Do you have space for dessert? a verified space cost semantics for CakeML programs

Citation for the original published paper (version of record):

Gómez Londoño, A., Åman Pohjala, J., Syeda, H. et al (2020)
Do you have space for dessert? a verified space cost semantics for CakeML programs
Proceedings of the ACM on Programming Languages, 4(OOPSLA)
http://dx.doi.org/10.1145/3428272

(article starts on next page)

Do You Have Space for Dessert? A Verified Space Cost Semantics for CakeML Programs

ALEJANDRO GÓMEZ-LONDOÑO, Chalmers University of Technology, Sweden
JOHANNES ÅMAN POHJOLA, CSIRO's Data61, Australia and University of New South Wales, Australia
HIRA TAQDEES SYEDA, Chalmers University of Technology, Sweden
MAGNUS O. MYREEN, Chalmers University of Technology, Sweden
YONG KIAM TAN, Carnegie Mellon University, USA

Garbage collectors relieve the programmer from manual memory management, but lead to compiler-generated machine code that can behave differently (e.g. out-of-memory errors) from the source code. To ensure that the generated code behaves exactly like the source code, programmers need a way to answer questions of the form: what is a sufficient amount of memory for my program to never reach an out-of-memory error?

This paper develops a cost semantics that can answer such questions for CakeML programs. The work described in this paper is the first to be able to answer such questions with proofs in the context of a language that depends on garbage collection. We demonstrate that positive answers can be used to transfer liveness results proved for the source code to liveness guarantees about the generated machine code. Without guarantees about space usage, only safety results can be transferred from source to machine code.

Our cost semantics is phrased in terms of an abstract intermediate language of the CakeML compiler, but results proved at that level map directly to the space cost of the compiler-generated machine code. All of the work described in this paper has been developed in the HOL4 theorem prover.

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

Additional Key Words and Phrases: compiler verification, cost semantics, space usage, garbage collection

ACM Reference Format:
Alejandro Gómez-Londoño, Johannes Åman Pohjola, Hira Taqdees Syeda, Magnus O. Myreen, and Yong Kiam Tan. 2020. Do You Have Space for Dessert? A Verified Space Cost Semantics for CakeML Programs. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 204 (November 2020), 29 pages. https://doi.org/10.1145/3428272

1 INTRODUCTION

High-level programming languages with runtimes that include a garbage collector (GC) provide a layer of abstraction that makes memory seem unbounded. This liberates the programmer from tedious and error-prone manual memory management but it leads to compiler-generated machine code that exhibits a form of *partiality*: the machine code will behave as the source semantics dictates, unless (or until) memory is exhausted.

Authors' addresses: Alejandro Gómez-Londoño, Chalmers University of Technology, Gothenburg, Sweden, alejandro.gomez@chalmers.se; Johannes Åman Pohjola, CSIRO's Data61, Sydney, Australia, University of New South Wales, Sydney, Australia, Johannes.Amanpohjola@data61.csiro.au; Hira Taqdees Syeda, Chalmers University of Technology, Gothenburg, Sweden, hira@chalmers.se; Magnus O. Myreen, Chalmers University of Technology, Gothenburg, Sweden, myreen@chalmers.se; Yong Kiam Tan, Carnegie Mellon University, Pittsburgh, USA, yongkiat@cs.cmu.edu.

2475-1421/2020/11-ART204
https://doi.org/10.1145/3428272

Well-written source-level programs stay clear of this partiality by making sure that the live data used by the program stays within some reasonable bound. For such programs, the GC can always reclaim enough memory to provide space for new allocations, even if there are an unbounded number of allocations during program execution.

For certain applications, programmers are keen to make sure that they stay clear of this partiality. In such circumstances, one has to find a way to answer the question: what is a sufficient amount of memory for my machine code executable to never reach an out-of-memory error? The answer clearly depends on the exact compilation strategy used. In this paper, we provide *a proof-based approach* for answering such questions in the context of the CakeML compiler.

The CakeML compiler [Tan et al. 2019] is a formally verified compiler for a high-level source language that has no bounds on memory and no bounds on integer size. However, the CakeML compiler targets real machine languages (x86-64, ARMv8, RISC-V, etc.) where memory and integers have hard bounds. The CakeML compiler inserts a verified GC and bignum library into the code that it produces in order to make it seem as if memory and integers are unbounded. However, these libraries can not always stop the machine code from hitting a hard resource bound—the machine code might, as a result, have to terminate with an out-of-memory error.

The partiality mentioned above is clearly visible in the top-level compiler correctness theorem for the CakeML compiler. This correctness theorem relates the set of behaviours allowed by the source semantics source_sem and the machine semantics machine_sem along the following lines:

$$\begin{array}{l}\text{machine\_sem}\ \mathit{ffi}\ (\text{compile}\ c\ \mathit{prog}) \subseteq \\ \text{extend\_with\_resource\_limit}\ (\text{source\_sem}\ \mathit{ffi}\ \mathit{prog})\end{array}$$

Here extend_with_resource_limit is a function that augments a set of behaviours with the option to exit early with an out-of-memory error, c is a compiler configuration and ffi is a model of the outside world.

The partiality that is expressed using extend_with_resource_limit means that liveness properties proved at the source level do not transfer to liveness properties at the machine code level. For example, suppose one proves a liveness property that a source program will forever print "y" using a program logic [Åman Pohjola et al. 2019]. It does *not* follow from the compiler correctness theorem that the generated machine code will forever do the same: the partiality means that only safety properties carry over. The safety property in our example is that, if the machine code produces output, then the output consists of only “y”s.

In this paper, we define a predicate is_safe_for_space that is sufficient to rule out this partiality and extend the CakeML compiler proofs to give stronger guarantees for when is_safe_for_space holds. The is_safe_for_space predicate defines a space cost semantics for CakeML programs, and the new compiler correctness theorem states that the cost semantics rules out all potential for early termination. The new top-level theorem has the following shape:

$$\begin{array}{l}\text{is\_safe\_for\_space}\ \mathit{ffi}\ c\ \mathit{prog}\ \ldots \Rightarrow \\ \text{machine\_sem}\ \mathit{ffi}\ (\text{compile}\ c\ \mathit{prog}) = \text{source\_sem}\ \mathit{ffi}\ \mathit{prog}\end{array}$$

Note that the new relationship between source and target semantics here is equality, not refinement: the (deterministic) source semantics defines exactly one permitted behaviour, and the machine semantics implements precisely that behaviour. This equality means that liveness properties proved for the source level carry over directly to liveness properties of the machine code.

Contributions. This paper’s contributions are:

- We define a formal space cost semantics for the CakeML programming language. The definition is stated in terms of one of the intermediate languages used by the CakeML compiler. This intermediate language is at a high enough level to avoid reasoning about data pointers and heap objects, and yet at a low enough level to allow precise reasoning about heap and stack space usage. In addition, the semantics is designed to handle the most common forms of pointer aliasing found in functional programming languages. The cost semantics only considers the live part of the state and, as a result, can be used to derive space bounds for programs that call memory allocation an unbounded number of times.

- We prove that the cost semantics is sound for an end-to-end verified compiler that relies on (verified) garbage collection for correct operation. This is the first such result. The proof covers not only the compiled program, but also the implementation of the GC and the bignum library. When the cost semantics is used to rule out early exits, we get a strong compiler correctness theorem in terms of equality of observable behaviour, since all out-of-memory errors and other resource bound errors are avoided.
- We show that the cost semantics is concrete enough to prove specific space bounds for a few sample programs and, once bounds have been proved, liveness properties proved at the level of source code transfer directly to liveness properties about the compiler-generated machine code. This paper is the first to demonstrate that this is possible in the context of a verified compiler for a language whose compilation relies on automatic memory management. We consider both finite and infinite time liveness properties.

All of the work presented in this paper has been developed in the HOL4 theorem prover [Slind and Norrish 2008] and is available at https://code.cakeml.org.

Limitations. We delimit the scope of our investigation as follows. Our primary goal in this paper is to make space cost reasoning *possible*; making it *convenient* for CakeML users is future work. We do not consider (external) dynamic allocation: the CakeML binary asks the OS for all the memory it will ever need up-front, and manages its own stack and heap within this statically allocated region. Hence our memory model does not need to consider questions like "will the OS give us enough space when we call `malloc`?". For CakeML programs that use the foreign function interface (FFI), we do not model the space cost of code outside the FFI boundary; this does not impact soundness, because the foreign function cannot give memory it allocates back to CakeML.

2 OVERVIEW

This section describes our overall design and explains how the problem is divided up into separate parts. Subsequent sections describe these parts in more detail.

2.1 Why Can Generated Code Exit Early?

The cost semantics needs to predict when early exits might happen, so let us start by looking at the circumstances under which the code emitted by the CakeML compiler resorts to an early exit. The circumstances are:

(H) the creation of a new heap element (e.g. a datatype constructor, array, or bignum integer) does not fit into the heap, even after a full GC run;
(S) a function or primitive operation attempts to allocate stack past the end of the memory region reserved for representing the stack;
(L) the program tries to create an object whose length exceeds what can be represented in the bits reserved for the length field in heap objects;
(F) the program tries to run an incompatible primitive, e.g. a floating-point instruction on a target architecture that does not support it.

Case H is an out-of-heap error. Case S is an out-of-stack error. Cases L and F are possibly more exotic. One could argue that the compiler should catch many instances of case L and F at compile time. However, for case L, this is not always possible because the length of new arrays and vectors can be computed dynamically. Regarding case F, we want to be able to compile CakeML's standard library (which includes floating-point primitives) to all target architectures; thus the compiler will generate some code for all primitives.

2.2 Where Are the Early Exits Generated?

The CakeML compiler uses 8 intermediate languages and makes in total more than 40 compilation passes over its input, but only two compilation passes insert code that can cause early exits. The relevant intermediate languages are the following:

- DataLang is an imperative language where values are abstract and integers arbitrarily large; there is no notion of garbage collector in this language (see Section 3).
- WordLang has a similar structure to DataLang but values are machine words and memory is an array of machine words; the garbage collector is an opaque primitive.
- StackLang has a concrete stack: the stack is a fixed-size array of machine words. The GC stops being a primitive; the compiler inserts code implementing it.

Early exits for cases H, L and F are inserted by the compilation pass that converts DataLang into WordLang, and early exits for case S are inserted by the WordLang to StackLang pass.

2.3 At What Level of Abstraction Should the Cost Semantics Be Expressed?

At first glance, it seems most natural to express the cost semantics at the level of the source semantics. However, since we are interested in sound, concrete and *tight* bounds rather than asymptotic bounds, a source-level approach would have several drawbacks.

The CakeML compiler makes many function-call-related optimisations [Owens et al. 2017] that significantly improve speed and space usage. Because these optimisations mostly happen before the compiler phases that can introduce early exits, a source-level cost semantics must either (1) use very loose approximations of space usage, or (2) specify exactly which optimisations will be applicable on the given program, essentially re-implementing the compiler inside the cost semantics.

We consider both alternatives unacceptable. Our approach is based on the insight that instead of re-implementing the compiler inside the cost semantics, we can obtain the same precision by applying the compiler optimisations to the program under consideration *before* space cost analysis.

Hence our cost semantics is expressed at the level of the DataLang intermediate language. This allows us to be very precise with respect to resource usage without encumbering the cost semantics with compiler implementation details.

The main drawback of our approach is that users need to reason about an intermediate representation of their program after closure conversion. A certain non-compositionality is also inherent in this approach: since CakeML is a whole-program compiler, we cannot in general reuse the cost analysis of individual modules or functions across different programs. The applications we have in mind at present are statically allocated embedded systems, where memory is precious and must be declared up-front. For situations where loose or asymptotic bounds are acceptable, our cost semantics is much too detailed to be practical, and a source-level cost semantics with generous over-approximation of actual memory usage would likely be a better tool.

2.4 Definition of is_safe_for_space

As motivated above, we define our cost semantics based on the DataLang level of abstraction. The following is our definition of is_safe_for_space, which is our criterion for determining whether a *source-level program* is safe for space.

$$
\begin{array}{l}
\text{is\_safe\_for\_space}\ \mathit{ffi}\ c\ \mathit{prog}\ \mathit{stack\_heap\_limit} \stackrel{\text{def}}{=} \\
\quad \text{let}\ \mathit{data\_prog}\ =\ \text{fst}\ (\text{to\_data}\ c\ \mathit{prog})\ ;\ \mathit{word\_prog}\ =\ \text{to\_word}\ c\ \mathit{prog}\ \text{in} \\
\qquad \text{data\_lang\_safe\_for\_space}\ \mathit{ffi}\ \mathit{data\_prog} \\
\qquad\quad (\text{compute\_limits}\ c\ c.\text{data\_conf.has\_fp\_ops}\ c.\text{data\_conf.has\_fp\_tern}\ \mathit{stack\_heap\_limit}) \\
\qquad\quad (\text{compute\_stack\_frame\_sizes}\ c\ \mathit{word\_prog})\ \text{Start\_location} \wedge c.\text{data\_conf.gc\_kind} \neq \text{None}
\end{array}
$$

In this definition, to_data compiles the source program *prog* to DataLang; then data_lang_safe_for_space is used to decide whether the resulting DataLang program is safe for space (see Section 3).

The data_lang_safe_for_space predicate takes several arguments: the initial state of the foreign function interface *ffi*, the DataLang program *data_prog*, the configuration of limits, the mapping describing the size of each stack frame, and finally the start location in the program.

The definition above mentions to_word which compiles the source program *prog* to WordLang. The input source program is compiled to WordLang in order to compute the size of stack frames for each function that appears in the DataLang program. The cost semantics for DataLang tracks stack usage based on the provided stack frame size mapping (see Section 5).

The last conjunct of the definition requires the compiler configuration c to have gc_kind not equal to None (i.e. some garbage collector needs to be used). The other alternatives are Simple for a non-generational copying GC [Myreen 2010], and Generational for a generational collector [Sandberg Ericsson et al. 2019]. Our cost semantics requires a GC to be installed, therefore None is a disallowed configuration. We have proved our cost semantics sound w.r.t. the implementation of both the Simple and the Generational GC.

2.5 A Note on Semantics

The semantics of CakeML, and all of its intermediate languages, is defined in the functional big-step semantics style [Owens et al. 2016]. The core of such a semantics is a clocked big-step evaluation function evaluate which maps (state, program) pairs to (state, result) pairs. The state includes a clock which decrements at every instruction that might potentially induce divergence (such as function calls); if the clock runs out, evaluate aborts with a special timeout result. The state also includes a trace of all I/O events that have happened so far.

The top-level observable semantics function (called semantics) is defined based on the evaluate function described above. It returns a set of *behaviours*, where a behaviour is one of the following:

Terminate *reason events* — indicates that, for some clock value, evaluate terminates in a well-defined way (for a specific *reason*) after producing the I/O events *events*.

Diverge *events* — indicates that, for every clock value, evaluate times out, and *events* is the supremum of the I/O traces produced by evaluate for different initial clock values.

Fail — indicates that the semantics gets stuck for some clock value; well-typed source programs do not get stuck.

The function extend_with_resource_limit extends a set of behaviours to allow early termination with an out-of-memory error, i.e. Terminate where the reason is Resource_limit_hit. Here $\preccurlyeq$ checks whether the first list is a prefix of the second, l is a finite list of characters, and ll is a finite or infinite list of characters.

$$
\begin{array}{l}
\text{extend\_with\_resource\_limit}\ \mathit{behaviours} \stackrel{\text{def}}{=} \\
\quad \mathit{behaviours} \cup \\
\quad \{\ \text{Terminate Resource\_limit\_hit}\ \mathit{io\_list} \mid \exists\, t\ l.\ \text{Terminate}\ t\ l \in \mathit{behaviours} \wedge \mathit{io\_list} \preccurlyeq l\ \} \cup \\
\quad \{\ \text{Terminate Resource\_limit\_hit}\ \mathit{io\_list} \mid \exists\, ll.\ \text{Diverge}\ ll \in \mathit{behaviours} \wedge \mathit{io\_list} \preccurlyeq ll\ \}
\end{array}
$$

2.6 Structure of the Proofs

The aim of our proofs is to show that the observational semantics is preserved completely, i.e. the semantics functions are related with equality = rather than ... ⊆ extend_with_resource_limit ... as described in the introduction.

Nearly all compiler phases preserve observational semantics with equality, so no changes are required to those. Recall from Section 2.1 that the two phases that use the weaker relationship are: the DataLang-to-WordLang phase, which quits on out-of-heap errors and cases L and F from Section 2.1; and the WordLang-to-StackLang phase, which quits on out-of-stack errors.

For both of these phases, we define a predicate which implies that the observational semantics is related by = directly. For the DataLang-to-WordLang phase, this is data_lang_safe_for_space. For the WordLang-to-StackLang phase, we define a similar predicate, called word_lang_safe_for_space.

In order to avoid burdening the user with proofs in two cost semantics, we instrument DataLang with enough stack size tracking to prove that data_lang_safe_for_space implies word_lang_safe_for_space. As a result, users only need to prove data_lang_safe_for_space, as shown in Section 2.4.

3 DATALANG AND ITS SEMANTICS

As of this paper, DataLang has two roles: it acts as an intermediate language of the CakeML compiler, and it defines the heap and stack cost semantics for the compiler.

3.1 DataLang As an Intermediate Language

DataLang is an imperative language with abstract values, stateful storage of local variables, and a call stack. The semantics of DataLang models primitive values with the following datatype:

```
v  =  Number int
     | Word64 word64
     | CodePtr num
     | RefPtr num
     | Block timestamp tag (v list)
```

Here Number represents an arbitrarily large integer. Word64 is a 64-bit machine word. CodePtr is a code pointer, and RefPtr is a pointer to mutable state (such as ML arrays). The Block constructor is more interesting: it is used to encode datatype constructors, tuples and vectors. For instance, the CakeML list [1,2] is represented using DataLang Blocks as:

```
Block 8 cons_tag [Number 1;
                  Block 7 cons_tag [Number 2;
                                    Block 0 nil_tag []]]
```

The tag values, cons_tag and nil_tag, indicate the source-level constructor that each Block represents; this tag information is used for pattern matching. The timestamps of the blocks are 8, 7 and 0, respectively; we will explain the purpose of timestamps in Section 3.3.

The runtime state of DataLang's semantics is represented by a record type state shown in Figure 1. The fields locals and refs represent the finite maps of local variables (v num_map) and references (v ref num_map) respectively. The stack is a list of frames, each frame containing only the relevant variables that should be restored after a call is completed. On exception, the length of the stack is set to be equal to handler, dropping the most recent frames and setting the value of handler according to the new current frame. The global field contains an optional reference to an array of global variables. The space field is a guaranteed amount of space available in the heap, and can be increased by doing allocation. This is for bookkeeping only; the DataLang semantics maintains the fiction that more space can always be allocated. Finally, the remaining fields (some

```
α state = ⦇
  locals : v num_map;
  refs : ref num_map;
  stack : stack list;
  handler : num;
  global : num option;
  space : num;
  code : (num × prog) num_map;
  ffi : α ffi_state;
  clock : num;
  ...
⦈

ref = ValueArray (v list) | Bytes bool (word8 list)
```

Fig. 1. The definition of the DataLang state.

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

Fig. 2. DataLang's abstract syntax.

of them elided in Figure 1) pertain to the code store, the state of the foreign function interface, and the semantic clock, respectively.

DataLang's abstract syntax (see Figure 2) provides most of the expected features for an imperative language. A notable omission is looping constructs because functional programs use (tail) recursion, which is available as part of Call. In the abstract syntax, var (a type alias for the type of natural numbers) represents variable names; var_set is a set of local variables that are to be included in the stack frame when performing a Call, and should be considered live by the garbage collector when allocating (MakeSpace). The evaluation of a DataLang program returns an optional result along with a new state. We give a few samples of DataLang's evaluation semantics next.

The simplest program is Skip. It does nothing. The result is (None,s) because it does not interrupt the normal forwards flow of execution, and the state s is unchanged.

$$\text{evaluate }(\text{Skip},s) \overset{\text{def}}{=} (\text{None},s)$$

Sequencing (Seq) continues execution from c_1 to c_2 as long evaluation of c_1 indicates (with None) that execution is to continue forwards:

```
evaluate (Seq $c_1$ $c_2$,$s$) $\overset{\text{def}}{=}$
 let ($res$,$s_1$) = evaluate ($c_1$,$s$) in
  if $res$ = None then evaluate ($c_2$,$s_1$) else ($res$,$s_1$)
```

All of the primitive operations are performed by Assign, which deletes unused variable bindings, then reads the values of its arguments, and finally performs the primitive operations using the helper function do_app.

```
evaluate (Assign $dest$ $op$ $args$ $names\_opt$,$s$) $\overset{\text{def}}{=}$
  case cut_state_opt $names\_opt$ $s$ of
   Some $s$ ⇒
   case get_vars $args$ $s$.locals of
    Some $xs$ ⇒
    case do_app $op$ $xs$ $s$ of
     Rval ($v$,$s$) ⇒ (None,set_var $dest$ $v$ $s$)
    | Rerr $e$ ⇒ (Some (Rerr $e$),$s$)
  | ... ⇒ (Some (Rerr (Rabort Rtype_error)),$s$)
```

3.2 A DataLang Program from a User's Point of View

To illustrate the look and feel of a DataLang program from a user's point of view, we show below what a `foldl` function for lists looks like for users of our cost semantics.

At the source level, we define the `foldl` function as follows:

```
fun foldl f e l = case l of [] => e
                          | (x::xs) => foldl f (f e x) xs
```

By compiling `foldl` down to DataLang and then performing simple rewriting in the logic over the resulting DataLang AST, we arrive at the following presentation of how DataLang's evaluate executes the `foldl` function. Here the HOL term is expressed in terms of a shallowly embedded state-exception monad that has been proved sound w.r.t. evaluate. Note that source-level function names (in this case only foldl) are visible in this representation (lines 1 and 16).

```
1.  foldl [0; 1; 2] evaluates as
2.      4 := TagLenEq 0 0 [0]
3.      if_var 4 (return 1)
4.          do 6  := ElemAt 0 [0]
5.             7  := ElemAt 1 [0]
6.             13 := ElemAt 1 [2]
7.             14 := EqualInt 1 [13]
8.             if_var 14
9.               do 15 := ElemAt 0 [2]
10.                   call_ptr 16 { 2; 7 } [6; 1; 2; 15]
11.                   18 := 16
12.              od
13.              do call_stub_1 17 { 2; 7 } [6; 1; 2]
14.                   18 := 17
15.              od
16.            tailcall_foldl [7; 18; 2]
17.         do
```

At DataLang variables are natural numbers. On line 1, arguments 2, 1, 0 respectively correspond to `f`, `e`, `l` in the source code. The assignment (:=) on line 2 uses TagLenEq to check whether its argument `l` is a block with payload length and tag equal to zero—in other words, whether it is the empty list. The result is stored in variable 4. Line 3 branches based on the value in variable 4. On true, `return 1` is executed (returning `e`). On false, execution continues at line 4. Lines 4 and 5 read the contents of the Block representation of list cons using ElemAt. Lines 6–15 deal with the function closure in variable 2. Closures are represented as Block values with a payload consisting of a CodePtr to the function body, the arity of the function minus 1, and the values of all of the free variables of the closure body. Lines 6–8 branch on the arity of the closure. If the closure has the desired arity (in this case 2, we store arity minus 1), then lines 9–12 evaluate the application of the closure (`f e x`) by making a function call (with CodePtr as its last argument) and then storing the result in variable 18. Otherwise, lines 13–15 call library code that deals with the special case of an arity mismatch, which is allowed (see Owens et al. [2017]). Finally, line 16 performs a recursive tail-call with variables 7 (the tail of input list `l`), 18 (the new value for `e`), and 2 (the function `f`).

3.3 DataLang As a Cost Semantics

DataLang provides a convenient level of abstraction for reasoning about space consumption since functions are first-order and data has predictable size. However, DataLang's semantics has no notion of the heap, does not specify which data elements are heap allocated, and does not represent stack frames in a way that makes their size clear. Therefore, we need to add some mechanisms to make DataLang suitable for accurate space measurements. We add elements to the semantics state of DataLang to model the following:

1. A measurement of heap cost: the total space consumed by all values that would be heap-allocated by the implementation. This measure should only count live data, and so needs to be unchanged by garbage collection.
2. A measurement for stack frame sizes, and subsequently the call stack, that is consistent with their implementation in StackLang. We defer further explanation of stack costs to Section 5.
3. A signalling mechanism to track if at any point during execution either the stack or the heap surpassed some given limits. The signal is implemented as a new field called safe_for_space in the state record of DataLang.

These elements are represented as follows:

```
α state = ⦇
  ...
  safe_for_space : bool
  stack_frame_sizes : num num_map;
  limits : limits;
⦈
```

```
limits = ⦇
  heap_limit : num;
  length_limit : num;
  stack_limit : num;
  arch_64_bit : bool
⦈
```

The safe_for_space field is true as long as program evaluation stays within the limits. We say that a program *prog* is safe for space with respect to some *limits*, if every execution, regardless of the value of the initial clock *ck*, manages to keep safe_for_space set to true:

```
data_lang_safe_for_space ffi prog limits ss main ≝
 ∀ ck res s.
    evaluate (Call None (Some main) [] None,initial_state ffi prog limits ss ck) = (res,s) ⇒
      s.safe_for_space
```

At every memory allocation, the semantics computes the size of the live data in the heap, adds this number to the requested space k, and checks whether we might be exceeding the heap limit:

$$\text{size\_of\_heap}\ s + k \leq s.\text{limits.heap\_limit}$$

The semantics also checks that the stack size is below the stack limit at every function call. If either of these tests fail at any point, safe_for_space is set to false. Further down, after discussing aliasing, we will show the definition of size_of_heap.

Aliasing information. Before presenting our strategy for computing the live heap data, we explain how the semantics maintains aliasing information. Functional programs give rise to a lot of pointer aliasing. Consider, for example, the following snippet of ML code:

```
let val a = [1,2] in (a,0::a) end
```

This code evaluates to a tuple of two lists of integers, `[1,2]` and `[0,1,2]`. To accurately compute the size of this tuple value, the semantics needs to carry information from which we can infer that the memory representation of the two lists share a tail.

We add timestamps to the Block values of the DataLang semantics that let us detect when Block values are pointer-equal. Each new heap element gets a unique timestamp for all of its Blocks. Hence, by keeping timestamps invariant through a Block's lifetime, we can infer that any two Blocks that share a timestamp must refer to the same location on the heap.

The example of a tuple holding two integer lists above can be represented by the following value in DataLang by our semantics.

$$\begin{array}{l}
\text{block\_example} \overset{\text{def}}{=} \\
\quad \text{let } a \ = \\
\qquad\quad \text{Block } 8 \text{ cons\_tag} \\
\qquad\qquad [\text{Number } 1;\ \text{Block } 7 \text{ cons\_tag } [\text{Number } 2;\ \text{Block\_nil}]] \text{ in} \\
\qquad \text{Block } 10 \text{ tuple\_tag } [a;\ \text{Block } 9 \text{ cons\_tag } [\text{Number } 0;\ a]]
\end{array}$$

If we expand the above let-expression, it is clear that Blocks with timestamps 8 and 7 repeat.

We compute the size of all live data on the heap using a function called size_of that is aware of the meaning of timestamps. Before we define it, let us consider its application to the above example. We have that size_of returns 12 when applied to block_example. The size_of function counts each two-element Block as size 3 and each zero-element Block as size 0. The example above has 4 unique two-element Blocks, thus $4 \times 3 = 12$. The unit is machine words of heap space.

$$\vdash \text{fst (size\_of [block\_example] empty empty)} = 12$$

It is worth mentioning that a naive size measure that ignores aliasing information would have produced an over-approximation of $6 \times 3 = 18$, because there are 6 non-empty Blocks in the block_example.

Computing the size of the heap. The following are some of the equations of the definition of size_of. Other equations are provided further down.

```
size_of [] $refs$ $seen$ $\overset{\text{def}}{=}$ $(0,refs,seen)$
size_of [Number $i$] $refs$ $seen$ $\overset{\text{def}}{=}$
  (if is_smallnum $i$ then $0$ else bignum_size $i$,$refs$,$seen$)
size_of [Block $ts$ $tag$ $vs$] $refs$ $seen$ $\overset{\text{def}}{=}$
  if $vs = [] \lor ts \in seen$ then $(0,refs,seen)$
  else
    let $(n,refs',seen')$ = size_of $vs$ $refs$ $(\{ts\} \cup seen)$ in
      $(n + |vs| + 1,refs',seen')$
size_of $(x{::}xs)$ $refs$ $seen$ $\overset{\text{def}}{=}$
  let $(n_1,refs_1,seen_1)$ = size_of $xs$ $refs$ $seen$ ;
      $(n_2,refs_2,seen_2)$ = size_of $[x]$ $refs_1$ $seen_1$ in
    $(n_1 + n_2,refs_2,seen_2)$
```

Small numbers are stored within their containing block or stack frame, and hence they have heap size 0; bignums use heap space proportional to the number of digits in their binary representation.

Empty Blocks are stack-allocated and have heap size 0. The size_of function ignores Blocks with timestamps that are present in *seen*. In all other cases, Blocks add the length of their payload plus one to the first return value of size_of. The size_of function uses *seen* to avoid counting the same Block twice.

The size_of function avoids counting references twice by deleting them from the reference store that it carries in the *refs* variable:

```
size_of [RefPtr $r$] $refs$ $seen$ $\overset{\text{def}}{=}$
  case lookup $r$ $refs$ of
    None $\Rightarrow$ $(0,refs,seen)$
  | Some (ValueArray $vs$) $\Rightarrow$
      (let $(n,refs',seen')$ = size_of $vs$ (delete $r$ $refs$) $seen$
       in $(n + |vs| + 1,refs',seen')$)
  | Some (ByteArray $v_2$ $bs$) $\Rightarrow$ $(|bs|$ div $4 + 2$,delete $r$ $refs$,$seen)$
```

We define size_of_heap as size_of applied to all of the values stored in the DataLang state's stack and global variables.

```
size_of_heap $s$ $\overset{\text{def}}{=}$
  let $(n,\_,\_)$ = size_of (stack_to_vs $s$) $s$.refs empty in $n$
```

The size_of_heap function is used in the semantics whenever an operation that would allocate heap space is executed: if ever size_of_heap plus the amount of heap space requested exceeds the limits, we set is_safe_for_space to false.

4 PROVING SOUNDNESS OF HEAP COST

Before this work, the DataLang-to-WordLang phase of the compiler had a correctness theorem phrased in terms of ⊆ and extend_with_resource_limit in order to allow early exits:

$$\ldots \Rightarrow$$
$$\text{semantics}_{\text{word}}\ \mathit{ffi}\ (\text{compile}\ c\ \mathit{prog}) \subseteq \text{extend\_with\_resource\_limit}\ (\text{semantics}_{\text{data}}\ \mathit{ffi}\ \mathit{prog})$$

As part of this work, we have proved a new alternative correctness theorem which states that, if data_lang_safe_for_space is true, then all behaviours are preserved by equality =.

$$\begin{array}{l} \ldots \wedge \text{data\_lang\_safe\_for\_space}\ \mathit{ffi}\ \mathit{prog}\ \ldots \Rightarrow \\ \text{semantics}_{\text{word}}\ \mathit{ffi}\ (\text{compile}\ c\ \mathit{prog})\ =\ \text{semantics}_{\text{data}}\ \mathit{ffi}\ \mathit{prog} \end{array}$$

One can read this as saying that cost semantics for DataLang is sound. The following subsections discuss our proof of this soundness result.

4.1 Proving evaluate-Level Simulation

Each proof about the relationship between observational semantics (i.e. semantics) is based on a theorem relating the evaluate functions of the languages involved. In order to prove the new semantics theorem that was sketched above, we need to update the main evaluate simulation theorem to state that DataLang's evaluate correctly predicts any early exits that the generated WordLang program might have resorted to.

The theorem describing the evaluate simulation has the following shape, which is similar to most CakeML compiler phases [Tan et al. 2019]. One can informally read it as follows: if the input program *prog* evaluates to some result (res,s_1) without hitting a dynamic type error (Rabort Rtype_error), then the compiled program, comp c *prog*, will evaluate to a final state that is similar enough according to a state relation state_rel. Here variable c is a compiler configuration.

$$\begin{array}{l} \vdash \text{evaluate}_{\text{data}}\ (\mathit{prog},s) = (\mathit{res},s_1) \wedge \text{state\_rel}\ c\ s\ t \wedge \mathit{res} \neq \text{Some (Rerr (Rabort Rtype\_error))} \Rightarrow \\ \quad \exists\, t_1\ \mathit{res}_1. \\ \quad\quad \text{evaluate}_{\text{word}}\ (\text{comp}\ c\ \mathit{prog},t) = (\mathit{res}_1,t_1) \wedge \\ \quad\quad (\mathit{res}_1 = \text{Some NotEnoughSpace} \Rightarrow \\ \quad\quad\quad t_1.\text{ffi.io\_events} \preccurlyeq s_1.\text{ffi.io\_events} \wedge \boxed{(c.\text{gc\_kind} \neq \text{None} \Rightarrow \neg s_1.\text{safe\_for\_space})}\ ) \wedge \\ \quad\quad (\mathit{res}_1 \neq \text{Some NotEnoughSpace} \Rightarrow \text{state\_rel}\ c\ s_1\ t_1 \wedge \ldots) \end{array}$$

Compared with other CakeML compiler phases, the unusual part here is the special case for the NotEnoughSpace result. For this result, the original theorem only concluded that the WordLang state's I/O events are a prefix (≼) of the I/O events produced by the DataLang program *prog*.

For the cost semantics proofs, we added the part in a $\boxed{\text{box}}$. This box adds that, whenever WordLang resorts to a NotEnoughSpace error result, the DataLang evaluation predicts that this might happen, if a supported GC configuration is used. Thus for the user to prove that the WordLang program never exits early, it suffices to prove that DataLang says it will not happen.

The proof of the evaluate simulation theorem sketched above is complicated and long. The original proof builds on some 35,000 lines of invariant definitions and proofs. All of these were updated to cope with the change highlighted above. Handling early exits due to reasons L and F (from Section 2.1) was straightforward. The cases that arise when heap space runs out (case H) are much more interesting and will be discussed in the following subsections.

4.2 Notation and Invariants

In this section, we explain the heap abstractions we use to prove our heap cost analysis sound wrt. the WordLang program in Section 4.3. The state relation used in the DataLang-to-WordLang proofs is defined in terms of several layers of abstraction. Fortunately, the most abstract intermediate layer is sufficient for our proofs. In that layer, the heap is modelled as a list of heap_elements:

$$\begin{array}{l}
(\alpha,\ \beta)\ \mathsf{heap\_element}\ =\\
\quad \mathsf{Unused\ num}\\
\ \ |\ \mathsf{ForwardPointer\ num}\ \alpha\ \mathsf{num}\\
\ \ |\ \mathsf{DataElement}\ (\alpha\ \mathsf{heap\_address\ list})\ \mathsf{num}\ \beta
\end{array}$$

$$\alpha\ \mathsf{heap\_address}\ =\ \mathsf{Pointer\ num}\ \alpha\ |\ \mathsf{Data}\ \alpha$$

During normal program execution, the heap consists of only DataElements and Unused. ForwardPointers only exist while the GC runs. The natural number (type num in HOL) in Pointers is the address. We dereference pointers using heap_lookup based on a natural number address a:

$$\begin{array}{l}
\mathsf{heap\_lookup}\ a\ []\ \stackrel{\text{def}}{=}\ \mathsf{None}\\
\mathsf{heap\_lookup}\ a\ (x{::}xs)\ \stackrel{\text{def}}{=}\\
\ \ \mathsf{if}\ a = 0\ \mathsf{then\ Some}\ x\\
\ \ \mathsf{else\ if}\ a < \mathsf{el\_length}\ x\ \mathsf{then\ None}\\
\ \ \mathsf{else\ heap\_lookup}\ (a - \mathsf{el\_length}\ x)\ xs
\end{array}$$

$$\begin{array}{l}
\mathsf{el\_length}\ (\mathsf{Unused}\ l)\ \stackrel{\text{def}}{=}\ l + 1\\
\mathsf{el\_length}\ (\mathsf{ForwardPointer}\ n\ d\ l)\ \stackrel{\text{def}}{=}\ l + 1\\
\mathsf{el\_length}\ (\mathsf{DataElement}\ xs\ l\ data)\ \stackrel{\text{def}}{=}\ l + 1
\end{array}$$

In the DataLang-to-WordLang proofs, the relationship between DataLang's values and their abstract heap representation is specified by the predicate v_inv. We show the Number and Block cases of v_inv below. A number is represented as a value that will fit in a register if the number is small enough, and otherwise by a pointer to a heap element containing the large number. We omit the definition of Bignum, which is a form of DataElement.

$$\begin{array}{l}
\mathsf{v\_inv}\ c\ (\mathsf{Number}\ i)\ (x{,}f{,}t{,}heap)\ \stackrel{\text{def}}{=}\\
\ \ \mathsf{if\ is\_smallint}\ i\ \mathsf{then}\ x = \mathsf{Data}\ (\mathsf{Word}\ (\mathsf{Smallnum}\ i))\\
\ \ \mathsf{else}\\
\ \ \ \exists\, ptr.\\
\ \ \ \ \ \ x = \mathsf{Pointer}\ ptr\ (\mathsf{Word}\ 0w)\ \land\\
\ \ \ \ \ \ \mathsf{heap\_lookup}\ ptr\ heap = \mathsf{Some}\ (\mathsf{Bignum}\ i)
\end{array}$$

In our work, we changed the definition of v_inv for the Block case: we added a parameter t which dictates how timestamps stored in Blocks map to addresses in the heap. The fact that the timestamp dictates the representation address means that Blocks are pointer equal if their timestamp coincide. The new part is highlighted with a $\boxed{\text{box}}$.

$$\begin{array}{l}
\mathsf{v\_inv}\ c\ (\mathsf{Block}\ ts\ n\ vs)\ (x{,}f{,}t{,}heap)\ \stackrel{\text{def}}{=}\\
\ \ \mathsf{if}\ vs = []\ \mathsf{then}\ x = \mathsf{Data}\ (\mathsf{Word}\ (\mathsf{BlockNil}\ n))\ \land \ldots\\
\ \ \mathsf{else}\\
\ \ \ \exists\, ptr\ xs.\\
\ \ \ \ \ \ \boxed{\mathsf{lookup}\ t\ ts = \mathsf{Some}\ ptr\ \land}\\
\ \ \ \ \ \ \mathsf{list\_rel}\ (\lambda\, v\ x.\ \mathsf{v\_inv}\ c\ v\ (x{,}f{,}t{,}heap))\ vs\ xs\ \land\\
\ \ \ \ \ \ x = \mathsf{Pointer}\ ptr\ (\mathsf{Word}\ (\mathsf{ptr\_bits}\ c\ n\ |xs|))\ \land\\
\ \ \ \ \ \ \mathsf{heap\_lookup}\ ptr\ heap =\\
\ \ \ \ \ \ \ \ \mathsf{Some}\ (\mathsf{DataElement}\ xs\ |xs|\ (\mathsf{BlockTag}\ n{,}[]))
\end{array}$$

Finally, the next subsection uses the following combination of heap_lookup and el_length.

$$\mathsf{get\_len}\ heap\ p\ \stackrel{\text{def}}{=}\ \mathsf{case\ heap\_lookup}\ p\ heap\ \mathsf{of\ None}\ \Rightarrow\ 0\ |\ \mathsf{Some}\ x\ \Rightarrow\ \mathsf{el\_length}\ x$$

$$\frac{}{\text{traverse}\ heap\ p_1\ []\ p_1}$$

$$\frac{\text{traverse}\ heap\ p_1\ vs_1\ p_2 \qquad \text{traverse}\ heap\ \ p_2\ vs_2\ p_3 \qquad \text{set}\ vars = \text{set}\ (vs_1 \mathbin{++} vs_2)}{\text{traverse}\ heap\ p_1\ vars\ p_3}$$

$$\frac{}{\text{traverse}\ heap\ p_1\ [\text{Data}\ d]\ p_1}$$

$$\frac{\text{mem}\ n\ p_1}{\text{traverse}\ heap\ p_1\ [\text{Pointer}\ n\ t]\ p_1}$$

$$\frac{\text{heap\_lookup}\ n\ heap = \ \text{Some}\ (\text{DataElement}\ xs\ l\ d) \qquad \text{traverse}\ heap\ (n{::}p_1)\ xs\ p_2}{\text{traverse}\ heap\ p_1\ [\text{Pointer}\ n\ t]\ p_2}$$

Fig. 3. Definition of traverse.

4.3 Correctness of Heap Allocation and size_of

The DataLang semantics decides that a heap allocation is *not safe for space* if the following test returns *true*. Here k is the number of words of space that have been requested.

$$s\text{.limits.heap\_limit} < \text{size\_of\_heap}\ s + k$$

This section describes our soundness proof for this test, i.e. why this test at the DataLang level must return true whenever an allocation failure happens at the WordLang level.

At the WordLang level, a heap allocation failure happens only when not enough space is available after a full (compacting) GC run. Since the GC has run, we can assume that all of the DataElements in the heap are reachable from the root variables. And since the WordLang space test has failed, we can assume that the total amount of Unused space in the heap—call it sp—is not sufficient to satisfy the allocation request, i.e. $sp < k$. Thus it suffices to show:

$$s\text{.limits.heap\_limit} \leq \text{size\_of\_heap}\ s + sp$$

which is equivalent to:

$$s\text{.limits.heap\_limit} - sp \leq \text{size\_of\_heap}\ s$$

The left-hand side above is the same as the sum of the lengths of all heap elements in the DataElement-filled part of the heap. We will call this part of the heap: $heap$. Thus it suffices to prove:

$$\text{sum}\ (\text{map el\_length}\ heap) \leq \text{size\_of\_heap}\ s$$

We have now arrived at the tricky part of this proof: the statement above requires us to prove that every data element in $heap$ must be counted (at least once) by size_of_heap, which is defined in terms of the size_of function. This is tricky because the size_of function has a slight disconnect from semantic state: it skips blocks with timestamps that it has accumulated in its $seen$ argument and deletes reference values from its *refs* argument during recursion, which means that it cannot evaluate all reference pointers that it encounters even when they exist in the actual heap.

In order to make this proof manageable, we introduce a new inductively defined relation, called traverse, which captures abstractly the traversal patterns that size_of implements using its arguments $seen$ and *refs*. The definition of traverse is shown in Figure 3. The traverse relation takes four arguments: $heap$, p_1, $vars$, p_2. Here $heap$ is the heap being traversed; p_1 and p_2 are lists of addresses which can be viewed as states: p_1 is the input state, and p_2 is the output state; finally $vars$ is a working list of heap addresses under consideration. The first rule states that the output state must be equal to the input state if $vars$ is empty. The second rule shows how the working list can be split and the state threaded through. The third rule states that traverse can skip data elements on the working list. The fourth rule is more interesting: it states that traverse can skip a pointer if that pointer is already in the input state. The last rule allows traverse to lookup a heap element and place its payload on the working list. For the the last rule, it is worth noting that the traversal of the payload happens from state $n{::}p_1$, i.e. a state where the currently visited address n has already been added to state p_1; this allows traverse to break cycles in the graph of pointers in the heap.

With this definition of traverse, we can prove the following lemma that puts a lower bound on size_of. The following lemma assumes that we have DataLang $values$ that are v_inv-related to some $roots$ and $heap$, and that $refs$ are in a similar manner (ref_inv) represented in $heap$. If those assumptions hold, then traverse $heap$ [] $roots$ p_2 is true for some final state p_2. Furthermore, the sum of get_len applied to all addresses in p_2 is ≤ the first component of the result of size_of.

$$\begin{array}{l}
\vdash \mathsf{size\_of}\ values\ refs\ \mathsf{empty} = (n,r,s)\ \wedge \\
\quad \mathsf{v\_inv\_list}\ c\ roots\ (values,f,t,heap)\ \wedge \\
\quad (\forall\, n.\ n \in \mathsf{reachable\_refs}\ values\ refs \Rightarrow \mathsf{ref\_inv}\ c\ n\ refs\ (f,t,heap,be)) \Rightarrow \\
\quad\ \exists\, p_2.\ \mathsf{traverse}\ heap\ []\ roots\ p_2 \wedge \mathsf{sum}\ (\mathsf{map}\ (\mathsf{get\_len}\ heap)\ p_2) \le n
\end{array}$$

The proof of this lemma requires stating fiddly assumptions about the accumulated arguments of size_of, but is otherwise a reasonably straightforward proof by induction over the recursive structure of the size_of function.

Let us continue the soundness proof for the check of running out of space. In that context, we use the lower bound lemma from above to establish that there exists a p_2 such that:

$$\begin{array}{l}
\mathsf{sum}\ (\mathsf{map}\ (\mathsf{get\_len}\ heap)\ p_2) \le \mathsf{size\_of\_heap}\ s\ \wedge \\
\mathsf{traverse}\ heap\ []\ roots\ p_2
\end{array}$$

With this knowledge, it suffices to prove:

$$\mathsf{sum}\ (\mathsf{map}\ \mathsf{el\_length}\ heap) \le \mathsf{sum}\ (\mathsf{map}\ (\mathsf{get\_len}\ heap)\ p_2)$$

The rest of the proof establishes that every element of $heap$ has its address included in p_2 and is thus counted (at least once) in sum (map (get_len $heap$) p_2). The fact that every heap address is included in p_2 follows from the fact that a full GC has been run immediately prior to this, and from the following lemma which states that traverse finds all reachable addresses:

$$\vdash \mathsf{traverse}\ heap\ []\ roots\ p_2 \Rightarrow \mathsf{reachable\_addresses}\ roots\ heap \subseteq \mathsf{set}\ p_2$$

This concludes our sketch of the proof that the DataLang check for heap exhaustion is sound with respect to WordLang's check. The target language, WordLang, operates over a lower level of abstraction, but fortunately all of the tricky proofs were confined to the algorithm-level described above rather than lower layers of data refinements that lie between DataLang and WordLang.

4.4 Lessons Learned

Doing heap cost analysis at a level of abstraction where there is no heap has the advantage that reasoning can be carried out at a level closer to the source program. But when defining the size_of function, we were faced with an interesting trade-off between accuracy and ease of reasoning. Our implementation exploits timestamps to avoid counting the same block twice in the presence of aliasing. This significantly improves the tightness of our bounds, at the cost of encumbering the definition with accumulator arguments to keep track of which tags have been seen. This leads to a definition that fails to satisfy some natural algebraic laws; for example, size_of does not in general distribute over list append. It does for heaps that are well-formed in the sense that distinct data elements have distinct tags, but carrying around such well-formedness properties through proofs is cumbersome.

In situations where space is plentiful, precision might be less important than the question of whether there is a bound at all. There it might be more useful to have an imprecise size function that is tailored for ease of reasoning. To this end we have defined approx_of, an alternative to size_of that does not track timestamps and hence has nicer algebraic properties. We prove that approx_of is a sound over-approximation of size_of.

Another option is to make size_of even tighter by adding timestamps to data elements other than blocks. For example, our version will count pointer-equal bignums twice if they are aliased.

5 PROVING SOUNDNESS OF STACK COST

The DataLang and WordLang intermediate languages do not commit to a concrete implementation of the stack, and do not allow the programmer to manipulate the stack directly. The semantics of both languages model the stack as a list of *stack frames*, which consist of binding environments for local variables plus optional exception handlers. This list is allowed to grow unboundedly large; hence the semantics of both languages act as if stack space is unbounded.

In this section, we show how to make the DataLang and WordLang semantics stack space aware. As in Section 3, we add fields to their state records that track stack usage. These fields are a form of ghost state: they have no effect on the program's semantics beyond the fields themselves. But they are sound predictions of the program's maximum stack usage, and the compiler correctness theorem for the WordLang to StackLang phase—where the stack is implemented in a bounded memory region—shows that early exits due to out-of-stack errors never happen unless thus predicted.

As a first step, we annotate WordLang stack frames with an optional size (num option), measured in machine words:

$$\begin{array}{l}\mathsf{stack\_frame}\ =\\ \quad \mathsf{StackFrame}\ (\mathsf{num\ option})\ \mathsf{local\_env}\ (\mathsf{handler\ option})\end{array}$$

The intuition is that None here denotes positive infinity, or in other words, a stack frame whose size we have no upper bound for. Its inclusion allows us to preserve soundness in the presence of language features that are not safe for space.[1]

Note that we cannot simply compute a bound for the stack frame from the size of the local environment. This is because the environment does not necessarily contain *all* stack-allocated variables, only those that are treated as roots by the GC; moreover, this is before register allocation, so we do not yet know which local variables will be stack-allocated and which will be stored in registers. Moreover, a stack frame is allocated at the beginning of a function, but during the execution of the function there can be unused areas of the stack frame that are not apparent from inspecting the abstract representation of the local environment.

The size of the entire stack can then be computed as follows:

$$\begin{array}{l}\mathsf{stack\_size}\ (\mathsf{StackFrame}\ n\ l\ \mathsf{None}{::}stack)\ \overset{\text{def}}{=}\\ \quad \mathsf{option\_binop}\ (+)\ n\ (\mathsf{stack\_size}\ stack)\\ \mathsf{stack\_size}\ (\mathsf{StackFrame}\ n\ l\ (\mathsf{Some}\ handler){::}stack)\ \overset{\text{def}}{=}\\ \quad \mathsf{option\_binop}\ (+)\ (\mathsf{option\_map}\ ((+)\ 3)\ n)\ (\mathsf{stack\_size}\ stack)\\ \mathsf{stack\_size}\ []\ \overset{\text{def}}{=}\ \mathsf{Some}\ 1\end{array}$$

[1] The only language feature of WordLang whose stack usage we do not provide bounds for is the Install instruction for dynamic code evaluation. At present, this instruction is not targeted by the CakeML compiler.

The fact that this is not just a straightforward list sum exposes two compiler-specific implementation details that we include for the sake of more precise bounds: the empty stack is one word long, and installing an exception handler requires three words of stack space.

We annotate the WordLang state with an extra field stack_max, which records the largest stack_size seen so far during the WordLang execution. This field is updated to the maximum of the old value and the current stack_size whenever a WordLang instruction that potentially allocates stack is executed; the relevant instruction for our purposes is function calls.

To populate the stack frames with sizes, we assume that the state also contains a mapping, called stack_frame_sizes, which maps function names to stack frame sizes. It is possible to do symbolic computations about stack usage without committing to any particular mapping. To obtain sound and concrete bounds, the tooling we use in Section 7 obtains the actual stack frame sizes by evaluating the compiler in logic down to StackLang. This avoids cluttering the cost semantics with details of how register allocation is implemented.

These annotations allow us to soundly predict out-of-stack errors, as shown by the compiler correctness theorem for the WordLang-to-StackLang phase:

$$\begin{array}{l}
\vdash \mathsf{evaluate}\ (prog,s) = (res,s_1) \wedge res \neq \mathsf{Some\ Error} \wedge \mathsf{state\_rel}\ k\ f\ f'\ s\ t\ lens \wedge \ \ldots \\
\quad \Rightarrow \\
\quad \exists\, ck\ t_1\ res_1. \\
\qquad \mathsf{evaluate}\ (\mathsf{fst}\ (\mathsf{comp}\ prog\ bs\ (k,f,f')),t\ \mathsf{with\ clock}\ :=\ t.\mathsf{clock} + ck) = (res_1,t_1) \wedge \\
\qquad \mathsf{if}\ \mathsf{option\_map\ compile\_result}\ res \neq res_1\ \mathsf{then} \\
\qquad\quad res_1 = \mathsf{Some}\ (\mathsf{Halt}\ (\mathsf{Word}\ 2w)) \wedge \\
\qquad\quad t_1.\mathsf{ffi.io\_events} \preccurlyeq s_1.\mathsf{ffi.io\_events} \wedge \\
\qquad\quad \boxed{s_1.\mathsf{stack\_max} > s_1.\mathsf{stack\_limit}} \\
\qquad \mathsf{else} \\
\qquad\quad \ldots
\end{array}$$

The $\boxed{\text{boxed}}$ conjunct is the novelty and the key: it states that if StackLang evaluation yields an unexpected result (i.e. res and res' disagree), then this must have been due to an early exit that was predicted by WordLang evaluation exceeding the stack budget at some point.

In order to allow reasoning about costs in just the one semantics, we lift this stack cost semantics from WordLang to DataLang. The treatment of function calls does not change significantly between the two languages, so that aspect of the semantics is mostly the same. The main difference is that many native operators of DataLang, such as equality and bignum arithmetic, are implemented by canned code in WordLang. When this code features calls to subroutines, the DataLang semantics must make sure to update stack_max accordingly. Most of these subroutines are either tail-recursive or not recursive, in which case the stack consumption can be characterised as the largest of the involved WordLang stubs' stack frames.

The operator with the most interesting stack usage is probably the equality operator, which can compare arbitrarily nested trees of Blocks; its WordLang implementation must recursively step through these pointer structures and compare the payloads for equality. We prove that its stack usage is bounded from above by a metric on the constructor depth of the DataLang values that the pointer structures refine.

The DataLang-to-WordLang compiler also pastes in canned code that implements the bignum library. This code required some special attention regarding stack usage. The bignum library is reachable from any DataLang integer arithmetic operation that fails to fit within small enough numbers. The WordLang code implementing the bignum library is automatically generated from a higher-level specification [Myreen and Curello 2013] and consists of several nested WordLang functions. To ease the effort, we developed a verified tool that can automatically infer maximum stack depths of WordLang functions where all cycles in the call graph consist of tail-calls. The bignum library fits within this subset of WordLang.

5.1 Lessons Learned

Proving soundness of the stack cost semantics involved a tedious and cumbersome invariant preservation proof, but the effort invested helped us gain insight. Even though the stack cost semantics is relatively straightforward compared to heap cost, doing a formal soundness proof was invaluable for getting the cost semantics right down to every detail. There were a number of more or less subtle mistakes we made in early drafts of the semantics, that would have been difficult to catch and diagnose without formal proof:

- The WordLang semantics does not explicitly distinguish between whether the current local variables have already been pushed to the stack or not; this requires some care to avoid counting the current stack frame twice in the tally.
- In the StackLang implementation of function calls, stack allocation is done in two increments: enough space for the function arguments is allocated by the caller, then the callee allocates space for the remaining local variables. Our cost semantics abstracts away from this timing detail, which makes it important that we update stack_max before rather than after function calls; otherwise, our bounds will be unsound in case the Call instruction aborts.
- We initially modelled tail-calls as not changing stack size, but this is unsound if the tail-call is to another function with a larger stack size.
- Exception handler allocation needs to be counted separately from the rest of the stack frame size, as shown above, because the same function may be called both with and without exception handlers.

6 TOP-LEVEL COMPILER THEOREM WITH COST

We have proved a new end-to-end correctness statement for the entire CakeML compiler. In the theorem below, compile performs the entire compilation chain from concrete syntax down to machine code. The new theorem leverages is_safe_for_space to show that, for any successful compilation, execution from any machine state ms that has the compiler-generated $code$ and $data$ installed will produce exactly the same $behaviours$ as the source semantics.

$$\begin{array}{l}
\vdash \mathsf{compile}\ \mathit{cc}\ \mathit{prelude}\ \mathit{input} = (\mathsf{Success}\ (\mathit{code},\mathit{data},c),c') \Rightarrow \\
\quad \exists\, \mathit{behaviours}\ \mathit{source\_decs}. \\
\qquad \mathsf{semantics\_init}\ \mathit{ffi}\ \mathit{prelude}\ \mathit{input} = \mathsf{Execute}\ \mathit{behaviours} \wedge \\
\qquad \mathsf{parse}\ (\mathsf{lexer\_fun}\ \mathit{input}) = \mathsf{Some}\ \mathit{source\_decs} \wedge \\
\qquad \forall\, \mathit{ms}. \\
\qquad\quad \mathsf{is\_safe\_for\_space}\ \mathit{ffi}\ \mathit{cc}\ (\mathit{prelude} \mathbin{++} \mathit{source\_decs})\ (\mathsf{read\_limits}\ \mathit{cc}\ \mathit{ms}) \wedge \\
\qquad\quad \mathsf{installed}\ \mathit{code}\ \mathit{data}\ \ldots\ \mathit{mc}\ \mathit{ms} \Rightarrow \\
\qquad\quad \mathsf{machine\_sem}\ \mathit{ffi}\ \mathit{ms} = \mathit{behaviours}
\end{array}$$

Here we assume is_safe_for_space (i.e. require the user to prove it), but we conclude an equality machine_sem $\mathit{ffi}\ \mathit{ms} = \mathit{behaviours}$ instead of the weaker previous formulation that used $\subseteq$ and extend_with_resource_limit as explained in the introduction.

Here read_limits is a function that computes the relevant limits for the cost semantics based on information from the compiler configuration cc and the initial machine state ms.

7 PROVING THAT PROGRAMS ARE SAFE FOR SPACE

The aim of this paper is to provide a cost semantics that can be used to carry liveness properties proved at the source level down to the machine code level. In this section, we demonstrate that we can do exactly that with our new cost semantics on several examples.

7.1 A First Example

As a first example, we use a CakeML implementation of the Unix `yes` command, shown in Figure 4. This program prints its argument to `stdout` indefinitely.

```
fun put_line l = let
  val s = l ^ "\n"
  val a = Word8Array.array 0 (Word8.fromInt 0)
  val _ = #(put_char) s a (* ffi call *)
in () end;

fun printLoop l = (put_line l; printLoop l)

val _ = printLoop "y"
```

Fig. 4. Implementation of yes.

7.1.1 Is `yes` *Safe for Space?* Before we delve into a formal proof, let us convince ourselves that `yes` is indeed safe for space.

At first glance, we see a number of expressions within put_line that cause memory allocation. For example, string concatenation requires allocating space for the resulting string. Thus any call to printLoop, which recursively calls put_line indefinitely, will perform an unbounded number of allocations. This is fine since none of the variables in the body of put_line remain in scope, and hence will eventually be garbage collected. This in turn means that the heap footprint of printLoop, as measured by size_of_heap, does not increase between loop iterations.

As for the stack, it is enough to notice that (1) put_line is a non-recursive terminating function that consumes a bounded amount of stack space, and (2) printLoop is tail-recursive, and thus its recursive calls to itself do not grow the stack.

Informally, we conclude that `yes` must be safe for space, even though it is not clear yet with respect to what heap and stack bounds.

7.1.2 Is `yes` *Safe for Space, Formally?* We formalise the above intuition by showing that evaluation of the `yes` program satisfies is_safe_for_space as defined in Section 2.4. In order to avoid encumbering the proofs with a deeply embedded semantics, we have developed a sound and complete shallowly embedded representation of DataLang programs as a state monad for doing space cost reasoning; an example of this monadic representation is shown in Section 3.2.

Most of the initial DataLang code generated by the compiler can easily be evaluated in-logic from the concrete initial state; it is only when we reach the body of printLoop that things get interesting. The body of the printLoop looks as follows in the proof.

$$\begin{array}{l}\mathsf{Seq}\ (\mathsf{Call\_put\_line}\ (\mathsf{Some}\ (1, \{0\}))\ [0]\ \mathsf{None})\\ \qquad (\mathsf{Call\_printLoop}\ \mathsf{None}\ [0]\ \mathsf{None})\end{array}$$

This corresponds very closely to the source program. The local variable 0 holds the value of "y". Abbreviations to make function calls readable are automatically installed; for example, Call_printLoop abbreviates $\lambda\, ret.$ Call ret (Some 285), where 285 is the code location where the DataLang code generated from printLoop happens to be installed. From this point onwards, `put_line` will recursively call itself indefinitely, and thus data_is_safe_for_space can be proven by complete induction over the semantic clock, and provide us with the following bounds:

$$\begin{array}{l}\vdash \text{the (size\_of\_stack s.stack)} + 17 \leq \text{s.limits.stack\_limit} \wedge \\ \quad \text{size\_of\_heap s} + 11 \leq \text{s.limits.heap\_limit} \wedge \ldots \Rightarrow \\ \qquad \text{(snd (evaluate(Seq (Call} \ldots\text{) (Call} \ldots\text{)))).safe\_for\_space}\end{array}$$

This shows that as long as there are 11 words (88 bytes) of heap and 17 words (136 bytes) of stack left when calling printLoop, we will not run out of memory. (We are compiling to a 64-bit architecture, thus machine words are 8 bytes long.)

The formal proof closely resembles our earlier informal argument, but the details of the formal proof are omitted here.

The resulting is_safe_for_space theorem for the entire yes program is:

$$\vdash \text{is\_safe\_for\_space } \mathit{ffi} \text{ yes\_x64\_conf yes\_prog } (56{,}89)$$

Here, the 56 and 89 are the concrete stack and heap bounds measured in machine words. These bounds are obtained during the course of the proof. They are larger than the bounds for the call to printLoop because the surrounding program (e.g. standard library) allocates on the execution up to the point of the call to printLoop.

Having established that our program satisfies is_safe_for_space, a similar top-level correctness theorem, to the one shown in Section 6, can be instantiated to read:

$$\begin{array}{l}\vdash 56 \leq \mathit{stack\_limit} \wedge 89 \leq \mathit{heap\_limit} \wedge \\ \quad \text{read\_limits yes\_x64\_conf } \mathit{ms} = (\mathit{stack\_limit}, \mathit{heap\_limit}) \wedge \\ \quad \text{installed yes\_code} \ldots \mathit{ms} \wedge \ldots \Rightarrow \\ \qquad \text{machine\_sem} \ldots \mathit{ms} = \text{semantics\_prog} \ldots \text{yes\_prog}\end{array}$$

The equality in the theorem above allows us to carry over any liveness property from the source semantics into the machine code semantics.

For our example, we can prove that the `yes` source-level program will produce an infinite stream of "y" characters on `stdout`.

$$\begin{array}{l}\text{semantics\_prog} \ldots \text{yes\_prog} = \\ \quad \{\text{Diverge (lrepeat [put\_str\_event "y"])}\}\end{array}$$

Such a theorem is easy to establish thanks to a program logic for non-terminating CakeML programs [Åman Pohjola et al. 2019], where proving this liveness property for the main loop is a 15-line proof. Unfolding the abstractions of the program logic to obtain a corresponding theorem about the CakeML semantics requires some additional boilerplate.

Finally, we combine these two theorems from above to obtain the same liveness property at the level of the compiler-generated machine code:

$$\begin{array}{l}\vdash 56 \leq \mathit{stack\_limit} \wedge 89 \leq \mathit{heap\_limit} \wedge \\ \quad \text{read\_limits yes\_x64\_conf } \mathit{ms} = (\mathit{stack\_limit}, \mathit{heap\_limit}) \wedge \\ \quad \text{installed yes\_code} \ldots \mathit{ms} \wedge \ldots \Rightarrow \\ \qquad \text{machine\_sem} \ldots \mathit{ms} = \\ \qquad\quad \{\text{Diverge (lrepeat [put\_str\_event "y"])}\}\end{array}$$

One can read this as saying: in a machine state ms where there are 56 words of stack and 89 words of heap available, and where the compiler output yes_code is installed and ready to run, execution from ms can exhibit one and only one behaviour: it will produce an infinite stream of "y" on `stdout`. In this case, the theorem is about x86-64 machine code. Since our cost semantics is not tied to a particular architecture, the same result could be reproduced for e.g. ARMv8 or RISC-V with no change to the space cost reasoning.

7.2 A Higher-Order Example

Higher-order functions are convenient abstractions that encapsulate common programming patterns in a generic and re-usable form. When it comes to space cost, higher-order functions provide "familiar" patterns of heap and stack consumption which can be described in terms of their functional arguments, simplifying reasoning considerably. Moreover, since the program structure remains constant proof patterns can be mostly reused from one specific instance to another.

7.2.1 `all` *Is Safe.* As an example of how higher-order functions can make space reasoning simpler, consider the CakeML function `all` in Figure 5 which uses the higher-order function `foldl` to compute the conjunction of all (boolean) elements in a list.

```
fun all l = foldl (fn a b => a andalso b) true l
```

Fig. 5. Implementation of `all`

We are interested in describing how a call to `all` affects space consumption. First, we look at the general structure of `foldl` (Section 3.2) and note that (1) its definition is tail recursive, thus no stack should be consumed aside from that required to execute `(f e x)`, and that (2) the only possible sources of heap growth are the sizes of the resulting values of `(f e x)` (the updated initial value) and any heap space required to execute `f` itself. It is then clear that the space consumption of `foldl` depends closely on that of its closure argument `f`.

Given this insight on `foldl` we can intuitively understand the space consumption of `all` by examining the closure argument provided to `foldl`: `andalso` takes two boolean arguments and computes their conjunction; under reasonable assumptions, we may conclude that:

- no stack should be consumed given that `andalso` can be defined using branching primitives,
- the size of the resulting value is constant since all boolean values have the same size, and
- no heap allocation is required inside `andalso`.

We can then informally conclude that a call to `all` should not use the heap or grow the stack. Thus, if a program's space consumption is within the available limits before calling `all`, it should remain within the limits after the function returns.

7.2.2 `all` *Is Safe, Formally!* To formalise our previous conclusion we show that a call to `all` satisfies is_safe_for_space in a manner similar to the one presented in Section 7.1.2.

```
andalso [0; 1] ≝
  if_var 1 (return 0)
  do 3 := Cons false_tag []
    return 3
  od
```

Fig. 6. DataLang implementation of `andalso`

The DataLang implementation of `andalso` (Figure 6) contains only a branching primitive (if_var), two return statements, and a Cons operation with empty payload; none of these actions consume heap, nor do they require the creation of a stack frame. Thus, our initial intuition about `andalso`'s space consumption is validated. The DataLang implementation of `foldl` (Section 3.2) is tail-recursive and only contains operations that either perform a boolean check (TagLenEq,EqualInt) or access an argument variable (ElemAt); thus, the closure call (lines 10 and 13) is the only place where more heap or stack might be required.

Since `all` is defined as a direct call to `foldl`, it is sufficient to show safe_for_space preservation for the body of `foldl` in an environment with the appropriate arguments; this is formally expressed by the following lemma:

$$\begin{array}{l}
s.\text{safe\_for\_space} \land \\
s.\text{locals} = \\
\;\{\; 0 \longrightarrow \; vl; \\
\quad 1 \longrightarrow \; \text{Block } 0 \; true\_tag \; []; \\
\quad 2 \longrightarrow \; \text{Block } ts\_f \; tag\_f \;\; [\text{CodePtr and\_loc};\; \text{Number } 1]\} \land \\
\text{repbool\_list } vl \; n \; ts \land \text{repbool\_safe\_heap } s \; vl \land \\
\text{the (size\_of\_stack } s.\text{stack}) + \text{the } s.\text{locals\_size} < s.\text{limits.stack\_limit} \land \\
\text{size\_of\_heap } s \leq s.\text{limits.heap\_limit} \land \ldots \Rightarrow \\
\;\; (\text{snd (foldl\_body } s)).\text{safe\_for\_space}
\end{array}$$

The assumptions on this lemma ensure the state of the program is safe_for_space and contains the appropriate arguments. The stack frame s.locals should contain the three initial arguments to `foldl` with concrete values for the closure (`andalso`) and initial value (`true`). repbool_list checks that vl is an appropriate DataLang representation of a list of booleans (similar to the one presented in Section 3.1) with length n and timestamps bounded by ts. repbool_safe_heap expresses timestamp unicity: it states that if a Block in the heap has the same timestamp as a Block in vl, then they are the same block. Finally, the last two assumptions state that the current measurements of heap and stack are within the space limits; these assumptions can be tweaked to accommodate any extra space needed to safely perform the evaluation.

The proof that is by induction on n—the length of the list represented by vl—where foldl_body is the RHS in the DataLang definition of `foldl`. The base case ($n = 0$) is trivial as it implies vl represents the empty list (Block 0 0 []). For the inductive case most of the program can be symbolically evaluated since initial variables have a concrete value (from s.locals) or in the case of vl a well defined structure (from repbool_list); the program performs the closure call in line 10 (as we have the correct arity) and it does not increase the stack or the heap usage. Lastly, the inductive hypothesis is applied over the tail-recursive call to `foldl` (line 17); all assumptions can be re-established by evaluation except for the bound on size_of_heap. This requires showing that the heap size at the recursive call is less than or equal to the initial size, which holds because the arguments are becoming smaller. Note that this exercises the way size_of infers aliasing information from timestamps: if our analysis failed to account for the structure sharing between the lists `x::xs` and `xs` and did not distinguish live memory from garbage, we would be forced to conclude that `foldl` uses $O(|\texttt{cs}|^2)$ heap space.

The resulting theorem can be used to prove concrete space bounds of a program using `all`, e.g. we have proven that a program that directly calls `all` with list `[true,false,true,true,false]` consumes 56 words of stack and 78 words of heap.

7.2.3 From `all` *to* `sum`. The CakeML function `sum` in Figure 7 adds all integers in a list and is defined using `foldl` much like `all`.

```
fun sum l = foldl (fn a b => a + b) 0 l
```

Fig. 7. Implementation of sum

Proving safe_for_space for sum brings a new challenge: its space conumption depends on the size of the (bignum) integers in the list and the largest integer (magnitude) encountered during summation. The key differences compared with all are highlighted with a $\boxed{\text{box}}$ below. Variables 1 and 2 are updated to the new initial value and function closure; repint_list and repint_safe_heap are again predicates over vl but for an integer list instead of booleans. Changes to the heap and stack bounds were required since unlike andalso, integer addition might consume heap and stack space as it can involve bignum operations; sum_stack_size is the maximum stack size any of the additions performed will require, while sum_heap_size return the maximum heap usage; biggest_acc_size computes the amount of extra heap required to store the biggest accumulator value of the whole execution. Moreover, biggest_num_size is the maximum size of a number in vl, which is required since the value obtained in line 6 of foldl (Section 3.2) might be counted twice due to aliasing.

$$
\begin{array}{l}
s.\mathsf{safe\_for\_space} \wedge \\
s.\mathsf{locals} = \\
\{\ 0 \longrightarrow\ vl; \\
\quad 1 \longrightarrow\ \boxed{\mathsf{Number}\ 0}; \\
\quad 2 \longrightarrow\ \mathsf{Block}\ ts\_f\ tag\_f\ \ [\mathsf{CodePtr}\ \boxed{\mathsf{add\_loc}};\ \mathsf{Number}\ 1]\} \wedge \\
\boxed{\mathsf{repint\_list}}\ vl\ n\ ts \wedge \boxed{\mathsf{repint\_safe\_heap}}\ s\ vl \wedge \\
\mathsf{the}\ (\mathsf{size\_of\_stack}\ s.\mathsf{stack}) + \mathsf{the}\ s.\mathsf{locals\_size} + \boxed{\mathsf{sum\_stack\_size}\ s\ 0\ vl} \\
\quad < s.\mathsf{limits.stack\_limit} \wedge \\
\mathsf{size\_of\_heap}\ s\ + \\
\boxed{\mathsf{sum\_heap\_size}\ s\ 0\ vl +\ \mathsf{biggest\_acc\_size}\ s\ 0\ vl +\ \mathsf{biggest\_num\_size}\ s\ 0\ vl} \\
\quad \leq s.\mathsf{limits.heap\_limit} \wedge \ldots \Rightarrow \\
\quad (\mathsf{snd}\ (\mathsf{foldl\_body}\ s)).\mathsf{safe\_for\_space}
\end{array}
$$

In summary, all adjustments required for sum are in relation to the closure argument and its particular space consumption, thus aside from calling + instead of andalso and the extra space required, the proofs for all and sum share many of the same tactics and techniques. To show an instantiation, we have proven that a program that directly calls sum with list $[1, 2, 3, 4, 10^{64}]$ consumes 56 words of stack and 108 words of heap.

7.3 Other Examples

7.3.1 Generating Pseudorandom Numbers. A *linear congruential generators* (LCG) is a kind of pseudorandom number generator. The basic idea is that if x_i is the current element of the pseudorandom number sequence, the next element is generated by the equation $x_{i+1} = (ax_i + c) \bmod m$ for fixed values of a, c, m. We implemented a program that produces an infinite stream of LCG-generated numbers on stdout. The source code is shown in Figure 8.

The LCG shares structural similarities with the earlier examples but differs in several key ways that have bearing on space-cost reasoning. First, it exercises more language features and reasoning techniques, including truly nested recursive function calls. In particular, lcgLoop recursively calls n2l_acc which tail-recursively constructs a list in accumulator passing style. Second, the LCG features more comprehensive use of arithmetic. Arithmetic over small numbers has no stack or heap cost. However, once the numbers are large enough, arithmetic starts to incur the stack and heap costs of invoking the bignum library. The stack cost for bignum operations is not dependent

```
fun n2l_acc n acc =
  if n < 10 then hex n :: acc
  else n2l_acc (n div 10)
                (hex (n mod 10) :: acc)

fun num_to_string n =
  n2l_acc n [#"\n"]

fun put_chars cs =
  case cs of [] => ()
  | x::xs => (put_char x ; put_chars xs)

fun print_num n =
  put_chars (num_to_string n)
```

```
fun lcg a c m x =
  (a * x + c) mod m

fun lcgLoop a c m x =
  let
    val x1 = lcg a c m x
    val u = print_num x1
  in
    lcgLoop a c m x1
  end

val _ = lcgLoop 8121 28411 134456 42
```

Fig. 8. Implementation of lcg. The definition of put_char is elided.

on the size of the given integers, but the heap cost of course depends on how large the numbers are. Note that for programs that only use small numbers, one has to prove that the numbers stay small enough to avoid the cost of bignum operations.

We have proved the code shown in Figure 8 to be safe for space (with stack bound 182 and heap bound 199). We proved this by showing that the code stays within the range of small enough integers to avoid triggering CakeML's bignum library. Our proof is largely agnostic to the precise values of the parameters so, in fact, lcgLoop can be called with different values of a, c, m, x with almost no change to the proofs (as long as the bounds described above are met).

7.3.2 List Reverse. In this example, we illustrate the precision advantages we gain by expressing the cost semantics in an intermediate language. Consider the following naive implementation of list reverse using list append (written here in SML syntax: @).

```
fun reverse [] = []
  | reverse(f::l) = reverse l @ [f]
```

Fig. 9. Naive implementation of reverse.

An informal source-level cost analysis would force us to conclude that since this function is not tail-recursive, it requires $O(n)$ stack space, where n is the length of the input list, to accommodate the stack frames of the n recursive calls reverse makes.

However, the CakeML compiler performs tail-call introduction before it reaches DataLang [Abrahamsson and Myreen 2017], and this optimisation triggers on the body of reverse. In other words, the compiler produces essentially the same code for reverse as it does for reverse' below:

```
fun reverse'_aux [] acc = acc
  | reverse'_aux (f::r) acc = reverse'_aux r (f::acc)

fun reverse' l = reverse'_aux l []
```

Fig. 10. Tail-recursive implementation of reverse.

Therefore, we can use our cost semantics to prove that our initial naive version of `reverse` uses only a constant amount of stack space:

$$\begin{array}{l}\vdash \mathsf{evaluate}\ (s, \mathsf{reverse\_body}) = (res, s') \land \ \ldots \ \Rightarrow \\ \quad \exists\, k.\ s'.\mathsf{stack\_max} = \mathsf{option\_map}\ (+\ k)\ s.\mathsf{stack\_max}\end{array}$$

We note that a source-level cost semantics would have to know exactly when the tail-call introduction optimisation kicks in to be able to prove such a property for `reverse`.

The stack costs are concrete enough that we could prove a theorem similar to the one above with a precise numeric value in place of k, and we could additionally consider heap cost to prove that `reverse` is safe for space. However, that is not the point here: our cost semantics is modular enough that when we are only interested in stack usage, we can reason about it separately by considering only stack_max and ignoring safe_for_space. This results in a simpler proof than the previous examples because we do not need to reason about heap usage at all.

7.4 Lessons Learned

The above examples illustrate the power and precision of our cost semantics, but they also hint at scalability challenges that we have yet to address. Each of the examples took us on the order of 8-12 person-days to complete, except `reverse` which took around 4 person-hours. The proof scripts are often 1000 lines or more, which is not unheard of in verification, but can certainly be improved. This is all for programs no longer than ~20 lines of CakeML; clearly, for longer programs to be viable, we need better methods that achieve higher developer productivity.

Broadly speaking, the most often used proof method is to symbolically evaluate the cost semantics, proving after each step that safe_for_space is preserved. A particularly tedious recurring challenge is proving size_of_heap inequalities between heaps with slightly different ordering or block nesting structure. For example, after a program destructs a non-empty list, the local variables will contain separate references to the head and tail instead of a single reference to the whole list. Since the new variables refer only to pre-existing heap elements, clearly the heap cannot have grown as a result of this operation. Unfortunately, this does not follow immediately from the definition of size_of: blocks nested within other blocks do not count if their parent block has been seen already, so we must first prove that if the whole list has been seen, then so has its head and tail. This requires additional invariants throughout the proof.

We believe such roadblocks could be smoothed considerably by developing tactics for automatic rewriting of heaps modulo AC and nesting, allowing common parts to be factored out and repeating parts to be eliminated. As an initial step towards this goal, we have proved that under certain natural well-formedness assumptions about the heap, size_of is invariant under list reordering.

Another way to improve scalability is by proving generic space safety theorems for standard library functions. As showcased in Section 7.2.3, similar programs can reuse parts of the proof structure, which greatly simplifies reasoning. However, this is currently done as a copy-paste recipe rather than as a single reusable tactic or theorem. It would be more satisfying to prove a generic result about `foldl` where we can obtain a bound by plugging in the cost of its function argument. We believe this is doable, at least for pure functions with simple heap allocation patterns. The main difficulty is that symbolic evaluation of DataLang programs tends to require very specific assumptions about the global state, so when we abstract over the function argument, we need to retain a concrete enough state to support symbolic evaluation.

8 RELATED WORK

There has been much interest in defining cost semantics for both imperative and functional programming languages. The main types of resources considered are *execution time* and *memory* *usage*, and the cost semantics aim to estimate worst-case bounds for these resources either at the source level or during transformation phases through compilers.

Source-level Cost Analysis. Source-level techniques enable static cost analysis. For instance, Hofmann and Jost [2003] provide static prediction of heap space usage for functional programs, and Jost et al. [2017] develop a type system with heap annotations for determining the execution costs of lazily evaluated functional languages. RelCost [Çiçek et al. 2017], CostIt [Çiçek et al. 2015], and RaML [Hoffmann et al. 2012] are resource-aware type systems for source-level programs based on refinement types. Guéneau et al. [2018] provide worst-case asymptotic time complexity of higher-order imperative programs. Wang et al. [2017] present an ML-like functional language with time-complexity annotations in indexed types. Handley et al. [2020] implement a system based on refinement types to enable reasoning about resource usage of pure Haskell programs in Liquid Haskell. Aspinall et al. [2007] develop a program logic for proving statements about resource consumption for the Java Virtual Machine Language (JVML), Atkey [2010] formalises a separation logic for heap-resource analysis within the Coq proof assistant, and Vasconcelos [2008] uses sized types to obtain upper bounds on dynamic space usage of functional programs. While these source-level analysis techniques provide formal estimates of cost, they ignore the effect of compilation and program transformation on resource consumption, leaving an inherent trust gap between the analysis and the actual machine code that runs.

Preservation of Resource Bounds through Compilation. Resource bounds estimated at the source level can be made accurate and certified by proving their preservation throughout the compilation chain. Crary and Weirich [2000] estimate upper bounds on resources through a decidable type system and a bounds-certifying compiler from the impure functional language PopCron to typed assembly. Resources are modelled as semantic clocks, and a resource-safe program is one for which the clock never expires. While this approach is best suited to modelling time (where resource usage is monotonic), it does in principle generalise to stack and heap usage because there is a mechanism to recover spent resources, provided allocation and deallocation is explicit in the program text. Since this assumption fails to hold in the presence of garbage collection, their approach is not well suited to languages with automatic memory management.

Paraskevopoulou and Appel [2019] develop a cost model for the CPS lambda-calculus, in which they derive time and space bounds for a closure conversion compilation phase in the Coq proof assistant. In our work we do not need to explicitly model the space cost of closure conversion; instead, we derive space bounds on code that has already been closure-converted. Their work is also notable for taking garbage collection into account: their measure of space usage assumes that an ideal, complete garbage collector is invoked often enough so that actual heap usage can only exceed the size of the reachable heap by a bounded amount. They can also give bounds for diverging programs. The heap is explicitly present in the memory models of their source and target languages. In contrast, we are able to lift our cost analysis to a level of abstraction where there is no notion of heap, by annotating values in the variable store with timestamps. Unlike Paraskevopoulou and Appel, we cash out our cost model using the completeness proofs for the real garbage collector implementation. Their runtime is stack-less, which allows them to sidestep the problem of finding roots in the stack. CakeML maintains its own stack, and so implements and verifies such root-finding. Finally, our work is fully integrated into an end-to-end verified compiler, allowing space bounds to be leveraged to transfer liveness properties all the way from source to machine code; theirs is not (yet).

Our technique for estimating stack space consumption through an end-to-end compiler is closely related to the CerCo project [Amadio et al. 2014]. The CerCo project has built a verified C compiler producing object binaries for the 8051 microcontroller in the Matita theorem prover. The compiler precisely estimates the non-asymptotic computational cost involving execution time and stack space usage of input programs at the source level. It also generates source-level annotations that correctly model low level costs. These invariants are then certified through automated theorem provers. Case studies include certifying the exact reaction time of Lustre dataflow programs compiled to C. While the CerCo project inspired our work on stack bounds, it does not consider heap usage, let alone garbage collection. Their compiler correctness proof only considers preservation of cost bounds and not functional correctness, whereas the CakeML compiler with our extensions considers both.

The CompCert compiler [Leroy 2009] has also been used to obtain formal resource bounds for C programs. Carbonneaux et al. [2014] develop a logic for reasoning at the source level about stack space consumption of the corresponding CompCert compiler output. They introduce resource consumption events to CompCert that are preserved by compilation and use the compiler itself to determine the actual size of stack frames. Besson *et. al* introduce finite memory and integer pointers to the memory model of CompCert, extend CompCert's front-end for this concrete memory model, and continue to verify its back-end layers to develop CompCertS in Coq [Besson et al. 2014, 2015, 2019]. CompCertS estimates the memory usage of individual functions directly at the C level, proves that compiled programs use no more memory than source programs, and ensures that the absence of memory overflow is preserved by compilation. It also provides stronger guarantees about arbitrary pointer arithmetic and avoids the miscompilation of programs performing bit-level pointer manipulation. Wang et al. [2019] enriches the memory model of CompCert with an abstract and bounded stack to develop Stack-Aware CompCertX, an extension of CompCert with compositional compilation. The main distinction between our work and these is the level of abstraction at which the cost semantics is expressed. In this respect, C is very similar to our WordLang: both languages give the programmer an explicit view of the heap and responsibility for managing heap memory, while abstracting the stack. We express our cost semantics in a language that abstracts away from the heap and features no explicit memory management.

9 CONCLUSION

We have presented a space cost semantics for CakeML programs that makes it possible to prove the absence of out-of-memory errors in the generated machine code. The semantics does so by estimating the resource usage of programs at an intermediate representation that avoids reasoning about pointers and heap objects, yet takes aliasing of data elements into account for an accurate estimate. The cost analysis is proven sound down to the machine code, and we have demonstrated that it can be used to carry source-level liveness properties down to machine code: the space analysis rules out all partiality induced by potential out-of-memory errors, and can be applied even to programs that make unboundedly many heap allocations.

In this paper, our primary goal was to make sound space cost reasoning about CakeML programs possible. What remains to show is how such reasoning can be made scalable; while our examples do exhibit interesting and relevant features like non-termination and unbounded allocation, they are admittedly small. There are several interesting ideas to explore in this direction. One is to use coarser overapproximations of the heap size metric to make analysis more compositional. Another is to develop a framework of sound abstractions of the monadic DataLang semantics.

ACKNOWLEDGMENTS

We thank Thomas Sewell for help with proofs. This research was supported with funding from the Swedish Foundation for Strategic Research; Wallenberg AI, Autonomous Systems and Software Program, Sweden; and the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

REFERENCES

Oskar Abrahamsson and Magnus O. Myreen. 2017. Automatically Introducing Tail Recursion in CakeML. In *Trends in Functional Programming - 18th International Symposium, TFP 2017, Canterbury, UK, June 19-21, 2017, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10788)*, Meng Wang and Scott Owens (Eds.). Springer, 118–134. https://doi.org/10.1007/978-3-319-89719-6_7

Roberto M. Amadio, Nicolas Ayache, Francois Bobot, Jaap P. Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, Randy Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, Ian Stark, and Paolo Tranquilli. 2014. Certified Complexity (CerCo). In *Foundational and Practical Aspects of Resource Analysis*, Ugo Dal Lago and Ricardo Peña (Eds.). Springer International Publishing, Cham, 1–18.

Johannes Åman Pohjola, Henrik Rostedt, and Magnus O. Myreen. 2019. Characteristic Formulae for Liveness Properties of Non-terminating CakeML Programs. In *Interactive Theorem Proving (ITP)*. LIPICS.

David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. 2007. A program logic for resources. *Theoretical Computer Science* 389, 3 (2007), 411 – 445.

Robert Atkey. 2010. Amortised Resource Analysis with Separation Logic. In *Programming Languages and Systems*, Andrew D. Gordon (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 85–103.

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2014. A Precise and Abstract Memory Model for C Using Symbolic Values. In *Programming Languages and Systems*, Jacques Garrigue (Ed.). Springer International Publishing, Cham, 449–468.

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2015. A Concrete Memory Model for CompCert. In *Interactive Theorem Proving*. Springer International Publishing, Cham, 67–83.

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2019. CompCertS: A Memory-Aware Verified C Compiler Using a Pointer as Integer Semantics. *Journal of Automated Reasoning* 63, 2 (01 Aug 2019), 369–392.

Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. 2014. End-to-end Verification of Stack-space Bounds for C Programs. *SIGPLAN Not.* 49, 6 (June 2014), 270–281.

Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational Cost Analysis. *SIGPLAN Not.* 52, 1 (Jan. 2017), 316–329.

Ezgi Çiçek, Deepak Garg, and Umut Acar. 2015. Refinement Types for Incremental Computational Complexity. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 406–431.

Karl Crary and Stephnie Weirich. 2000. Resource Bound Certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Boston, MA, USA) *(POPL '00)*. ACM, 184–198.

Armaël Guéneau, Arthur Charguéraud, and François Pottier. 2018. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification. In *ESOP 2018 - 27th European Symposium on Programming (LNCS - Lecture Notes in Computer Science, Vol. 10801)*. Springer.

Martin Adam Thomas Handley, Niki Vazou, and Graham Hutton. 2020. Liquidate Your Assets: Reasoning About Resource Usage in Liquid Haskell. In *Principles of Programming Languages (POPL)*. to appear.

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource Aware ML. In *Proceedings of the 24th International Conference on Computer Aided Verification* (Berkeley, CA) *(CAV'12)*. Springer-Verlag, Berlin, Heidelberg, 781–786.

Martin Hofmann and Steffen Jost. 2003. Static Prediction of Heap Space Usage for First-order Functional Programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. ACM, New York, NY, USA, 185–197.

Steffen Jost, Pedro Vasconcelos, Mário Florido, and Kevin Hammond. 2017. Type-Based Cost Analysis for Lazy Functional Languages. *Journal of Automated Reasoning* 59, 1 (01 Jun 2017), 87–120.

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Communications of the ACM* 52, 7 (2009). https://doi.org/10.1145/1538788.1538814

Magnus O. Myreen. 2010. Reusable Verification of a Copying Collector. In *Verified Software: Theories, Tools, Experiments (VSTTE) (Lecture Notes in Computer Science, Vol. 6217)*, Gary T. Leavens, Peter W. O'Hearn, and Sriram K. Rajamani (Eds.). Springer. https://doi.org/10.1007/978-3-642-15057-9

Magnus O. Myreen and Gregorio Curello. 2013. Proof Pearl: A Verified Bignum Implementation in x86-64 Machine Code. In *Certified Programs and Proofs (CPP)*, Georges Gonthier and Michael Norrish (Eds.). Springer, 66–81.

Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science)*, Peter Thiemann (Ed.). Springer, 589–615.

Scott Owens, Michael Norrish, Ramana Kumar, Magnus O. Myreen, and Yong Kiam Tan. 2017. Verifying Efficient Function Calls in CakeML. *Proc. ACM Program. Lang.* 1, ICFP, Article 18 (Sept. 2017), 27 pages.

Zoe Paraskevopoulou and Andrew W. Appel. 2019. Closure Conversion is Safe for Space. *Proc. ACM Program. Lang.* 3, ICFP, Article 83 (July 2019), 29 pages.

Adam Sandberg Ericsson, Magnus O. Myreen, and Johannes Åman Pohjola. 2019. A Verified Generational Garbage Collector for CakeML. *J. Autom. Reasoning* 63, 2 (2019), 463–488. https://doi.org/10.1007/s10817-018-9487-z

Konrad Slind and Michael Norrish. 2008. A Brief Overview of HOL4. In *Theorem Proving in Higher Order Logics (TPHOLs)*.

Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2019. The verified CakeML compiler backend. *Journal of Functional Programming* 29 (2019).

Pedro B Vasconcelos. 2008. *Space Cost Analysis Using Sized Types.* Ph.D. Dissertation. University of St. Andrews.

Peng Wang, Di Wang, and Adam Chlipala. 2017. TiML: A Functional Language for Practical Complexity Analysis with Invariants. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 79 (Oct. 2017), 26 pages.

Yuting Wang, Pierre Wilke, and Zhong Shao. 2019. An Abstract Stack Based Approach to Verified Compositional Compilation to Machine Code. *Proc. ACM Program. Lang.* 3, POPL, Article 62 (Jan. 2019), 30 pages.