructors which we could not simply remove by currying functions. Our CLOSLANG optimisations are mainly focused on curried functions, and so these cases are not well supported.

12.2 Garbage Collection

Although the benchmarks are not meant to stress test the GC, CakeML's (verified) GC could also cause some slowdown compared to the highly tuned GCs available in other

[20] They use the `IntInf` structure in SML's basis.

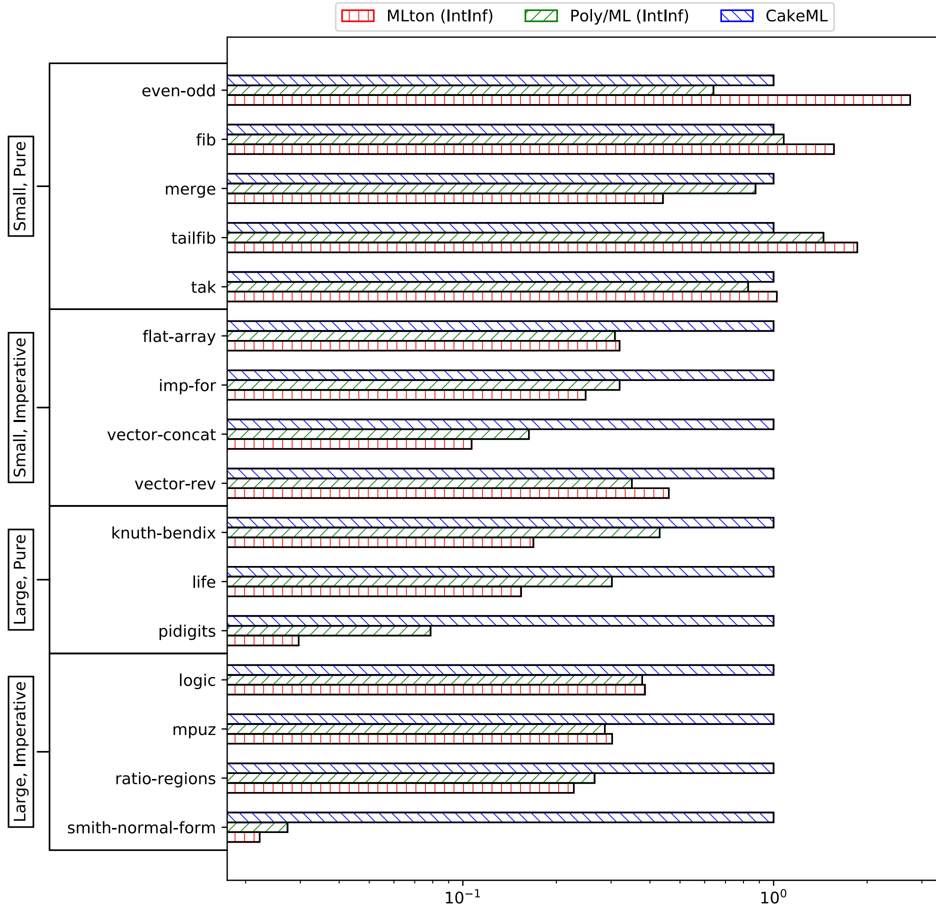


Fig. 12. Average execution times over 20 executions of the benchmarks for ML implementations with default unbounded integers, relative to CakeML.

compilers. Thus, we instrumented the CakeML compiler to record GC times for these benchmarks. The default CakeML heap size is 1000MB, and we progressively reduced it to 100MB and 10MB. The results are shown in Figure 14. Note that we have *removed* logarithmic scaling for the plot in order to better illustrate the GC times.

We first note that at the default heap size (1000MB), none of the benchmarks spend a significant amount of time running CakeML's GC. Thus, the results in the previous subsection are mainly due to compiler performance, rather than the GC. In particular, since we use a direct-style compiler, we do not incur as much heap allocation even on the benchmarks that make deeply-nested non-tail-recursive function calls. This is to be contrasted with CPS compilation where such function calls incur additional GC costs.

The results for smaller heap sizes are mostly as expected: CakeML progressively spends more time in the GC as heap sizes are decreased. On three of the benchmarks (`merge`, `vector-rev`, `ratio-regions`), CakeML runs out of memory when started with only 10MB of heap. It is important to note that this is *allowed* to happen by our compiler

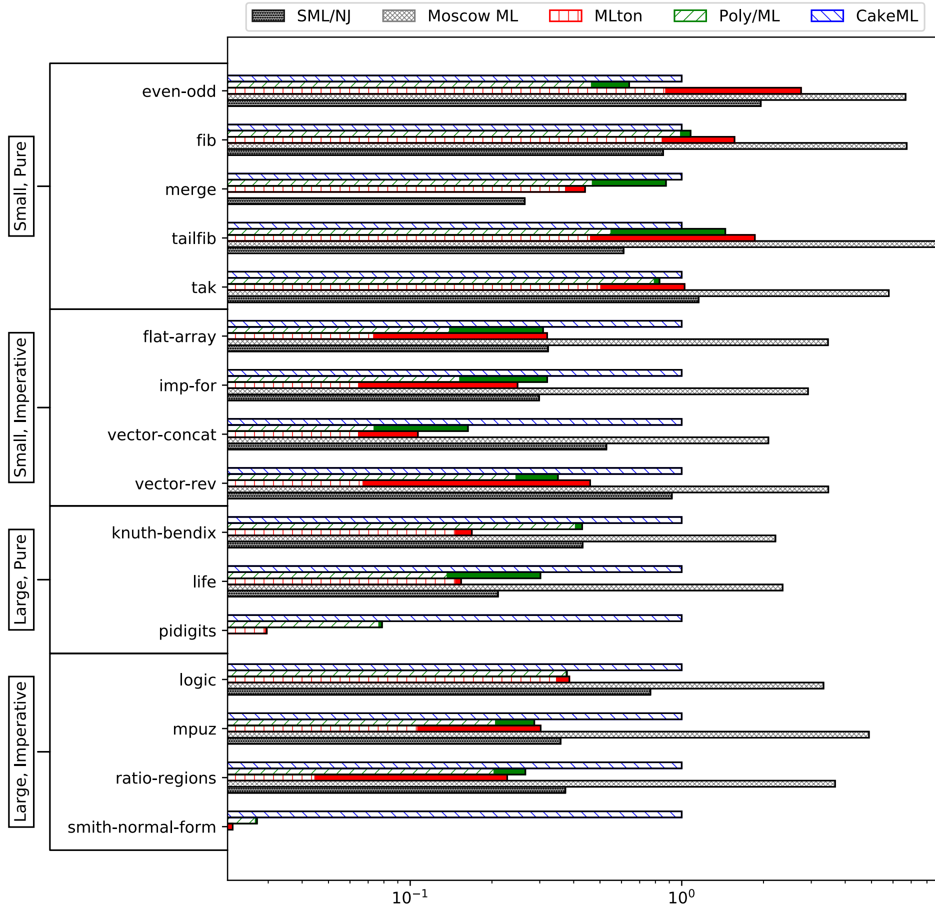


Fig. 13. Average execution time over 20 executions of the benchmarks for different ML implementations, relative to CakeML. Missing bars for SML/NJ and Moscow ML indicate that the benchmark either failed to compile, or did not terminate within 200s. The filled part of each bar for MLton and Poly/ML indicate extra time taken using unbounded integers.

correctness theorem, because we are compiling from source CakeML, which does not have a notion of finite machine memory, down to a target with limited resources.

The default CakeML garbage collector we used for all benchmarks is a copying collector (Myreen, 2010). We have recently verified a new, generational garbage collector (Ericsson *et al.*, 2017), at the point of writing, however, we have not yet tuned its performance heavily and so we have not included it in these benchmarks.

12.3 Comparison Across Compiler Optimisations

We study the utility of our compiler optimisations by selectively disabling groups of optimisations and examining the impact on benchmark performance. The 4 configurations are as follows:

- (CO) The CLOSLANG optimisations are disabled.

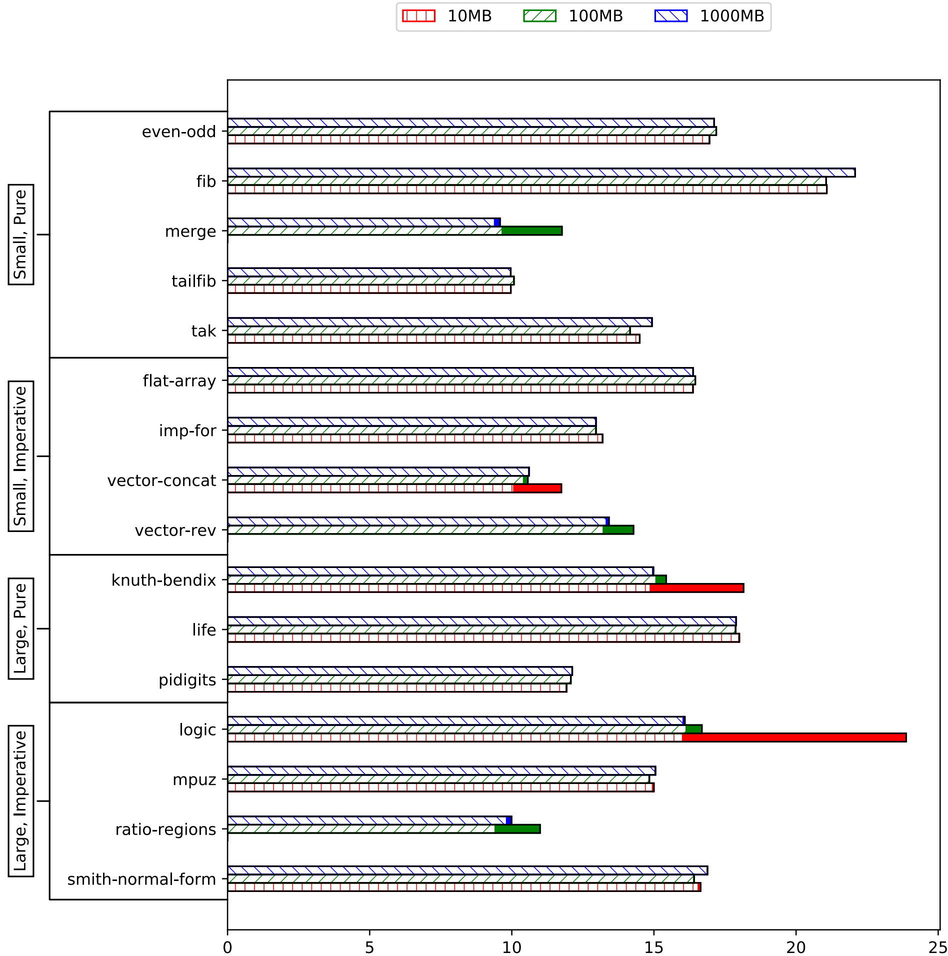


Fig. 14. Average execution time in seconds over 20 executions of the benchmarks for different heap sizes. The solidly filled part of each bar indicates time spent in the garbage collector. In the 10MB case, missing bars indicates that CakeML ran out of memory for that benchmark.

- (BO) The BVL optimisations are disabled.
- (RA) The register allocation algorithm uses a simple heuristic rather than IRC.
- (All) The default compiler with all optimisations enabled.

The benchmark results for each configuration is shown in Figure 15. Across all the benchmarks, our CLOSLANG optimisations clearly provide the most significant improvements. We note that the improvements are significant even for small benchmarks because they also help to optimise parts of the CakeML basis libraries that get used in the benchmarks.

This is perhaps unsurprising, because the optimisations performed in CLOSLANG are important optimisations for functional languages. In addition, because they happen relatively early in the compilation process, they also help improve key follow-up optimisations. For example, the register allocator may face less register pressure around call sites after multi-

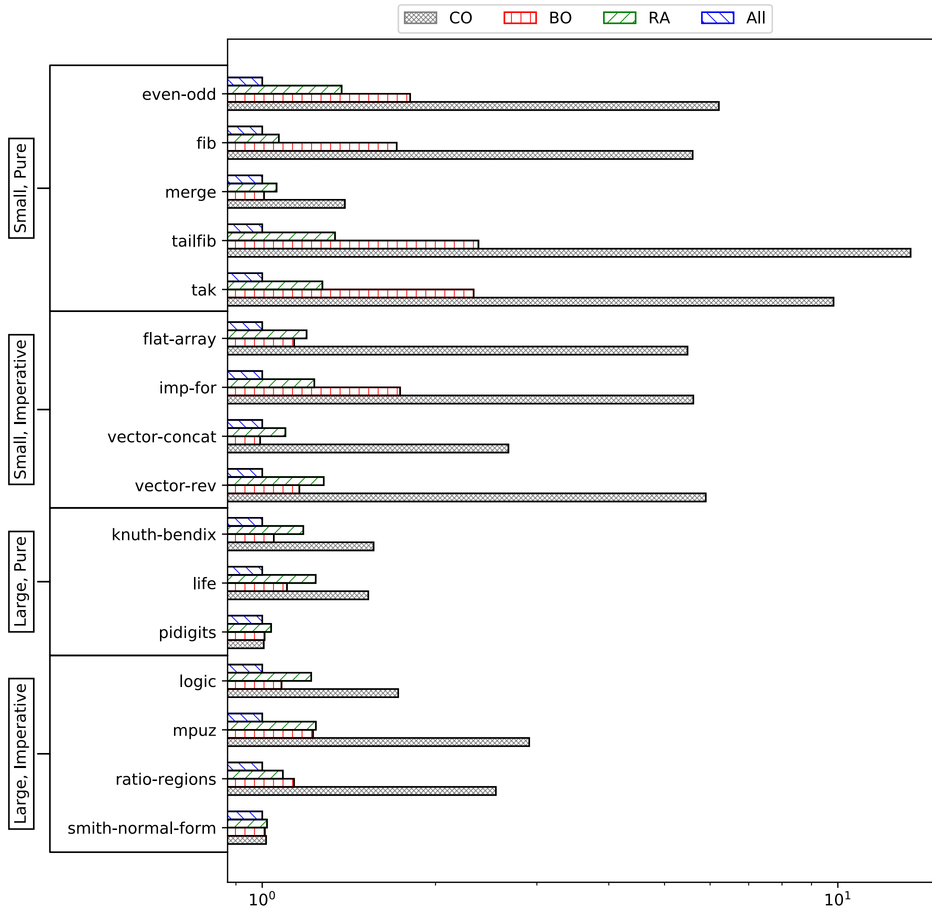


Fig. 15. Average executions over 20 executions of the benchmarks for different CakeML compiler configurations, relative to the (All) configuration.

argument functions get introduced. In Owens *et al.* (2017), we also observed that the multi-argument introduction optimisation provided the most significant improvements over the other CLOSLANG optimisations.

The BVL optimisations and register allocator provide slightly more modest improvements, but they also clearly have important roles to play. In particular, the inlining optimisations yield significant improvements for many of the small benchmarks.

Finally, we observe that the `pidigits` and `smith-normal-form` benchmarks see very little improvement from any compiler optimisations. This is, again, due to the fact that these benchmarks being dominated by bignum computations.

12.4 Register Allocation

To further validate the need for a good register allocation heuristic, we compared our implementation of the IRC algorithm (labelled IRC) against a simple colouring heuristic (labelled SIMPLE) that only makes local coalescing optimisations when it is about to

colour a node[21]. The evaluation was performed by comparing these two allocators on the clash graphs generated during the x64 bootstrap compilation process of the CakeML compiler.

At the point of writing, this process generates a total of 5110 clash graphs. Of these, we removed 2072 entries on which both allocators perfectly coalesce all available move instructions. These typically correspond to very small code table entries, which are not very interesting from a register allocation perspective. We further categorize the remaining 3038 clash graphs into those with a small number of moves ($1-100$), those with a medium number of moves ($101-1000$), and those with a large number of moves (> 1000).

These boundaries are rather arbitrarily chosen, but they serve to illustrate the different types of clash graphs that the register allocator might encounter in practice; we use the number of moves as a proxy for the complexity of the clash graphs. Indeed, it can be seen that the percentage of coalesceable moves i.e., those moves that do not already clash noticeably drops as we increase the number of moves. Nevertheless, a large percentage of the remaining moves are available for coalescing by our allocators.

The results are summarized in Table 1; for each category, we show the average % of coalesceable moves followed by the average % of moves coalesced *out of the coalesceable moves* by each register allocator. Note that these results are collected on the x64 target, so 9 registers are available for register allocation in both cases. Unsurprisingly, IRC manages to coalesce between $25-30\%$ more of the coalesceable moves than SIMPLE.

There are several possible directions for improving these results. Firstly, our characterization of suitable colouring functions is rather strict; as suggested by one of our reviewers, we have not considered the special case where two temporaries that always hold the same value can, in fact, be assigned the same register even though they might be simultaneously live. This could explain the drop in the number of coalesceable moves as we consider larger clash graphs. Nevertheless, we note that a significant portion of the moves remain coalesceable. Secondly, our register allocator is a textbook implementation of IRC, which can certainly be improved with further heuristics. As we have mentioned, this would have little impact on our correctness proofs since the allocator is treated purely as a heuristic. Lastly, the structure of our compiler currently requires registers to be reserved for spills at a later stage in the compiler. This was a necessary simplification for verification, but it would be interesting to explore the full power of the IRC algorithm, which explicitly rewrites the input program when spills are encountered.

13 Discussion of Related Work

There has been much interest in verified compilation and optimisation; CompCert, a verified optimising compiler for C, is perhaps the most well-known project. Like CompCert, our work focuses on verifying an entire compiler, rather than specific verified optimisations. In this section, we first give a comparison with the previous CakeML compiler, then we discuss related work for various parts of our new compiler.

[21] For each node SIMPLE is about to colour, it checks all the nodes that it has a move instruction with, and then tries to pick a colour that results in a move being coalesced.

Category	No. Clash Graphs	Avg. % Coalesceable	Avg. % Coal. IRC	Avg. % Coal. SIMPLE
Small (1 - 100 moves)	1214	88.9	78.2	46.9
Medium (101 - 1000 moves)	1278	84.3	72.4	44.4
Large (>1000 moves)	546	71.9	77.3	51.5

Table 1. *Performance of our IRC and SIMPLE allocators on the clash graphs generated during the CakeML compiler's bootstrap process. Note that the last two columns indicate the % of coalesced moves by the respective allocators out of the* coalesceable *moves.*

Detailed Comparison with Previous Compiler Our source language (CakeML) has been extended with an FFI, allowing for I/O within CakeML programs. We also added support for new primitive datatypes: strings, bytes, words, immutable vectors and mutable arrays. We have improved the source semantics by removing the pre-type-checking elaboration step; closure values now include the lexically scoped top-level environments (containing data constructor and top-level/module-top-level definitions).

The product of the previous compiler was a verified interactive loop (REPL) since our focus there was on end-to-end verification. We have not yet constructed a similar REPL for the new compiler. The previous compiler compiled from source to a single IL, then to stack-machine-based bytecode and finally to x86-64. The bytecode was designed so that each operation mapped to a fixed sequence of x86 instructions, and it was also designed to make verification of the GC as easy as possible. Unfortunately, the ease of verification also meant that the compiler had poor performance – we found the bytecode IL too low level for functional programming optimisations (multi-argument functions, lambda lifting, etc.) and too high level for backend optimisations. For example, it naively followed the semantics and allocated a closure on each additional argument to a function, pattern matches were not compiled efficiently (even for exhaustive, non-nested patterns), and the bytecode compiler only used registers as temporary storage within single bytecode instructions. The new version addresses all of these problems and splits each improvement into its own phase and IL in order to keep the verification of different parts as separate and as understandable as possible.

Optimisations The CompCert project has investigated a variety of verified optimisations, and some of our optimisations, e.g. compilation of parallel moves (Rideau *et al.*, 2008) is based on work done in Coq for CompCert. Coalescing register allocation was also verified for CompCert (Blazy *et al.*, 2010). However, CompCert still uses a translation validation approach for its register allocation phase (Rideau & Leroy, 2010). We have the same setup in our compiler, although we only use the translation validation approach when we need to evaluate the compiler in the logic; the main reason, like in CompCert, is for speed of evaluation. Our proof technique for the coalescing allocator also differs in that we do not prove correctness with respect to a full specification of the IRC algorithm. We are confident that our proof decoupling allows for other types of allocators, e.g. linear scan, to be verified on top of the intricate liveness analysis theorem. There has also been much interest in formally verified SSA-form middle ends: the CompCertSSA project (Barthe *et al.*, 2014) extended CompCert with a formally specified SSA form middle-end, and also investigated

formal verification of optimisations in their semantics (Demange *et al.*, 2015). Similarly, SSA-based optimisations were verified in the Vellvm project (Zhao *et al.*, 2013). Other work (Buchwald *et al.*, 2016) has focused on finding minimal SSA representations that are more efficient for these optimisations.

Garbage Collection GCMinor (McCreight *et al.*, 2010) is an intermediate language with GC primitives, that can be compiled down to CMinor with calls to a verified GC. They do not run into the same problem as we do because register allocation occurs in CompCert after CMinor. The main difference between our approaches is that they need to use an explicit shadow stack to track and modify live roots in the GC. Instead, our GC is implemented at a lower level, where it is allowed to directly inspect and modify the entire stack. This necessarily makes our proofs more complicated, as evidenced by the need for the permute oracle, but it is important, because we need to minimise (stack-related) function call overhead in a functional language such as CakeML.

Both GCMinor and our work focus on compilation for single processors, and so our GC algorithm and its related proofs work only for the non-concurrent setting. State-of-the-art, concurrent GCs have also been verified (Gammie *et al.*, 2015), although that work was not done in the context of verified compilation.

Compilers for Functional Languages The LambdaTamer project (Chlipala, 2010) focuses on proof and tactic engineering for efficient verification of compilers. The end product is a verified compiler for a functional language down to idealised assembly with register allocation, but without garbage collection.

The Cogent (O'Connor *et al.*, 2016) language has a proof-producing compiler down to C, which can be further compiled with CompCert, or via translation validation (Sewell *et al.*, 2013). It is a pure, functional and total language, aimed at reasoning for systems programming. Unlike our work, Cogent leaves the optimisation up to the C compiler and it does not need a garbage collector since their focus is on producing efficient snippets of systems code.

The verified Lisp implementation of Myreen & Davis (2011) is a precursor to the CakeML compilers and read-eval-print loop.

Compositional Compilers Compositional compilers have also received much attention recently; amongst other advantages, they allow for separate (modular) compilation and hence modular verification of large-scale programs. In this space, Compositional CompCert (Stewart *et al.*, 2015) extends CompCert to the compositional setting. More closely related to our work, Pilsner (Neis *et al.*, 2015) is a compositional compiler for an imperative, functional programming language – while our work has focused on realistic, end-to-end compilation, combining this with compositionality is certainly a task that warrants further work. More recently, Kang *et al.* (2016) describe a technique for verifying separate compilation in CompCert, in the case where only a single, verified compiler is used. Their technique has since been integrated into the CompCert development, and we believe it is applicable to our development as well.

Modelling Memory Usage High-level source semantics, such as CakeML's, typically do not have a notion of memory usage. In contrast, the amount of memory that can be accessed on the physical target machine is finite. For our compiler, we resolve this mismatch by allowing the compiled program to terminate early with an out-of-memory error.

CompCert instead uses an infinitely addressable memory in its target semantics and proves correctness against this semantics. The Peek framework (Mullen *et al.*, 2016) extends CompCert's x86 semantics with a fixed-size, 32-bit integer indexed memory. This is used to provide a target in which assembly level peephole optimizations can be easily verified. Their correctness theorem assumes that all pointers generated by CompCert fit within 32-bit integers. Going further, Quantitative CompCert (Carbonneaux *et al.*, 2014) modifies the target semantics to add an explicit notion of stack overflow. They also provide (automated) tools with which quantitative stack space bounds can be proved at the source level and refined down to the target, hence removing the possibility of stack overflow at the target.

The CerCo project (Amadio *et al.*, 2013) developed a verified C compiler that allows precise source-level proofs about the time and space consumption of the generate object code. Their method for formal reasoning about time and space consumption has also been adapted to apply to higher-order functional languages (Amadio & Régis-Gianas, 2011).

14 Conclusions

This paper has presented the structure of the latest verified compiler backend for CakeML. The design of the compiler attempts to mimic mainstream compilers, while still keeping the verification understandable and, most importantly, extensible. The entire development is approximately 100 000 lines of HOL4 proof scripts.

This latest version of the compiler backend is designed as a platform for future research, experimentation and student projects. We expect the compiler to evolve as the CakeML language evolves and as we add new features to the compiler. We believe there is plenty of room for improvement particularly in the lower part of the compiler, which is currently lacking common-subexpression elimination and peephole optimisations.

As a platform for student projects, the compiler has worked well: Masters and Bachelors-level students have thus far contributed a verified generational version of the garbage collector, (Ericsson *et al.*, 2017); a new verified optimisation that turns non-tail-recursive functions into tail-recursive functions, (Abrahamsson, 2017); and a tool for visualising the internals of the compiler, (Hjort *et al.*, 2017).

In the broader context, the new compiler backend fits into an ecosystem of tools built around the formal definition of the CakeML language and might have applications separately from CakeML. In the CakeML ecosystem, the compiler backend is a vital part in a proof-producing code extraction mechanism that allows verified binaries to be produced automatically from shallowly embedded HOL functions. We believe parts of the CakeML compiler backend could be used as components in implementations of compilers for other languages, e.g. Scheme or some cut-down version of Java.

Acknowledgements. We thank the anonymous reviewers for their helpful comments on drafts of this paper, and are grateful for good comments by Mike Gordon and Konrad Slind

on the conference version of this paper. The first author was supported by A*STAR, Singapore; the second author was partially supported by the Swedish Research Council, Sweden; the fourth author was partially supported by EPSRC Programme Grant EP/K008528/1, UK; and the fifth author was partially supported by EPSRC Grant EP/N028759/1, UK.

References

Abrahamsson, Oskar. (2017). Automatically introducing tail recursion in CakeML. *Trends in Functional Programming (TFP)*. Springer. To Appear.

Amadio, Roberto M., & Régis-Gianas, Yann. (2011). Certifying and reasoning on cost annotations of functional programs. *Pages 72–89 of:* Peña, Ricardo, van Eekelen, Marko C. J. D., & Shkaravska, Olha (eds), *Foundational and Practical Aspects of Resource Analysis - Second International Workshop, FOPARA 2011, Madrid, Spain, May 19, 2011, Revised Selected Papers*. Lecture Notes in Computer Science, vol. 7177. Springer.

Amadio, Roberto M., Ayache, Nicholas, Bobot, François, Boender, Jaap, Campbell, Brian, Garnier, Ilias, Madet, Antoine, McKinna, James, Mulligan, Dominic P., Piccolo, Mauro, Pollack, Randy, Régis-Gianas, Yann, Coen, Claudio Sacerdoti, Stark, Ian, & Tranquilli, Paolo. (2013). Certified complexity (CerCo). *Pages 1–18 of:* Lago, Ugo Dal, & Peña, Ricardo (eds), *Foundational and Practical Aspects of Resource Analysis - Third International Workshop, FOPARA 2013, Bertinoro, Italy, August 29-31, 2013, Revised Selected Papers*. Lecture Notes in Computer Science, vol. 8552. Springer.

Appel, Andrew W. (1992). *Compiling with continuations*. Cambridge University Press.

Barthe, Gilles, Demange, Delphine, & Pichardie, David. (2014). Formal verification of an SSA-based middle-end for CompCert. *ACM trans. program. lang. syst.*, **36**(1), 4:1–4:35.

Blazy, Sandrine, Robillard, Benoît, & Appel, Andrew W. (2010). Formal verification of coalescing graph-coloring register allocation. *Pages 145–164 of:* Gordon, Andrew D. (ed), *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Lecture Notes in Computer Science, vol. 6012. Springer.

Buchwald, Sebastian, Lohner, Denis, & Ullrich, Sebastian. (2016). Verified construction of static single assignment form. *Pages 67–76 of:* Zaks, Ayal, & Hermenegildo, Manuel V. (eds), *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*. ACM.

Carbonneaux, Quentin, Hoffmann, Jan, Ramananandro, Tahina, & Shao, Zhong. (2014). End-to-end verification of stack-space bounds for C programs. *Pages 270–281 of:* O'Boyle, Michael F. P., & Pingali, Keshav (eds), *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. ACM.

Chlipala, Adam. (2010). A verified compiler for an impure functional language. *Pages 93–106 of:* Hermenegildo, Manuel V., & Palsberg, Jens (eds), *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. ACM.

Demange, Delphine, Pichardie, David, & Stefanesco, Léo. (2015). Verifying fast and sparse SSA-based optimizations in Coq. *Pages 233–252 of:* Franke, Björn (ed),

Compiler Construction - 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9031. Springer.

Ericsson, Adam Sandberg, Myreen, Magnus O., & Pohjola, Johannes Åman. (2017). A verified generational garbage collector for CakeML. *Pages 444–461 of:* Ayala-Rincón, Mauricio, & Muñoz, César A. (eds), *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*. Lecture Notes in Computer Science, vol. 10499. Springer.

Fox, Anthony C. J., Myreen, Magnus O., Tan, Yong Kiam, & Kumar, Ramana. (2017). Verified compilation of CakeML to multiple machine-code targets. *Pages 125–137 of:* Bertot, Yves, & Vafeiadis, Viktor (eds), *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*. ACM.

Gammie, Peter, Hosking, Antony L., & Engelhardt, Kai. (2015). Relaxing safely: verified on-the-fly garbage collection for x86-TSO. *Pages 99–109 of:* Grove, David, & Blackburn, Steve (eds), *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. ACM.

George, Lal, & Appel, Andrew W. (1996). Iterated register coalescing. *ACM trans. program. lang. syst.*, **18**(3), 300–324.

Granlund, Torbjörn, *et al.* . (2017). *GNU MP: The GNU Multiple Precision Arithmetic Library*. 6.1.2 edn. `http://gmplib.org/`.

Guéneau, Armaël, Myreen, Magnus O., Kumar, Ramana, & Norrish, Michael. (2017). Verified characteristic formulae for CakeML. *Pages 584–610 of:* Yang, Hongseok (ed), *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Lecture Notes in Computer Science, vol. 10201. Springer.

Hjort, Rikard, Holmgren, Jakob, & Persson, Christian. (2017). The CakeML compiler explorer: Tracking intermediate representations in a verified compiler. *Trends in Functional Programming (TFP)*. Springer. To Appear.

Kang, Jeehoon, Kim, Yoonseung, Hur, Chung-Kil, Dreyer, Derek, & Vafeiadis, Viktor. (2016). Lightweight verification of separate compilation. *Pages 178–190 of:* Bodík, Rastislav, & Majumdar, Rupak (eds), *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. ACM.

Kumar, Ramana. 2016 (Feb.). *Self-compilation and self-verification*. Tech. rept. UCAM-CL-TR-879. University of Cambridge, Computer Laboratory.

Kumar, Ramana, Myreen, Magnus O., Norrish, Michael, & Owens, Scott. (2014). CakeML: a verified implementation of ML. *Pages 179–192 of:* Jagannathan, Suresh, & Sewell, Peter (eds), *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. ACM.

Leroy, Xavier. (2009). A formally verified compiler back-end. *J. autom. reasoning*, **43**(4), 363–446.

Matthews, David. (2017). *Poly/ML*. 5.7 edn. `http://www.polyml.org/`.

McCreight, Andrew, Chevalier, Tim, & Tolmach, Andrew P. (2010). A certified framework for compiling and executing garbage-collected languages. *Pages 273–284 of:* Hudak, Paul, & Weirich, Stephanie (eds), *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*. ACM.

MLton developers. (2017). *MLton*. `http://mlton.org/`.

Mullen, Eric, Zuniga, Daryl, Tatlock, Zachary, & Grossman, Dan. (2016). Verified peephole optimizations for CompCert. *Pages 448–461 of:* Krintz, Chandra, & Berger, Emery (eds), *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. ACM.

Myreen, Magnus O. (2010). Reusable verification of a copying collector. *Pages 142–156 of:* Leavens, Gary T., O'Hearn, Peter W., & Rajamani, Sriram K. (eds), *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings*. Lecture Notes in Computer Science, vol. 6217. Springer.

Myreen, Magnus O., & Curello, Gregorio. (2013). Proof pearl: A verified bignum implementation in x86-64 machine code. *Pages 66–81 of:* Gonthier, Georges, & Norrish, Michael (eds), *Certified Programs and Proofs (CPP)*. Lecture Notes in Computer Science, vol. 8307. Springer.

Myreen, Magnus O., & Davis, Jared. (2011). A verified runtime for a verified theorem prover. *Pages 265–280 of:* van Eekelen, Marko C. J. D., Geuvers, Herman, Schmaltz, Julien, & Wiedijk, Freek (eds), *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*. Lecture Notes in Computer Science, vol. 6898. Springer.

Myreen, Magnus O., & Owens, Scott. (2014). Proof-producing translation of higher-order logic into pure and stateful ML. *J. funct. program.*, **24**(2-3), 284–315.

Myreen, Magnus O., Gordon, Michael J. C., & Slind, Konrad. (2012). Decompilation into logic - improved. *Pages 78–81 of:* Cabodi, Gianpiero, & Singh, Satnam (eds), *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*. IEEE.

Neis, Georg, Hur, Chung-Kil, Kaiser, Jan-Oliver, McLaughlin, Craig, Dreyer, Derek, & Vafeiadis, Viktor. (2015). Pilsner: a compositionally verified compiler for a higher-order imperative language. *Pages 166–178 of:* Fisher, Kathleen, & Reppy, John H. (eds), *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. ACM.

O'Connor, Liam, Chen, Zilin, Rizkallah, Christine, Amani, Sidney, Lim, Japheth, Murray, Toby C., Nagashima, Yutaka, Sewell, Thomas, & Klein, Gerwin. (2016). Refinement through restraint: bringing down the cost of verification. *Pages 89–102 of:* Garrigue, Jacques, Keller, Gabriele, & Sumii, Eijiro (eds), *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. ACM.

Owens, Scott, Myreen, Magnus O., Kumar, Ramana, & Tan, Yong Kiam. (2016). Functional big-step semantics. *Pages 589–615 of:* Thiemann, Peter (ed), *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Lecture Notes in Computer Science, vol. 9632. Springer.

Owens, Scott, Norrish, Michael, Kumar, Ramana, Myreen, Magnus O., & Tan, Yong Kiam. (2017). Verifying efficient function calls in CakeML. *PACMPL*, **1**(ICFP), 18:1–18:27.

Rideau, Laurence, Serpette, Bernard P., & Leroy, Xavier. (2008). Tilting at windmills with Coq: Formal verification of a compilation algorithm for parallel moves. *J. autom. reasoning*, **40**(4), 307–326.

Rideau, Silvain, & Leroy, Xavier. (2010). Validating register allocation and spilling. *Pages 224–243 of:* Gupta, Rajiv (ed), *Compiler Construction, 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Lecture Notes in Computer Science, vol. 6011. Springer.

Romanenko, Sergei, Russo, Claudio, & Sestoft, Peter. (2013). Moscow ML owner's manual. version 2.10. 06.

Sevcík, Jaroslav, Vafeiadis, Viktor, Nardelli, Francesco Zappa, Jagannathan, Suresh, & Sewell, Peter. (2013). CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, **60**(3), 22:1–22:50.

Sewell, Thomas Arthur Leck, Myreen, Magnus O., & Klein, Gerwin. (2013). Translation validation for a verified OS kernel. *Pages 471–482 of:* Boehm, Hans-Juergen, & Flanagan, Cormac (eds), *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. ACM.

Slind, Konrad, & Norrish, Michael. (2008). A brief overview of HOL4. *Pages 28–32 of:* Mohamed, Otmane Aït, Muñoz, César A., & Tahar, Sofiène (eds), *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*. Lecture Notes in Computer Science, vol. 5170. Springer.

SML/NJ developers. (2017). *SML/NJ*. `http://www.smlnj.org/`.

Stewart, Gordon, Beringer, Lennart, Cuellar, Santiago, & Appel, Andrew W. (2015). Compositional CompCert. *Pages 275–287 of:* Rajamani, Sriram K., & Walker, David (eds), *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. ACM.

Tan, Yong Kiam, Owens, Scott, & Kumar, Ramana. (2015). A verified type system for CakeML. *Pages 7:1–7:12 of:* Lämmel, Ralf (ed), *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages, IFL '15, Koblenz, Germany, September 14-16, 2015*. ACM.

Tan, Yong Kiam, Myreen, Magnus O., Kumar, Ramana, Fox, Anthony C. J., Owens, Scott, & Norrish, Michael. (2016). A new verified compiler backend for CakeML. *Pages 60–73 of:* Garrigue, Jacques, Keller, Gabriele, & Sumii, Eijiro (eds), *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. ACM.

Yang, Xuejun, Chen, Yang, Eide, Eric, & Regehr, John. (2011). Finding and understanding bugs in C compilers. *Pages 283–294 of:* Hall, Mary W., & Padua, David A. (eds), *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. ACM.

Zhao, Jianzhou, Nagarakatte, Santosh, Martin, Milo M. K., & Zdancewic, Steve. (2013). Formal verification of SSA-based optimizations for LLVM. *Pages 175–186 of:* Boehm, Hans-Juergen, & Flanagan, Cormac (eds), *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. ACM.