# A New Verified Compiler Backend for CakeML

Yong Kiam Tan
IHPC, A*STAR, Singapore
tanyongkiam@gmail.com

Magnus O. Myreen
Chalmers University of Technology, Sweden
myreen@chalmers.se

Ramana Kumar
Data61, CSIRO / UNSW, Australia
ramana.kumar@data61.csiro.au

Anthony Fox
University of Cambridge, UK
anthony.fox@cl.cam.ac.uk

Scott Owens
University of Kent, UK
s.a.owens@kent.ac.uk

Michael Norrish
Data61, CSIRO / ANU, Australia
michael.norrish@data61.csiro.au

## Abstract

We have developed and mechanically verified a new compiler backend for CakeML. Our new compiler features a sequence of intermediate languages that allows it to incrementally compile away high-level features and enables verification at the right levels of semantic detail. In this way, it resembles mainstream (unverified) compilers for strict functional languages. The compiler supports efficient curried multi-argument functions, configurable data representations, exceptions that unwind the call stack, register allocation, and more. The compiler targets several architectures: x86-64, ARMv6, ARMv8, MIPS-64, and RISC-V.

In this paper, we present the overall structure of the compiler, including its 12 intermediate languages, and explain how everything fits together. We focus particularly on the interaction between the verification of the register allocator and the garbage collector, and memory representations. The entire development has been carried out within the HOL4 theorem prover.

***Categories and Subject Descriptors*** D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness proofs, Formal methods; F.3.1 [*Logics and meanings of programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification, Specification techniques, Invariants

***Keywords*** Compiler verification; ML; verified optimisations.

## 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; Ševčík 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 succesful 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 presents the most realistic verified compiler for a functional programming language to date.

- The new compiler has a fully featured source language, namely CakeML, which includes user-defined modules, signatures, mutually recursive functions, pattern matching, user-defined exceptions and datatypes, references, mutable arrays, immutable vectors, strings, etc.
- The compiler passes through all the usual compiler phases, including register allocation via Iterated Register Coalescing. It uses 12 intermediate languages that together allow implementation of optimisations at practically any level of abstraction.
- The compiler has efficient, configurable data representations and properly compiles the call stack into memory, including the ML-style exception mechanism.
- The compiler takes concrete syntax as input and produces concrete machine code in five real machine languages as output. It supports both 32-bit and 64-bit architectures.

None of these are new ideas in compiler implementation, and we freely take inspiration from existing compilers, including CompCert and OCaml. Our contribution here is the verification effort, especially 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 first contribution here is a design that allowed us to complete the verification while supporting good quality code (for multiple targets) and the implementation of further optimisations in the future.

Our second contribution focuses on the interaction of register allocation with copying garbage collection. A necessarily awkward ordering of compiler phases posed new technical challenges for the verification effort. The garbage collector (GC) must be introduced as a primitive when the data abstraction is removed; but this phase precedes phases (instruction selection, SSA-form introduction, and register allocation) that determine in exactly what order the GC will visit the roots. This awkward ordering of compiler phases makes it tricky to specify the semantics of intermediate languages that follow the introduction of the GC but precede register allocation.

The architecture of our compiler was designed to be extensible and we hope that it will serve the role of a verified compiler artefact for functional languages in future research.

All of our definitions and proofs are carried out in the HOL4 system. The code is available at `https://code.cakeml.org`.

## 2. Approach

In this section, we start with a brief overview of the compiler, the semantics of its intermediate languages, and the correctness proofs. Subsequent sections will expand on the details.

### 2.1 Compiler Implementation

The compiler passes through 12 intermediate languages (ILs). It 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, removal of data abstraction, 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.

The top-level compiler function takes a string and configuration as input, and produces, on a successful execution, a list of bytes and a number $m$. The bytes are machine code for the specific target architecture and $m$ is the number of foreign function interface (FFI) entry points the executable assumes it has access to. The generated machine code needs to be executed in a setting where it can jump to FFI functions for each index less than $m$.

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.[1] Encoding errors ought to be rare.

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.

[1] An encoding error happens 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.

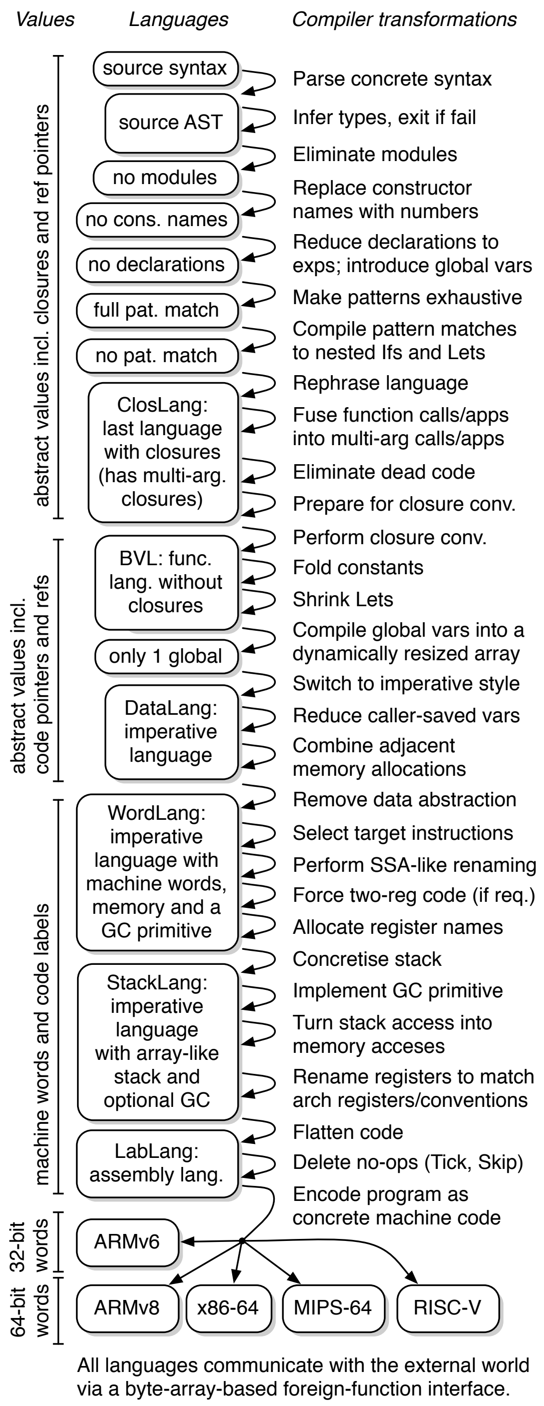

Values
Languages
Compiler transformations
abstract values incl. closures and ref pointers
source syntax
Parse concrete syntax
source AST
Infer types, exit if fail
Eliminate modules
no modules
Replace constructor names with numbers
no cons. names
Reduce declarations to exps; introduce global vars
no declarations
Make patterns exhaustive
full pat. match
Compile pattern matches to nested Ifs and Lets
no pat. match
Rephrase language
ClosLang: last language with closures (has multi-arg. closures)
Fuse function calls/apps into multi-arg calls/apps
Eliminate dead code
Prepare for closure conv.
abstract values incl. code pointers and refs
Perform closure conv.
BVL: func. lang. without closures
Fold constants
Shrink Lets
Compile global vars into a dynamically resized array
only 1 global
Switch to imperative style
DataLang: imperative language
Reduce caller-saved vars
Combine adjacent memory allocations
machine words and code labels
Remove data abstraction
WordLang: imperative language with machine words, memory and a GC primitive
Select target instructions
Perform SSA-like renaming
Force two-reg code (if req.)
Allocate register names
Concretise stack
StackLang: imperative language with array-like stack and optional GC
Implement GC primitive
Turn stack access into memory acceses
Rename registers to match arch registers/conventions
Flatten code
LabLang: assembly lang.
Delete no-ops (Tick, Skip)
Encode program as concrete machine code
32-bit words
ARMv6
64-bit words
ARMv8
x86-64
MIPS-64
RISC-V
All languages communicate with the external world via a byte-array-based foreign-function interface.

**Figure 1.** The structure of the new CakeML compiler.

We expect future extensions to the compiler to mostly be new transitions within the existing ILs, rather than adding new ILs.

### 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 observable FFI- and terminate/diverge-behaviours a program can exhibit. Below $\phi$ `ffi_state` is the type of the FFI state which models how the environment responds to a number of calls to the FFI.

$$\mathsf{semantics} : \phi\ \texttt{ffi\_state}\ \rightarrow\ \texttt{program}\ \rightarrow\ \texttt{behaviour}\ \texttt{set}$$

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 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. 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., removal of data abstraction 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:

$$\begin{array}{l}\mathsf{semantics}_{\mathrm{B}}\ \mathit{ffi}\ \mathit{new\_prog} \subseteq \\ \mathsf{extend\_with\_resource\_limit}\ (\mathsf{semantics}_{\mathrm{A}}\ \mathit{ffi}\ \mathit{prog})\end{array}$$

These high-level correctness theorems are easy to compose. In the proofs, we assume that the source IL's semantics does not Fail. At the top level we prove that a type correct program cannot Fail.

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.

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

The evaluate functions are also overloaded at each IL. They return a new state, $\mathit{new\_state}$, which includes the FFI state, and a result, $\mathit{res}$, 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.

In some proofs, extra fuel needs to be added to the clock in $\mathit{state}_1$ because the compiled code uses (a constant number of) extra ticks. Fuel is used for evaluation of the compiled code; the compiler itself is proved to always terminate. This extra fuel is usually existentially quantified along with $\mathit{new\_state}_1$ and $\mathit{res}_1$.

The evaluate theorems are proved by induction on the structure of evaluate for the source intermediate language. These proofs are said to *go in the direction of compilation* since the source semantics is in the antecedent of the implication and the target semantics is in the consequent. Such proofs follow the intuition of the compiler writer.

We only prove forward-style theorems when relating the compiler to the evaluate level of the semantics. They are sufficient[2] for proving the equivalence (or correspondence) of the observational semantics at the higher 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.

## 2.4 Removal of Abstractions

Removal of abstractions is a theme that can be used to describe most phases in our compiler. The original CakeML compiler's purpose was to get from source to target. Our new CakeML compiler attempts to provide the architecture for making this translation well, i.e., producing good code in the process. In particular, this goal requires enabling vital optimisations.

Register allocation is a transformation that we found to be one of the more complicated optimisations to support and we concentrate on it in this paper. Register allocation is tricky because it interacts in a subtle way with the copying garbage collector. Briefly speaking, the complication stems from the fact that the garbage collector is introduced before the layout of the stack has been concretised. The garbage collector depends on the stack, since the stack is where the collector looks for roots. The order in which it sees the roots has an effect on how the collector updates memory. The actual order in which the collector sees the roots is only fixed by the register allocator when it assigns names (i.e., locations) to the variables it spills onto the stack.

From a high level, the order of the roots does not affect the compiler's correctness. The challenge is how to communicate this fact through the compiler phases. The irrelevance of the order is a property that can easily be derived from the invariants within the proof about the removal of data abstraction, but the verification of the register allocator is separate (for good reason, because both are complicated proofs).

Our solution is to include a semantic mechanism, which we call a *permute oracle*, allowing us to alter the order in which roots are passed to the collector implementation. We use the permute oracle to prove that data abstraction holds for whatever order the register allocator decided to store the roots on the stack. Importantly, this semantic mechanism is local to one intermediate language. This approach is explained Section 7.

## 2.5 Multiple Targets

The compiler can produce code for several target architectures. The compiler is parameterised by a compiler configuration that carries around information about the target throughout the entire compiler. This configuration includes an instruction encoding function for an abstract syntax of general-purpose machine instructions. Since we

[2] Throughout this work, we only work with determinstic ILs; see the extensive discussion in Leroy (2009).

support both 64- and 32-bit architectures, we take care to make the data abstraction configurable to accommodate the different limits that these architectures impose.

### 2.6 Top-level Correctness Theorem

The top-level correctness theorem is 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 configuration is well-formed, that the generated program runs in a environment where the external world only modifies memory outside CakeML's memory region, and that 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 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, 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).

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.

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 with this feature.

A value in ClosLang's semantics is either a number, a word, an immutable block of values (constructor or vector), a pointer to an array, or a closure. 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, an evaluation environment (values 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.[3]

$$
\begin{aligned}
\mathsf{v} \;=\;& \mathsf{Number}\ \mathtt{int} \mid \mathsf{Word64}\ (64\ \mathtt{word}) \mid \mathsf{RefPtr}\ \mathtt{num} \\
\mid\;& \mathsf{Block}\ \mathtt{num}\ (\mathtt{v}\ \mathtt{list}) \\
\mid\;& \mathsf{Recclosure}\ (\mathtt{num}\ \mathtt{option})\ (\mathtt{v}\ \mathtt{list})\ (\mathtt{v}\ \mathtt{list}) \ldots \\
\mid\;& \mathsf{Closure}\ (\mathtt{num}\ \mathtt{option})\ (\mathtt{v}\ \mathtt{list})\ (\mathtt{v}\ \mathtt{list})\ \mathtt{num}\ \mathtt{exp}
\end{aligned}
$$

Having 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,

$$
\begin{aligned}
&\mathsf{v\_rel}\ (\mathsf{Closure}\ loc_1\ env_1\ args_1\ arg\_count_1\ exp_1) \\
&\quad(\mathsf{Closure}\ loc_2\ env_2\ args_2\ arg\_count_2\ exp_2)
\end{aligned}
$$

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), but will not go into details about this logical relation in this paper.

## 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.

$$
\begin{aligned}
\mathsf{v} \;=\;& \mathsf{Number}\ \mathtt{int} \mid \mathsf{Word64}\ (64\ \mathtt{word}) \mid \mathsf{CodePtr}\ \mathtt{num} \\
\mid\;& \mathsf{RefPtr}\ \mathtt{num} \mid \mathsf{Block}\ \mathtt{num}\ (\mathtt{v}\ \mathtt{list})
\end{aligned}
$$

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. Its 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.

$$
\begin{aligned}
\mathtt{exp} \;=\;& \mathsf{Var}\ \mathtt{num} \mid \mathsf{Raise}\ \mathtt{exp} \mid \mathsf{Tick}\ \mathtt{exp} \\
\mid\;& \mathsf{Let}\ (\mathtt{exp}\ \mathtt{list})\ \mathtt{exp} \mid \mathsf{Handle}\ \mathtt{exp}\ \mathtt{exp} \\
\mid\;& \mathsf{Op}\ \mathtt{op}\ (\mathtt{exp}\ \mathtt{list}) \mid \mathsf{If}\ \mathtt{exp}\ \mathtt{exp}\ \mathtt{exp} \\
\mid\;& \mathsf{Call}\ \mathtt{num}\ (\mathtt{num}\ \mathtt{option})\ (\mathtt{exp}\ \mathtt{list})
\end{aligned}
$$

Figure 2 shows an extract of BVL's functional big-step semantics, i.e., functions in HOL that define BVL's big-step semantics.

[3] Through the paper, we use HOL4's type definition syntax: each constructor name is followed by the types of its arguments (Haskell-style), but type constructors use postfix application (ML-style).

```
evaluate ([],$env$,$s$) = (Rval [],$s$)
evaluate ($x$::$y$::$xs$,$env$,$s$) =
  case evaluate ([$x$],$env$,$s$) of
    (Rval $v_1$,$s_1$) ⇒
      (case evaluate ($y$::$xs$,$env$,$s_1$) of
         (Rval $vs$,$s_2$) ⇒ (Rval ($v_1$ @ $vs$),$s_2$)
       | (Rerr $e$,$s_2$) ⇒ (Rerr $e$,$s_2$))
  | (Rerr $e$,$s_1$) ⇒ (Rerr $e$,$s_1$)
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 $vs$,$s_1$) ⇒ evaluate ([$x$],$vs$ @ $env$,$s_1$)
  | (Rerr $e$,$s_1$) ⇒ (Rerr $e$,$s_1$)
evaluate ([Op $op$ $xs$],$env$,$s$) =
  case evaluate ($xs$,$env$,$s$) of
    (Rval $vs$,$s_1$) ⇒
      (case do_app $op$ (rev $vs$) $s_1$ of
         Rval ($v$,$s_2$) ⇒ (Rval [$v$],$s_2$)
       | Rerr $err$ ⇒ (Rerr $err$,$s_1$))
  | (Rerr $e$,$s_1$) ⇒ (Rerr $e$,$s_1$)
evaluate ([Raise $x$],$env$,$s$) =
  case evaluate ([$x$],$env$,$s$) of
    (Rval $vs$,$s_1$) ⇒ (Rerr (Rraise (hd $vs$)),$s_1$)
  | (Rerr $e$,$s_1$) ⇒ (Rerr $e$,$s_1$)
evaluate ([Handle $x_1$ $x_2$],$env$,$s$) =
  case evaluate ([$x_1$],$env$,$s$) of
    (Rval $v$,$s_1$) ⇒ (Rval $v$,$s_1$)
  | (Rerr (Rraise $v$),$s_1$) ⇒ evaluate ([$x_2$],$v$::$env$,$s_1$)
  | (Rerr (Rabort $e$),$s_1$) ⇒ (Rerr (Rabort $e$),$s_1$)
evaluate ([Call $ticks$ $dest$ $xs$],$env$,$s$) =
  case evaluate ($xs$,$env$,$s$) of
    (Rval $vs$,$s_1$) ⇒
      (case find_code $dest$ $vs$ $s_1$.code of
         None ⇒ (Rerr (Rabort Rtype_error),$s_1$)
       | Some ($args$,$exp$) ⇒
           if $s_1$.clock < $ticks$ + 1 then
             (Rerr (Rabort Rtimeout_error),$s_1$ with clock := 0)
           else evaluate ([$exp$],$args$,dec_clock ($ticks$ + 1) $s_1$))
  | (Rerr $e$,$s_1$) ⇒ (Rerr $e$,$s_1$)
  ...

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

**Figure 2.** Extracts of BVL's semantics.

The evaluate function takes a list of BVL expressions exp and returns a list of values v corresponding to the expressions.

The exception mechanism shapes the look 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 results in an Rerr (Rabort Rtimeout_error); and hitting an error in the program results in Rerr (Rabort Rtype_error). Most expressions pass on exceptions that occur inside subexpressions, 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 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.

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. 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 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 state as opposed to in an environment. Subsequent languages, WordLang and StackLang, mimic DataLang in style and structure.

In DataLang's abstract syntax below, 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 always rewinds the stack to a well-defined stack frame. The finite sets of numbers (of type num_set) are cut-sets that keep track of the local variables which must be preserved past subroutine calls. Besides Call, MakeSpace 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 calls to the GC or to library code.

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

DataLang's semantics uses the same value type as BVL and operates over a state that is similar to BVL's. The most significant differences in the state are: DataLang has a stack; raising an exception affects the state of the stack, it rewinds the stack; and there is a notion of available space as described below.

The compiler performs a few optimisations in DataLang. In particular, the compiler combines memory allocations (MakeSpace) in straight-line code. The semantics of MakeSpace $n$ *names* is to guarantee that there are at least $n$ units of space available, while operations such as Cons consume space equal to one plus the length of the Block that is produced. Some operations (e.g. bignum addition) consume a statically unknown amount of space, which resets the available space to zero. In DataLang, this space measure is an abstract notion since there is no memory.

## 6. Removal of Data Abstraction

DataLang sets the stage for the removal of data abstraction. DataLang is compiled into a language called WordLang, which has an abstract syntax that at a glance looks very 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 $\alpha$ word_loc, which are machine words and labels. The type variable $\alpha$ indicates the length of the machine word. Our proofs assume that $\alpha$ 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 $n_1$.

```
α word_loc = Word (α word) | Loc num num
```

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, a global variable store, a stack, a word-addressed memory and the memory domain. It also contains a code table (looked up by function calls), a state for the FFI, a clock (used in functional big-step semantics), a handler pointer, and a flag controlling big-endianness. The last two components, namely the GC primitive and the permute oracle will be explained in Section 6.1 and Section 7.1 respectively.

```
(α, φ) state =
<| locals : (α word_loc num_map);
   store : (store_name ↦ α word_loc);
   stack : (α stack_frame list);
   memory : (α word → α word_loc);
   mdomain : (α word set);
   code : ((num × α prog) num_map);
   ffi : (φ ffi_state);
   clock : num;
   handler : num;
   be : bool;
   gc_fun : (α gc_fun_type);
   permute : (num → num → num) |>
```

The stack is a list of stack frames of the type shown below. The association list (alist) is an ordered mapping from variable names to values stored in the stack frame. The triple of numbers holds information about an optional exception-handling frame. The handler pointer always points to the latest handler frame in the stack, if there is one.

```
α stack_frame =
  StackFrame ((num, α word_loc) alist)
       ((num × num × num) option)
```

Importantly, 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

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 3. 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 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? We could have used the specific function from the Word-to-Stack compiler instead of the semantic function in the state. However, such a solution would have blurred the line between compiler implementation and semantics. In particular the compiler configuration, which determines the value representation used by the GC, would need to be a part of the semantics of WordLang.

```
gc s =
 let roots = enc_stack s.stack
 in
  case s.gc_fun (roots,s.memory,s.mdomain,s.store) of
   None ⇒ None
  | Some (new_roots,new_m,new_st) ⇒
     case dec_stack new_roots s.stack of
      None ⇒ None
     | Some new_stack ⇒
        Some
         (s with
          <|stack := new_stack; store := new_st;
           memory := new_m|>)

enc_stack [] = []
enc_stack (StackFrame l hndlr::st) = map snd l @ enc_stack st
```

**Figure 3.** The semantics of invoking WordLang's GC.

### 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 is made complicated by the big leap in abstraction level, the fact that WordLang can run a garbage collector, 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. 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-endianess (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 $\alpha$ `word_loc`).

$$\text{memory\_rel}\ \mathit{config}\ s.\text{refs}\ s.\text{space}\ t.\text{store}\ t.\text{memory}\ t.\text{mdomain}\ t.\text{be}\ \mathit{value\_pairs}$$

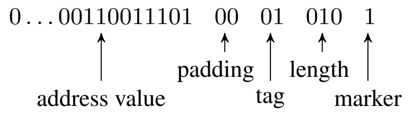
Here $s$.space is a lower bound: memory_rel is true for space $n$ if there is at least space for $n$ $\alpha$ `word_loc` values in the heap.

The data representation is kept configurable since it is hard to find one solution that works for all situations, in particular our support for both 64- and 32-bit architectures forces us to be flexible with the details of the data representation.

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, i.e. the collector leaves them untouched.

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.

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 that 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. list-cons is likely to be represented as two words in memory as opposed to the current three.

## 7. Register Allocation

Once data abstraction has been removed, the compiler runs instruction selection, an SSA-like variable transformation, and register allocation. We describe these transformations next.

In our context—a functional language with a copying garbage collector—verifying register allocation is more complicated than usual. 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.

In what follows, we explain how we have used 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 between natural numbers (i.e. WordLang variable names).

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[4]. 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 names 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, and the resulting list is added to the WordLang stack as a new stack frame. The semantics uses the following functions to push a stack frame onto the stack. Here the option indicates whether an exception handler is to be pushed, and the list_rearrange function permutes a list according to a given function.

```
env_to_list $env$ $bij\_seq$ =
  let $mover$ = $bij\_seq$ 0;
      $permute$ = $(\lambda n.\ bij\_seq\ (n + 1))$;
      $l$ = toAList $env$;
      $l$ = sort key_val_compare $l$;
      $l$ = list_rearrange $mover$ $l$
  in
    $(l, permute)$

push_env $env$ None $s$ =
  let $(l,p)$ = env_to_list $env$ $s$.permute
  in
    $s$ with
    <|stack := StackFrame $l$ None::$s$.stack; permute := $p$|>
push_env $env$ (Some $(w,h,l_1,l_2)$) $s$ =
  let $(l,p)$ = env_to_list $env$ $s$.permute;
      $h$ = Some ($s$.handler,$l_1$,$l_2$)
  in
    $s$ with
    <|stack := StackFrame $l$ $h$::$s$.stack; permute := $p$;
      handler := len $s$.stack|>
```

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*.

```
⊢ evaluate$_{\mathrm{D}}$ $(prog,s) = (res,s_1)$ ∧
  $res$ ≠ Some (Rerr (Rabort Rtype_error)) ∧
  state_rel $c$ $l_1$ $l_2$ $s$ $t$ [] $locs$ ⇒
   let $(res_1,t_1)$ =
         evaluate$_{\mathrm{W}}$
          (compile$_{\mathrm{DTW}}$ $c$ $n$ $l$ $prog$,$t$ with permute := $perm$)
   in
     ...
```

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.

```
⊢ evaluate$_{\mathrm{W}}$ $(prog,s) = (res,s_1)$ ∧ $res$ ≠ Some Error ∧
  state_rel $k$ $f$ $f'$ $s$ $t$ $lens$ ∧ $s$.permute = K I ∧ ... ⇒
   ∃ $ck$.
    let $(res_1,t_1)$ =
         evaluate$_{\mathrm{S}}$
          (compile$_{\mathrm{WTS}}$ $prog$ $bs$ $(k,f,f')$,
           $t$ with clock := $t$.clock + $ck$)
    in
     ...
```

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.

```
⊢ ... ⇒
   ∃ $perm'$.
    let $(res,rst)$ =
          evaluate$_{\mathrm{W}}$ ($prog$,$st$ with permute := $perm'$);
        $(res',rcst)$ =
          evaluate$_{\mathrm{W}}$
           (compile$_{\mathrm{WTW}}$ $t$ $k$ $a$ $c$ $((name,n,prog),col)$,
            $st$ with permute := $perm$)
    in
     ...
```

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. 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 latter point is as follows: for the oracle we picked in Word-to-Stack compilation, the Word-to-Word correctness theorem gives us some oracle such that the Word-to-Word transformations preserve program semantics. Since the Data-to-Word compilation works for any oracle, this choice of oracle can be used to instantiate its correctness theorem when we compose all of these 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.

## 7.2 Register Allocation, SSA Form and Instruction Selection

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, 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

[4] We treat calls to the GC similarly so that it only needs to look at the stack for the root set.

than the number of registers refer to the $n^{\text{th}}$ 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.

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*.

There are two simplifications to our register allocator: (1) the control flow graph of its input programs always forms a directed acyclic graph[5], and (2) we assume that two registers are kept for loading/saving from the stack.

Because of the first simplification, liveness analysis can be performed with a simple bottom-up traversal of the WordLang program instead of a more complicated fixed-point iteration. The first step in our verification is the characterisation of suitable colouring functions: given a function, $f$, the abstract liveness analysis phase checks that $f$ is injective over all live/clash sets of the program.

This abstract characterisation gels well with the required semantics preservation, i.e., correctness theorem involving the oracle. The crucial argument is as follows: we need to show that whenever we create a stack frame during the evaluation of the coloured program, there exists a permutation such that the values of the corresponding stack frame in the original program gets ordered in the same way. Injectivity allows us to construct this permutation directly, because it implies the existence of a bijection between the variable names in the two stack frames.

The other ingredient is a state invariant that holds across a forward simulation of coloured and original programs in their respective states. Here, we assume that every live starting variable in the original state corresponds to a variable in the coloured state under the colouring function and we prove that this continues to hold for the output local variables and outgoing live set if the colouring function used is suitable. These kinds of generalised inductive invariants are fairly standard in compiler proofs, and we mostly omit them for the rest of this paper.

Next, we 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. Correspondingly, we define and verify a colouring function checker that checks the aforementioned injectivity property over this tree. Crucially, this intermediary checker is designed to evaluate efficiently in the logic and it will be used later in Section 11.

Finally, we verify a graph colouring register allocator that produces the actual colouring function. One common property of many graph colouring algorithms is that they can be viewed as heuristics for choosing an appropriate order in which to pick colours for vertices of the graph. For correctness, we only need to verify the colour picking function, i.e., show that it always gives any two connected vertices distinct colours (and also that it generates the syntactic properties we need). Using this technique, we verified both a simple allocator and an Iterated Register Coalescing-based allocator (George and Appel 1996); since the latter 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.

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 $\phi$-functions. Instead, we implicitly perform $\phi$-elimination (replacing $\phi$-functions with variable movement) directly inside the SSA pass. Since this transformation renames variables (like register allocation), we again have to provide oracle permutations. The insight here is, similarly, to show that the SSA mapping defines an injective function.The reasoning about the interaction between stack frames and the oracle is similar to that used for the liveness analysis proofs.

*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 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. This is the usual form of a forward simulation-style proof, and it can be composed with our permute oracle-style theorems as well.

## 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

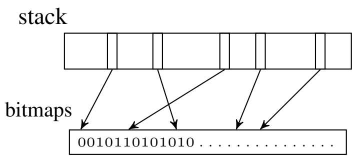
The translation from WordLang into StackLang compiles the abstract stack of WordLang into an array-like stack. Here, we implement the naming conventions used by the register allocator: WordLang names corresponding to stack variables are compiled into element lookups in stack frames, and those corresponding to registers are compiled into registers. In addition, we compile the parallel moves generated within WordLang down to single simple move instructions in StackLang.[6]

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 into the top entry of their stack frames. This is used to index into a bitmap table to obtain a bitmap that corresponds to the live positions in each stack frame. When the GC is called, it looks

[5] Control flow graphs are directed acyclic graphs in WordLang because all loops in CakeML are written using recursion and control-flow within functions can only flow forward. 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.

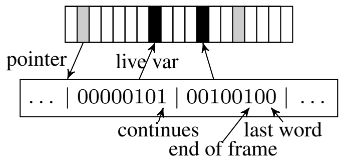
[6] Our implementation and proof of the parallel moves compilation step is a HOL formalisation of Rideau et al. (2008).

up and decodes the retrieved bitmap, and then uses it to consider only the variables that are live in each stack frame.[7]



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. 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 previously most 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.

[7] 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 faster.

```
α inst = Skip | Arith (α arith)
  | Const num (α word) | Mem mem_op num (α addr)
mem_op = Load | Load8 | Load32 | Store | Store8
  | Store32
α addr = Addr num (α word)
α arith = Binop binop num num (α reg_imm)
  | Shift shift num num num
  | AddCarry num num num num
shift = LogicalLeftShift | LogicalRightShift
  | SignedRightShift
cmp = Equal | Lower | Less | Test | NotEqual
  | NotLower | NotLess | NotTest
binop = Add | Sub | And | Or | Xor
α reg_imm = Reg num | Imm (α word)
```

**Figure 4.** The instruction datatype.

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.

### 8.3 Moving the Stack and Bitmaps into Memory

The next transformation moves StackLang's stack, bitmaps and global variable store into memory. 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.

### 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 4.

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.

### 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 the following abstract syntax. A LabLang program consists of a list of sections ($\alpha$ `sec`). Each section has a name and contains a list of assembly lines ($\alpha$ `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`).

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

### 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 a side effect of using a functional big-step semantics. 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 using an encoding function from the target configuration; it then performs a loop which computes the location of all labels and re-encodes all jumps and other label-dependent instructions. This loop is run until the lengths of all jump instructions is unchanged. The 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 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 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

```
⊢ config_ok cc mc ⇒
  case compile cc prelude input of
    Success (bytes,ffi_limit) ⇒
      ∃ behaviours.
        cakeml_semantics ffi prelude input =
          Execute behaviours ∧
        ∀ ms.
          code_installed (bytes,cc,ffi,ffi_limit,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
```

**Figure 5.** Top-level compiler correctness theorem.

to alter the state midway through this execution.

```
α asm = Inst (α inst) | Jump (α word)
  | Call (α word) | JumpReg num
  | Loc num (α word)
  | JumpCmp cmp num (α reg_imm) (α word)
```

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

The top-level correctness theorem relates the source semantics, the 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 typeable, and the observable semantics, semantics, of executing a CakeML program.

```
cakeml_semantics ffi prelude input =
  case parse (lex input) of
    None ⇒ CannotParse
  | Some prog ⇒
      if can_type_prog (prelude @ prog) then
        Execute (semantics ffi (prelude @ prog))
      else IllTyped
```

We define semantics in the style of Owens et al. (2016) as a function that returns a set of behaviours. 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. As mentioned earlier, an FFI state is an oracle that specifies how the environment will respond to calls to the FFI.

```
behaviour = Fail | Diverge (io_event stream)
  | Terminate outcome (io_event list)
```

The top-level correctness theorem is shown in Figure 5. Here $ms$ is the machine state, $mc$ is the machine configuration and extend_with_resource_limit adjusts the behaviours set to allow early exit on the outcome which signals a *resource-limit-hit*.

## 11. Evaluation of the Compiler in the Logic

One of the important properties of the first CakeML compiler is the ability to bootstrap itself in the logic. Bootstrapping the compiler in the logic has become harder to achieve in reasonable time for the new version because we have more transformations in the compiler, and some of these transformations scale poorly when evaluated in the logic. Register allocation is the most significant scalability bottleneck — even though it is fully verified, evaluating it in the logic on the large clash graphs of the compiler is infeasible.

In order to make evaluation in the logic feasible again, we opted for a translation validation approach for the register allocator that produces HOL theorems comparable to the ones produced by a direct evaluation. The translation validation produces theorems of the following form, which fits the top-level correctness theorem.

```
⊢ compile cc prelude input =
    Success (concrete_machine_code,number)
```

The translation validation approach is logically set up to avoid an in-logic execution of the register allocator function. The logical setup is simple: we store a list of colouring functions into the compiler configuration and make the register allocator check whether the next colouring it finds is a valid colouring for the current program fragment; if it is, then it uses the colouring, otherwise it runs the verified allocator. We run an SML version of the verified allocator to initialise the list of colouring functions.

Another bottleneck is the evaluation of the instruction encoder in the assembler. Here, a speed up was achieved by memoisation and use of specialised evaluation theorems. At the time of writing, the assembler loop's final exit condition is the most significant performance bottleneck. We believe it can be significantly improved, both by proving that some of the checks are always going to be true, and rephrasing the computation of the remaining checks.

## 12. 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.

***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 fixes all of these problems and further 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 slew 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 and 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 (Ullrich and Lohner 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 implemenation of Myreen and 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.

***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. 2014) 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 and Régis-Gianas 2011).

## 13. Conclusions

This paper has presented the structure of a new 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 new CakeML compiler is designed as a platform for future research and experimentation. For example, we believe many parts of the compiler are suitable for extensions: CLOSLANG is a language suitable for projects on optimisation within functional languages; BVL is a simple abstract language were many transformations are simple to verify; and WORDLANG can easily be a platform for the implementation of lower-level optimisations.

The entire CakeML compiler can be connected up to program synthesis tools or other verified CakeML applications, while parts of the new compiler backend can be used for other language implementations. For example, it would probably be a simple student project to construct a passable compiler for a first-order Lisp by starting from BVL and adjusting the configurable data representations to suit Lisp.

## Acknowledgments

We thank Mike Gordon, Konrad Slind and the anonymous reviewers for their helpful comments on drafts of this paper. 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/K040561/1, UK.

## References

R. M. Amadio and Y. Régis-Gianas. Certifying and reasoning on cost annotations of functional programs. In R. Peña, M. C. J. D. van Eekelen, and O. Shkaravska, editors, *Foundational and Practical Aspects of Resource Analysis (FOPARA), Revised Selected Papers*, volume 7177 of *LNCS*. Springer, 2011. doi:10.1007/978-3-642-32495-6_5.

R. M. Amadio, N. Ayache, F. Bobot, J. P. Boender, B. Campbell, I. Garnier, A. Madet, J. McKinna, D. P. Mulligan, M. Piccolo, R. Pollack, Y. Régis-Gianas, C. Sacerdoti Coen, I. Stark, and P. Tranquilli. Certified complexity (CerCo). In U. Dal Lago and R. Peña, editors, *Foundational and*

*Practical Aspects of Resource Analysis (FOPARA), Revised Selected Papers*. Springer, 2014.

G. Barthe, D. Demange, and D. Pichardie. Formal verification of an SSA-based middle-end for CompCert. *ACM Trans. Program. Lang. Syst.*, 36 (1), Mar. 2014. doi:10.1145/2579080.

S. Blazy, B. Robillard, and A. W. Appel. Formal verification of coalescing graph-coloring register allocation. In A. D. Gordon, editor, *European Symposium on Programming (ESOP)*. Springer, 2010. doi:10.1007/978-3-642-11957-6_9.

Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-end verification of stack-space bounds for C programs. *SIGPLAN Not.*, 49 (6), June 2014. doi:10.1145/2666356.2594301.

A. Chlipala. A verified compiler for an impure functional language. In M. V. Hermenegildo and J. Palsberg, editors, *Principles of Programming Languages (POPL)*. ACM, Jan. 2010. doi:10.1145/1707801.1706312.

D. Demange, D. Pichardie, and L. Stefanesco. Verifying fast and sparse SSA-based optimizations in Coq. In B. Franke, editor, *Compiler Construction (CC)*. Springer, 2015. doi:10.1007/978-3-662-46663-6_12.

P. Gammie, A. L. Hosking, and K. Engelhardt. Relaxing safely: verified on-the-fly garbage collection for x86-TSO. In D. Grove and S. Blackburn, editors, *Programming Language Design and Implementation (PLDI)*. ACM, 2015. doi:10.1145/2813885.2738006.

L. George and A. W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3), May 1996. doi:10.1145/229542.229546.

R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. In S. Jagannathan and P. Sewell, editors, *Principles of Programming Languages (POPL)*, 2014. doi:10.1145/2535838.2535841.

X. Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43 (4), 2009. doi:10.1007/s10817-009-9155-4.

A. McCreight, T. Chevalier, and A. Tolmach. A certified framework for compiling and executing garbage-collected languages. In *International Conference on Functional Programming (ICFP)*. ACM, 2010. doi:10.1145/1863543.1863584.

E. Mullen, D. Zuniga, Z. Tatlock, and D. Grossman. Verified peephole optimizations for CompCert. In C. Krintz and E. Berger, editors, *Programming Language Design and Implementation (PLDI)*. ACM, 2016.

M. O. Myreen. Reusable verification of a copying collector. In G. T. Leavens, P. W. O'Hearn, and S. K. Rajamani, editors, *Verified Software: Theories, Tools, Experiments (VSTTE)*. Springer, 2010. doi:10.1007/978-3-642-15057-9_10.

M. O. Myreen and J. Davis. A verified runtime for a verified theorem prover. In M. C. J. D. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *Interactive Theorem Proving (ITP)*, 2011.

G. Neis, C. Hur, J. Kaiser, C. McLaughlin, D. Dreyer, and V. Vafeiadis. Pilsner: a compositionally verified compiler for a higher-order imperative language. In K. Fisher and J. H. Reppy, editors, *International Conference on Functional Programming (ICFP)*, 2015. doi:10.1145/2784731.2784764.

L. O'Connor, C. Rizkallah, Z. Chen, S. Amani, J. Lim, Y. Nagashima, T. Sewell, A. Hixon, G. Keller, T. C. Murray, and G. Klein. COGENT: certified compilation for a functional systems language. *CoRR*, abs/1601.05520, 2016.

S. Owens, M. O. Myreen, R. Kumar, and Y. K. Tan. Functional big-step semantics. In P. Thiemann, editor, *European Symposium on Programming (ESOP)*, LNCS. Springer, 2016.

L. Rideau, B. P. Serpette, and X. Leroy. Tilting at windmills with Coq: Formal verification of a compilation algorithm for parallel moves. *J. Autom. Reason.*, 40(4), May 2008. doi:10.1007/s10817-007-9096-8.

S. Rideau and X. Leroy. Validating Register Allocation and Spilling. In R. Gupta, editor, *Compiler Construction*, volume 6011 of *LNCS*. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-11970-5_13.

J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3), 2013. doi:10.1145/2487241.2487248.

T. A. L. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified OS kernel. In *Programming Language Design and Implementation (PLDI)*. ACM, 2013. doi:10.1145/2491956.2462183.

G. Stewart, L. Beringer, S. Cuellar, and A. W. Appel. Compositional CompCert. In S. K. Rajamani and D. Walker, editors, *Principles of Programming Languages (POPL)*. ACM, 2015. doi:10.1145/2676726.2676985.

Y. K. Tan, S. Owens, and R. Kumar. A verified type system for CakeML. In *Implementation and Application of Functional Programming Languages (IFL)*. ACM Press, 2015. doi:10.1145/2897336.2897344.

S. Ullrich and D. Lohner. Verified construction of static single assignment form. *Archive of Formal Proofs*, Feb. 2016. `http://afp.sf.net/entries/Formal_SSA.shtml`, Formal proof development.

X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In M. W. Hall and D. A. Padua, editors, *Programming Language Design and Implementation (PLDI)*, 2011. doi:10.1145/1993498.1993532.

J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formal verification of SSA-based optimizations for LLVM. In H. Boehm and C. Flanagan, editors, *Programming Language Design and Implementation (PLDI)*. ACM, 2013. doi:10.1145/2491956.2462164.