A flat reachability-based measure for CakeML's cost semantics

Alejandro Gomez-Londoño
alejandro.gomez@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

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

ABSTRACT

The CakeML project has recently developed a verified cost semantics that allows space cost reasoning of CakeML programs. With this space cost semantics, compiled machine code can be proven to have tight memory bounds ensuring no out-of-memory errors occur during execution. This paper proposes a new cost semantics which is designed to make proofs about space costs significantly simpler than they were with the original version. The work described here has been developed in the HOL4 theorem prover.

CCS CONCEPTS

• **Software and its engineering** → **Formal software verification**; **Compilers**.

KEYWORDS

compiler verification, cost semantics, space usage

ACM Reference Format:

Alejandro Gomez-Londoño and Magnus O. Myreen. 2018. A flat reachability-based measure for CakeML's cost semantics. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Functional languages present programmers with many abstractions (e.g., polymorphism, garbage collection, ADTs, among others) to aid them in the development of complex programs. However, these features often come at the cost of an increase in memory usage and an unclear space usage; making it difficult to judge whether a program has enough memory to run.

The CakeML project has recently developed a verified cost semantics [5] which, at its core, uses a measuring function embedded in the semantics to determine if, at any given point, a program has run out of memory. The cost semantics has been proved sound, which means machine code generated by the CakeML compiler will never produce out-of-memory errors if the CakeML cost semantics has ruled them out. Unforunately, reasoning using the the original cost semantics for CakeML requires considerable effort.

This paper improves on the original cost semantics. This paper defines an alternative space measuring function, which is defined in stages: it first computes the set of reachable nodes, and then computes the sum of the size of the data at those nodes. In contrast, the old space measuring function did everything at once: it discovered the reachable part of the heap through recursive descent. The new formulation is expected to lead to tidier proofs than the previous measuring function. Our initial experiments suggest that proofs about space cost are significantly more manageable with the new formulation.

This paper makes the following contributions:

- In this paper, we define a new reachability-based measuring function (Section 3), which is designed to be significantly simpler to work with than the original (Section 2.3).
- We demonstrate (in Section 4) how the new formulation overcomes some of the most significant problems of the original formulation.
- Finally, we discuss (in Section 5) ways in which the new formulation of the space cost semantics can be proved sound so that future space cost proofs can use the new formulation.

All of this work is machined-checked using the HOL4 theorem in the context of the CakeML compiler verification project. It should be noted that this work is currently in an early stage of development, but of sufficient maturity to be presented at IFL. We are confident that the main result of this paper can be fully completed in time for the post-proceedings submission.

2 A VERIFIED COST SEMANTICS

The cost semantics for the CakeML compiler [5] is expressed at the level of its DataLang intermediate language.

DataLang is an intermediate language approximately in the middle of the CakeML compiler. It is an imperative intermediate language with nested tuple-like values and reference pointers, but no function values. It appears right before memory becomes finite and the garbage collector is introduced. The semantics of DataLang is expressed in the form of a (functional) big-step semantics.

The semantics for DataLang acts as a cost semantics for CakeML by maintaining a boolean-valued `safe_for_space` field in the semantic state of the operational semantics. This field is set to false whenever a semantic space cost measurement predicts that the current use of space might exceed the configured space limits for heap or stack space.

This paper focuses on the measurement of heap space. At each allocation of new memory, the semantics for DataLang computes the size of the currently live data using a measure called `size_of`. This `size_of` function computes the space consumption of all reachable values from the root values obtained from the stack and global variables. This `size_of` function is defined to carefully track

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

aliasing by keeping track of pointer-equal values, and is unchanged by garbage collection as it only consider live (reachable) data.

To prove the space safety of a CakeML program, one must show that for some limit—concrete or abstract—the semantics of its DataLang representation never sets `safe_for_safe` to `false`. Once space safety is established, it can be extended all the way to the level of machine code, thanks to the soundness proof for the cost semantics w.r.t. to the CakeML compiler.

The rest of this section introduces: the DataLang semantics, the space semantics, and the definition of the original heap space measure, i.e. `size_of`. More details on the original DataLang's operational and costs semantics can be found in prior work [5, 9].

2.1 DataLang at a glance

DataLang is an imperative language with abstract values, stateful storage of local variables, and a call stack. In the compiler-stack, it sits between the more abstract functional languages and the low-level languages with word-based value representations.

To give a sense of how CakeML programs look when compiled into DataLang consider the following CakeML function expressed in CakeML source syntax (which is very similar to SML syntax).

```
fun app123 x = let a = [1,2,3] in a ++ x end
```

This function prepends the list `[1,2,3]` to its input. The result of compiling this function to DataLang is shown in Figure 1.

```
line 0 :   app123 [0] evaluates as
line 1 :     MakeSpace 9
line 2 :     1 := Cons nil_tag []
line 3 :     2 := Const 3
line 4 :     3 := Cons cons_tag [2; 1]
line 5 :     4 := Const 2
line 6 :     5 := Cons cons_tag [4; 3]
line 7 :     6 := Const 1
line 8 :     7 := Cons cons_tag [6; 5]
line 9 :     8 := ListAppend [7; 0]
line 10 :    return 8
```

Figure 1: DataLang code for a function that prepends `[1,2,3]` to its argument

At first, the DataLang presentation of the code might seem significantly different. However, on closer inspection, we hope the reader will see the similarity. In DataLang primitive operation are always assigned (:=) to a local variables, which are represented as natural numbers. On line 0, argument 0 corresponds to the source code binding `x`. Line 1 allocates 9 slots of space (that's 3 slots per `Cons`). Line 2 creates a value representing an empty list using the primitive operation `Cons` and a number tag (`nil_tag`) denoting the `nil` constructor for lists. In line 3 a `Const` operation creates the number literal 3. Line 4 combines local variables 1 (`[]`) and 2 (3) into the singleton list `[3]` using `Cons` and the corresponding list constructor tag (`cons_tag`); using the same process, lines 5 through 8 create the DataLang representation of the list `[1,2,3]`. Then, Line 9 applies `ListAppend`—which appends the two lists-shaped values—variables 0 (the argument) and 7 (`[1,2,3]`).

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

Figure 2: DataLang's abstract values

Primitive values in DataLang are modeled by the data type presented in Figure 2. `Number` is 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 arrays). The `Block` constructor represents contiguous values in memory, and encodes datatype constructors, tuples and vectors.

An example of a DataLang value is shown in Figure 3 which shows the DataLang. This is a value that can result from a call to `app123` with the empty list as the argument. `Block` values, with `cons_tag` and `nil_tag` indicate the source-level constructor that each Block represents. Furthermore, timestamp values 8, 7, and 6 uniquely identify each block.

```
app123_nil ≝ Block 8 cons_tag [Number 1;
               Block 7 cons_tag [Number 2;
                 Block 6 cons_tag [Number 3;
                   Block 0 nil_tag []]]]
```

Figure 3: `Block` representation of CakeML list `[1,2,3]`

The semantics state is defined as the record type shown in Figure 4. 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. The `global` field contains an optional reference to an array of global variables. Space limits are kept in a record with fields for heap and stack limits. The boolean flag safe_for_space is set to false when space limits have been exceeded. The remaining fields are not of relevance for the presentation here.

The semantics of DataLang is defined as a functional big-step semantics [8]. In this style of semantics, a clocked big-step evaluation function, `evaluate`, takes a (program, state) pair as input, and returns a (result, state) pair as output. As an example, consider the evaluation of `app123` with the empty list as argument, which results in value `app123_nil`. Note that the program is given to `evaluate` as a DataLang AST (`app123_prog`) and arguments are local variables in the state.

```
evaluate (app123_prog,
          s with locals := { 0 ↦ Block 0 nil_tag [] })
  = (app123_nil, s′)
```

To better visualize intermediate steps of evaluations, the DataLang semantics can also be expressed as a shallowly embedded state-exception monad. This is the representation used in `app123` which can partially evaluate the first three operations by unfolding bind applications:

```
α state = <|
  locals : v num_map;
  refs : v ref num_map;
  stack : stack list;
  global : num option;
  limits : limits;
  safe_for_space : bool;
  clock : num;
  ...
  |>

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

limits = <|
  heap_limit : num;
  stack_limit : num;
  ...
  |>
```

Figure 4: The definition of the DataLang state.

```
app123 (s with locals := { 0 ↦ Block 0 nil_tag [] })
 = ( 4 := Const 2
     5 := Cons cons_tag [4; 3]
     6 := Const 1
     7 := Cons cons_tag [6; 5]
     8 := ListAppend [7; 0]
     return 8)
    s with <| locals := { 0 ↦ Block 0 nil_tag []
                          1 ↦ Block 0 nil_tag []
                          2 ↦ Number 3
                          3 ↦ Block 3 cons_tag
                                [Number 3;
                                 Block 0 nil_tag []] };
              ...
           |>
```

2.2 Embedded cost semantics

As previously stated, DataLang's costs semantics is embedded into its operational semantics. Therefore, proving space safety of app123 is a matter of proving the following statement:

$$\begin{aligned}\vdash\ & s.\text{limits.heap\_limit} = mh \wedge \\ & s.\text{limits.stack\_limit} = ms \wedge \\ & s.\text{safe\_for\_space} \wedge \\ & \text{evaluate}\ (\text{app123\_prog}, s) = (res, s') \implies \\ & \quad s'.\text{safe\_for\_space}\end{aligned}$$

This is, given stack space mh and heap space ms; the evaluation of app123_prog preserves safe_for_space, thus signalling that the program's memory consumption falls within the given bounds.

Internally, the safe_for_space flag is updated at every space-consuming operation, for example, at function calls and whenever new values are created. Auxiliary functions size_of_heap and size_of_stack are used to update safe_for_space in one of two ways. If k slots of new heap space are to be used (e.g. as part of MakeSpace), then safe_for_space is updated as follows:

$$\begin{aligned}& s\ \text{with} \\ & \text{safe\_for\_space} := \\ & \quad (s.\text{safe\_for\_space} \wedge \\ & \quad \text{size\_of\_heap}\ s + k \leq s.\text{limits.heap\_limit})\end{aligned}$$

Similarly, if k slots of new stack space are to be comsumed (e.g. as part of a function call), then safe_for_space is updated as follows:

$$\begin{aligned}& s\ \text{with} \\ & \text{safe\_for\_space} := \\ & \quad (s.\text{safe\_for\_space} \wedge \\ & \quad \text{size\_of\_stack}\ s + k \leq s.\text{limits.stack\_limit})\end{aligned}$$

The important work is performed by the size_of_heap and size_of_stack functions. This paper focuses on improving the formulation of the heap space measure and thus size_of_heap.

The original formulation of size_of_heap is shown below. Here stack_to_vs is an auxiliary function that computes a list of root values from local variables (s. locals), the call-stack (extract_stack), and global references (global_to_vs). The root values are given to the measuring function size_of, which computes the size of heap elements reachable from these initial elements.

```
size_of_heap s =def
  let (n, _, _) =
      size_of (stack_to_vs s) s.refs LN in
    n

stack_to_vs s =def
  toList s.locals ++
  extract_stack s.stack ++
  global_to_vs s.global
```

The main workhorse of this definition is the size_of function, which is the topic of the next section.

2.3 The original heap measure: size_of

At the core of DataLang's cost semantics is the heap space measuring function size_of. This function is responsible for computing a space consumed by all values reachable from the given values. Figure 5 shows its definition with *seen* (a set of timestamps), and *refs* as additional arguments.

The measurement of most values (CodePtr, Word64, and Number) is straightforward, as it is either constant, already accounted for within another structure (e.g. stack frames), or measured by a function without considering other values. Whereas, the handling of Block and RefPtr values is more involve and where most of the complexity of size_of comes from. In the case of Block values, a set of already-measured (*seen*) timestamps is kept to avoid counting identical blocks multiple times; this mechanisms assumes a bijection between timestamps and the blocks in memory. For RefPtr, pointers are removed from references map (*refs*) once they are counted, this is to only follow a reference once.

The definition of size_of succeeds at providing tight bounds, mitigating the effects of aliasing, and traversing all live data; however, perhaps due to its precise and concrete nature, it can be challenging to reason about. The main hurdle with size_of is the linearity of its traversal, where initial measurements at the front

```
size_of [] refs seen ≝ (0, refs, seen)
size_of (x :: xs) refs seen ≝
  let (n1, refs1, seen1) = size_of xs refs seen ;
      (n2, refs2, seen2) = size_of [x] refs1 seen1 in
   (n1 + n2, refs2, seen2)
size_of [Word64 v0] refs seen ≝ (3, refs, seen)
size_of [Number i] refs seen ≝
  (if is_smallnum i then 0 else bignum_size i, refs, seen)
size_of [CodePtr v1] refs seen ≝ (0, refs, seen)
size_of [RefPtr r] refs seen ≝
  case lookup r refs of
   None ⇒ (0, refs, seen)
  | Some (ValueArray vs) ⇒
   (let (n, refs', seen') = size_of vs (delete r refs) seen
      in
     (n + |vs| + 1, refs', seen'))
  | Some (ByteArray v2 bs) ⇒
   (|bs| div (arch_size lims div 8) + 2, delete r refs, seen)
size_of [Block ts tag vs] refs seen ≝
  if vs = [] ∨ isSome (lookup ts seen) then (0, refs, seen)
  else
   let (n, refs', seen') =
        size_of vs refs (insert ts () seen) in
     (n + |vs| + 1, refs', seen')
```

Figure 5: Definition of the old `size_of`.

of the argument list directly affect subsequent ones through pointers or timestamps. Thus, conceptually simple properties (e.g. the reordering of values) are hard to prove and apply.

3 A FLAT REACHABILITY-BASED MEASUREMENT

This section shows the definition of our new heap cost measuring function, `flat_size_of`, which is to replace the original `size_of`. In a nutshell, `flat_size_of` takes a set of root addresses, computes the set of all addresses reachable from that initial set, and then sums the size of the heap element that is at one of the reachable addresses. We will go through the detail below.

First of all, DataLang has no immediate notion of heap address. For the purposes of the definition of `flat_size_of`, we define a type for DataLang addresses. An address is either a timestamp (`TStamp`) of a Block (remember each block has a unique timestamp) or a pointer to a reference (`RStamp`).

```
addr = TStamp num | RStamp num
```

Given a list of root values, we can compute, using `to_addr`, the root addresses. Note that this function does not recurse into the values inside `Block`, because it only wants to collect the immediate addresses of these values.

```
to_addr [] ≝ ∅
to_addr (Block ts v0 v1 :: xs) ≝ { TStamp ts } ∪ to_addr xs
to_addr (RefPtr ref :: xs) ≝ { RStamp ref } ∪ to_addr xs
```

The following value kinds do not have addresses in this representation. We will deal with these in a different way below.

```
to_addr (Number v4 :: xs) ≝ to_addr xs
to_addr (Word64 v5 :: xs) ≝ to_addr xs
```

Once the root addresses have been collected, we can neatly compute the set of all reachable addresses using the reflexive transitive closure (*) of a `next`-relation, which is defined further down.

```
reachable_v refs blocks roots ≝
  { y | ∃x. x ∈ roots ∧ (next refs blocks)* x y }
```

The `next`-relation relates an address a to the addresses that are one-step reachable from a. Here *blocks* is a mapping from timestamps to `Block` values.

```
next refs blocks (TStamp ts) r ≝
  r ∈ block_to_addrs blocks ts
next refs blocks (RStamp ref) r ≝
  r ∈ ptr_to_roots refs ref

block_to_addrs blocks ts ≝
  case lookup ts blocks of
  | Some (Block _ _ vs) ⇒ to_addr vs
  | _ ⇒ ∅

ptr_to_roots refs p ≝
  case lookup p refs of
  | Some (ValueArray vs) ⇒ to_addr vs
  | _ ⇒ ∅
```

With these functions we can state the set of all reachable addresses as follows.

```
reachable_v refs blocks (to_addr roots)
```

Next we need to compute the heap space consumed by a heap element at a specific address using `size_of_addr`. The `flat_measure` function will be explained below.

```
size_of_addr lims refs blocks (TStamp ts) ≝
  case lookup ts blocks of
   Some (Block _ _ vs) ⇒ 1 + |vs| + flat_measure lims vs
  | _ ⇒ 0
size_of_addr lims refs blocks (RStamp p) ≝
  case lookup p refs of
   None ⇒ 0
  | Some (ValueArray vs) ⇒ 1 + |vs| + flat_measure lims vs
  | Some (ByteArray _ bs) ⇒ |bs| div (arch_size lims div 8) + 2
```

In the above definition, we can see that an address of a `Block` $t\ n\ vs$ has size $1 + |vs| +$ `flat_measure lims` *vs*. Here 1 is the for the header of the heap element; $|vs|$ is for the length of the payload of the heap element; and `flat_measure lims` *vs* is to account for the heap elements that are immediately reachable from this block, but have no address. The definition of `flat_measure`, shown in Figure 6, counts `Block` and `RefPtr` values as having zero size, because they are already counted elsewhere.

Now we have a way to compute the set of reachable addresses and to compute size of a heap element at each address. Our final

$$
\begin{aligned}
&\texttt{flat\_measure}\ \mathit{lims}\ [\,]\ \stackrel{\text{def}}{=}\ 0\\
&\texttt{flat\_measure}\ \mathit{lims}\ (x :: y :: ys)\ \stackrel{\text{def}}{=}\\
&\quad \texttt{flat\_measure}\ \mathit{lims}\ [x]\ +\ \texttt{flat\_measure}\ \mathit{lims}\ (y :: ys)\\
&\texttt{flat\_measure}\ \mathit{lims}\ [\texttt{Word64}\ v_0]\ \stackrel{\text{def}}{=}\ 3\\
&\texttt{flat\_measure}\ \mathit{lims}\ [\texttt{Number}\ i]\ \stackrel{\text{def}}{=}\\
&\quad \text{if small\_num}\ \mathit{lims}.\text{arch\_64\_bit}\ i\ \text{then}\ 0\\
&\quad \text{else bignum\_size}\ \mathit{lims}.\text{arch\_64\_bit}\ i\\
&\texttt{flat\_measure}\ \mathit{lims}\ [\texttt{Block}\ v_{13}\ v_{14}\ v_{15}]\ \stackrel{\text{def}}{=}\ 0\\
&\texttt{flat\_measure}\ \mathit{lims}\ [\texttt{CodePtr}\ v_{16}]\ \stackrel{\text{def}}{=}\ 0\\
&\texttt{flat\_measure}\ \mathit{lims}\ [\texttt{RefPtr}\ v_{17}]\ \stackrel{\text{def}}{=}\ 0
\end{aligned}
$$

Figure 6: The definition of `flat_measure`

definition makes use of `sum_img` which sums the application of a given function f to all elements of a finite set s.

$$\texttt{sum\_img}\ f\ s\ \stackrel{\text{def}}{=}\ \texttt{fold\_finite\_set}\ (\lambda\, a\ b.\ f\ a\ +\ b)\ s\ 0$$

The top-level definition of the new heap measure is the following. This definition sums the size of all `Word64` and large `Number` values in the roots using `flat_measure`. This is added to `sum_img` of `size_of_addr` applied to every reachable address in the heap.

$$
\begin{aligned}
&\texttt{flat\_size\_of}\ \mathit{lims}\ \mathit{refs}\ \mathit{blocks}\ \mathit{roots}\ \stackrel{\text{def}}{=}\\
&\quad \texttt{flat\_measure}\ \mathit{lims}\ \mathit{roots}\ +\\
&\quad \texttt{sum\_img}\ (\texttt{size\_of\_addr}\ \mathit{lims}\ \mathit{refs}\ \mathit{blocks})\\
&\qquad (\texttt{reachable\_v}\ \mathit{refs}\ \mathit{blocks}\ (\texttt{to\_addr}\ \mathit{roots}))
\end{aligned}
$$

Even though this definition is very different in formulation from the original `size_of`, shown in Figure 5, it computes the same number.

4 IMPROVING ON SIZE_OF

To illustrate the challenges of reasoning about `size_of`, consider the following reordering property:

$$\texttt{size\_of}\ [x, y]\ \mathit{refs}\ \mathsf{LN}\ =\ \texttt{size\_of}\ [y, x]\ \mathit{refs}\ \mathsf{LN}$$

Intuitively, this property must hold for a measuring function as the values considered are the same. However, with `size_of` both sides of the equality might perform completely different traversals:

$$
\begin{aligned}
&\texttt{size\_of}\ [y]\ \mathit{refs}\ \mathsf{LN}\ =\ (n_{y1}, \mathit{refs}_{y1}, \mathit{seen}_{y1})\ \land\\
&\texttt{size\_of}\ [x]\ \mathit{refs}\ \mathsf{LN}\ =\ (n_{x1}, \mathit{refs}_{x1}, \mathit{seen}_{x1})\ \land\\
&\texttt{size\_of}\ [y]\ \mathit{refs}_{x1}\ \mathit{seen}_{x1}\ =\ (n_{y2}, \mathit{refs}_{y2}, \mathit{seen}_{y2})\ \land\\
&\texttt{size\_of}\ [x]\ \mathit{refs}_{y1}\ \mathit{seen}_{y1}\ =\ (n_{x2}, \mathit{refs}_{x2}, \mathit{seen}_{x2})\ \Rightarrow\\
&\quad (n_{y1} + n_{x2}, \mathit{refs}_{x2}, \mathit{seen}_{x2})\ =\ (n_{x1} + n_{y2}, \mathit{refs}_{y2}, \mathit{seen}_{y2})
\end{aligned}
$$

This mismatch between applications exposes the following problems:

- There is no straightforward relation between the two measurements of $[x]$ (or those of$[y]$) as `size_of` is applied to different arguments.
- All blocks in $[x]$ and $[y]$ with the same timestamps must have the same contents; otherwise, the order in which blocks are counted will affect the result due to aliasing mitigation.

These issues can be overcome by introducing well-formedness conditions on $[x]$ and $[y]$, and by generalizing the property statement to one more suited for induction (e.g. list permutations). However, these kinds of hurdles appear more often than one might want for such a crucial function.

In stark contrast, reordering can be trivially proved for `flat_size_of`. First, a call to `flat_measure` traverses a list to add non-root values, and is thus unaffected by permutations. Similarly, the initial root set computed by `to_addr` is the union of all addresses in the list of values and is again unaffected by reordering. Therefore, the remaining application of `sum_img` is being applied to the same arguments.

This ease of reasoning is what makes `flat_size_of` better suited for proofs of space safety as shown in the rest of this section.

4.1 A layout for space safety proofs

As mentioned before, to prove the space safety of a DataLang program one must show the preservation of safe_for_space through its evaluation (Section 2.2). As most DataLang programs are composed of multiple recursive functions, it is often necessary to separately prove space safety for some of them. To prove a function is space safe, one generally needs three kinds of assumptions:

(A1) The current space consumption is below the limits or roughly `size_of` + $M \leq$ heap_limit, where M is any extra space the function body needs.
(A2) A description of the arguments to the function, e.g., a list-shaped block, a number within 0 and 255, among others.
(A3) That the function is defined in s.code and its body corresponds with the code being evaluated

Resulting in the following layout:

$$
\begin{aligned}
\vdash\ &\mathsf{A1}\ \land\ \mathsf{A2}\ \land\ \mathsf{A3}\ \land\\
&s.\mathsf{safe\_for\_space}\ \land\\
&\texttt{evaluate}\ (\mathsf{fun\_body}, s) = (\mathit{res}, s')\ \Rightarrow\\
&\quad s'.\mathsf{safe\_for\_space}
\end{aligned}
$$

Proofs are by complete induction on the semantic clock and symbolic evaluation of the function body. Assumption (A2) should allow the evaluation of most of the function body. Moreover, intermediate updates to safe_for_space can be resolved using (A1). Once the recursive call is reached, assumption (A3) replaces the function call with the function's body such that the inductive hypothesis can be applied. At this point in the proof, assumptions must be established again for the state at the function call. (A3) is trivial as s.code does not change. (A2) might require work, but well-formed function code correctly operates on its values and thus provides good arguments. The proof of (A1) shown below is where things are more likely to become tricky:

$$
\begin{aligned}
\vdash\ &\ldots\\
&\texttt{size\_of\_heap}\ s\ +\ M\ s\ \leq\ s.\mathsf{limits.heap\_limit}\ \Rightarrow\\
&\quad \texttt{size\_of\_heap}\ s'\ +\ M\ s'\ \leq\ s'.\mathsf{limits.heap\_limit}
\end{aligned}
$$

Here, we must show that the space required at the recursive call (`size_of_heap` $s' + M\ s'$) is still less than heap_limit, assuming the space was enough in the original call. This amounts to proving that the required space decreases as the function recurses:

$$\vdash \ \ldots \ \Rightarrow$$
$$\texttt{size\_of\_heap}\ s' + M\ s' \ \le\ \texttt{size\_of\_heap}\ s + M\ s$$

This follows the intuition that function calls should take either progressively less space, or require an extra amount of memory bounded by M.

4.2 A tail recursive example

Consider a hypothetical tail-recursive function `ftail` with the following features:

- Takes a list of numbers as argument.
- Operates over the head of the list consuming constant space.
- Makes a tail-recursive call with the tail of the list.

Now assume we wanted to prove `ftail` space safe for concrete argument `[1,2,3]`. Instantiating the proof layout from the previous section, we arrive at the proof goal shown below:

$$\begin{array}{l}
\vdash \ \texttt{size_of_heap}\ s + C \ \le\ s.\text{limits.heap_limit} \ \wedge \\
\quad \texttt{lookup}\ \text{''ftail''}\, s.\text{code} = \text{Some}\ \texttt{ftail_body} \ \wedge \\
\quad s.\text{locals} = \\
\qquad \{\ 0 \ \mapsto\ \texttt{Block}\ 8\ \texttt{cons_tag}\ [\texttt{Number}\ 1, \\
\qquad\qquad\qquad \texttt{Block}\ 7\ \texttt{cons_tag}\ [\texttt{Number}\ 2, \ldots]]\} \ \wedge \\
\quad s.\text{safe_for_space} \wedge \\
\quad \texttt{evaluate}\ (\texttt{ftail_body}, s) = (res, s') \ \Rightarrow \\
\quad\ s'.\text{safe_for_space}
\end{array}$$

Where C is the (constant) space the function uses to operate.

Using assumptions (A1), (A2), and (A3), most of the proof can proceed by evaluation; until the tail recursive call to `ftail` is reached and we must establish assumption (A1) again:

$$\texttt{size\_of\_heap}\ s' \ \le\ \texttt{size\_of\_heap}\ s$$

Which by definition of `size_of_heap` and the abbreviation of `extract_stack` s.stack ++ `global_to_vs` s.global as *rest* simplifies to:

$$\begin{array}{l}
\texttt{size_of}\ ([\texttt{Block}\ 7\ \texttt{cons_tag}\ [\texttt{Number}\ 2, \ldots]]] ++ rest) \\
\qquad refs\ \text{LN} \\
\le \\
\texttt{size_of}\ ([\texttt{Block}\ 8\ \texttt{cons_tag}\ [\texttt{Number}\ 1, \\
\qquad\qquad \texttt{Block}\ 7\ \texttt{cons_tag}\ [\texttt{Number}\ 2, \ldots]]] ++ rest) \\
\qquad refs\ \text{LN}
\end{array}$$

And given the equality `size_of` $rest\ refs$ LN = $(n, refs', seen)$ can be rewritten further to:

$$\begin{array}{l}
\texttt{size_of}\ [\texttt{Block}\ 7\ \texttt{cons_tag}\ \ldots]\ refs'\ seen \ \le \\
\quad \texttt{size_of}\ [\texttt{Block}\ 8\ \texttt{cons_tag}\ \ldots]\ refs'\ seen
\end{array}$$

At this point, it would appear the proof is almost done, as we are essentially testing if the space occupied by a list (`[1,2,3]`) is greater than that of its tail (`[2,3]`), which it must be. However, due to `size_of`'s handling of timestamps and the fact that *seen* is symbolic, one can not show this inequality without additional assumptions. Concretely, one can think of a scenario where from the timestamps in the block only 8 is in *seen*; this will result in the measurement being 0 at the right of the inequality and 4 on the left, a clear falsehood.

$$\begin{array}{l}
8 \in seen \wedge 7 \notin seen \wedge \ \ldots \wedge \\
\texttt{size_of}\ [\texttt{Block}\ 7\ \ldots]\ refs'\ seen = (4, refs'', seen'') \ \wedge \\
\texttt{size_of}\ [\texttt{Block}\ 8\ \ldots]\ refs'\ seen = (0, refs''', seen''') \ \Rightarrow \\
4 \le 0
\end{array}$$

Therefore, the proof goal must be extended with a predicate ensuring that if timestamps 8 is in *seen* it must be the case that 7 and all other subsequent timestamps in the list are also in *seen.*

Proving such predicate and all its associated lemmas takes considerable work, to the point that, similar mechanisms in existing space safety proofs take around 25% of the Theory file. The issue is further aggravated by the fact that this kind of predicates can not be easily generalized for all types of values and must be re-written every time a new type is used.

If we were to switch our reasoning to `flat_size_of` our proof goal could be greatly simplified:

$$\begin{array}{l}
\texttt{flat_size_of}\ refs\ blocks\ ([\texttt{Block}\ 7\ \ldots] ++ rest) \\
\le \\
\texttt{flat_size_of}\ refs\ blocks\ ([\texttt{Block}\ 8\ \ldots] ++ rest)
\end{array}$$

While we can no longer "drop" *rest* from the roots, `flat_size_of` more than makes up for this with its use of sets and relations to represent the reachable memory. To showcase this, consider the following lemma:

$$\begin{array}{l}
\texttt{flat_measure}\ lims\ x = \texttt{flat_measure}\ lims\ x \ \wedge \\
\texttt{reachable_v}\ refs\ blocks\ (\texttt{to_addrs}\ x) \subseteq \\
\quad \texttt{reachable_v}\ refs\ blocks\ (\texttt{to_addrs}\ y) \ \Rightarrow \\
\ \ \texttt{flat_size_of}\ lims\ refs\ blocks\ x \ \le \\
\quad\ \ \texttt{flat_size_of}\ lims\ refs\ blocks\ y
\end{array}$$

Which states that if the reachable set of addresses from two roots x and y are subsets, then the space measurement of x done by `flat_size_of` must be less than that of y. Using this lemma the proof goal becomes trivial:

$$\begin{array}{l}
\{\texttt{TStamp}\ 7\} \cup \texttt{reachable_v}\ \ldots\ (\texttt{to_addrs}\ rest) \\
\quad \subseteq \{\texttt{TStamp}\ 8, \texttt{TStamp}\ 7\} \cup \texttt{reachable_v}\ \ldots\ (\texttt{to_addrs}\ rest)
\end{array}$$

This would conclude the proof with little more than basic set reasoning.

Is this ease of reasoning in the presence of (possibly) aliased values what makes `flat_size_of` a suitable measuring function for a cost semantics. In particular, the reachability-based approach to gathering live data aids the function, and its reasoning, to not be concerned with "where" a value is or how it is structured, and focus solely in its effect on the space measurement. In contrast, reasoning about `size_of` constantly requires additional safeguards structural guarantees that should only concern the memory model.

5 SOUNDNESS

At the time of submission, we have not yet started proving soundness of the new version of the cost semantics. However, our intention is to prove soundness of the new cost semantics so that all future uses of CakeML's space cost semantics can make use of this improved formulation that we have presented in this paper.

There are two options for proving soundness: (1) soundness of the new formulation can be proved w.r.t. the original formulation;

or (2) the old formulation could be replaced by the new formulation in the definition of the DataLang semantics. Option (1) is neatly self-contained to mostly a proof about the relationship between the old and the new space measures. However, if the new formulation is to truly replace the old one, then option (2) is the right way to go, even though it requires redoing some fiddly proofs in the middle of the correctness proofs of the CakeML compiler. At the time of writing, we are leaning towards option (2), since we do not want the old one to become a burden for proof maintenance.

We expect to have soundness of the new version of the cost semantics proved by IFL's post-proceedings deadline.

6 RELATED WORK

Work on verified cost semantics of verified compiler is available for the CompCert [7] and CakeML [6] compilers. Carbonneaux et al. [4] develop a source level logic for stack space reasoning that translates to the CompCert compiler output. Besson *et al.* extends CompCert's memory model with finite memory and integer pointers in CompCertS [1–3]; which allows for memory usage estimates of C functions that are proven to be bounds of the compiled code. The cost semantics of the CakeML compiler [5] is to our knowledge the only verified costs semantics for a high-level garbage-collected language and as such is a good candidate for further research in the topic.

7 CONCLUSION

This paper has proposed a new flat reachability based heap space cost measure for CakeML's verified space cost semantics. Early experiments suggests that the new formulation of the heap measure is significantly more pleasant to use in proofs of space safety. We plan to replace the original formulation of the heap measure with this new one. The hope is that the new formulation will make reasoning of space cost scale beyond simple examples.

REFERENCES

[1] 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.

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

[3] 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.

[4] 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.

[5] 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 (2020), 204:1–204:29. https://doi.org/10.1145/3428272

[6] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 179–192. https://doi.org/10.1145/2535838.2535841

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

[8] 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.

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