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

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 reasoning about the space safety 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 safety 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. 2021. A flat reachability-based measure for CakeML's cost semantics. In *33rd Symposium on Implementation and Application of Functional Languages (IFL '21), September 1–3, 2021, Nijmegen, Netherlands.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3544885.3544887

1 INTRODUCTION

Functional languages aid the development of complex programs by providing programmers with many abstractions (e.g., polymorphism, garbage collection, ADTs, among others). However, these abstractions often come at the cost of increased memory usage and compiler complexity. These drawbacks make it difficult to judge space safety, i.e., how much memory a program will need in order to run without encountering out-of-memory errors.

To avoid out-of-memory errors, the CakeML project has recently developed a verified cost semantics [8] that makes it possible to prove the space safety of programs generated by the CakeML compiler. CakeML's cost semantics predicts when a program runs out of memory using a space measuring function, `size_of`. The `size_of` function is used at all allocation sites to check if the memory usage has surpassed a given limit. To prove that a program does not run out of memory, it is enough to show that `size_of`, as used by the formal semantics, stays within the limits.

IFL '21, September 1–3, 2021, Nijmegen, Netherlands

ACM ISBN 978-1-4503-8644-9/21/09.
https://doi.org/10.1145/3544885.3544887

At its core, the measuring function `size_of` uses a single recursive descent to discover reachable nodes in the heap and compute their sum (while taking aliasing into account in order to avoid gross over-approximations). Space safety proofs must then carefully use the definition and properties of `size_of` to reach Q.E.D. Unfortunately, the current formulation of `size_of` is not naturally compositional, making its definition tricky to use, and forcing related properties to require complicated assumptions to hold.

The non-compositionality of `size_of` makes reasoning about space safety a cumbersome endeavor. The majority of a space safety proof is dedicated to tedious by-hand accounting of arguments and establishing complicated assumptions. Overall, while space safety can be established using CakeML's cost semantics, its utility is severely limited by the amount of effort necessary to complete these `size_of` proofs.

This paper makes it easier to reason about CakeML's cost semantics by defining (and proving soundness of) an alternative space measuring function, called `flat_size_of`. The new function `flat_size_of` is defined in two steps: first, it computes the set of reachable nodes, and then computes the sum of the size of the data at those nodes. The new formulation is compositional and, thus, one can express properties and conduct proofs more naturally than with `size_of`. Our initial experiments suggest that `flat_size_of` makes space safety proofs less cumbersome (i.e., require less assumptions) and more manageable (i.e., shorter theory files) than `size_of` equivalents.

This paper makes the following contributions:

- Defines `flat_size_of` (in Section 3) as a new reachability-based measuring function, which is significantly simpler to work with than the original `size_of` (Section 2.3).
- Demonstrates (in Section 4) how `flat_size_of` overcomes some of the most significant problems of `size_of`.
- Discusses (in Section 5) how `flat_size_of` was proved sound, and how future space safety proofs can use it.

All the work presented in this paper is machined-checked using the HOL4 theorem prover in the context of the CakeML compiler verification project; artifacts and example proofs can be found here.

2 A VERIFIED COST SEMANTICS

The cost semantics for the CakeML compiler [8] 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 measuring function called `size_of`. This `size_of` function computes the space consumption of all values that are reachable from the root values obtained from the stack and global variables. This `size_of` function is defined to carefully track 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 (concrete or abstract) limit that the semantics of its DataLang representation never sets `safe_for_space` 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 [8, 15].

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 appends its input to the list `[1,2,3]`. 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 appends its argument to `[1,2,3]`

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, the result of a primitive operation is always assigned (:=) to a local variable, which is represented as a natural number. On line 0, argument 0 corresponds to the source code binding `x`. Line 1 allocates 9 slots of space, since for each of the three `Cons` space for the constructor, head, and tail is required. 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. On 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. Here `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 representation of the CakeML list `[1,2,3]`. This is the value resulting 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 stack 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.

```
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.

The semantics of DataLang is defined as a functional big-step semantics [14]. 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 evaluation, the DataLang semantics can also be expressed as a shallowly embedded state-exception monad. This is the representation used in `app123` and by partially evaluating the first three operations we can inspect its intermediate 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:

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

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:

```
s with
safe_for_space :=
 (s.safe_for_space ∧
  size_of_heap s + k ≤ s.limits.heap_limit)
```

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:

```
s with
safe_for_space :=
 (s.safe_for_space ∧
  size_of_stack s + k ≤ s.limits.stack_limit)
```

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 a 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 ∅ 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 the space consumed by all values reachable from the initial list of root values. Figure 5 shows its definition with *seen* (a set of timestamps), and *refs* as additional arguments.

```
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 `size_of`.

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. By contrast, the handling of `Block` and `RefPtr` requires additional bookkeeping to avoid counting the same value twice (aliasing), and as such, is where most of the complexity of `size_of` lies. In the case of `Block` values, a set of already-measured (*seen*) timestamps is kept to avoid counting identical blocks multiple times; this mechanisms relies on 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.

To illustrate how `size_of` handles aliasing consider the following examples:

With x equal to `Block` *ts tag* [`Number 1`] throughout:

(B1) $ts \notin seen \implies$
`size_of` $[x]$ *refs seen* $= (2, refs, \{ts\} \cup seen)$

(B2) $ts \in seen \implies$
`size_of` $[x]$ *refs seen* $= (0, refs, seen)$

(B3) $ts \notin seen \wedge ts \in seen' \wedge$
`size_of` *xs refs seen* $= (n, refs', seen') \implies$
`size_of` $(x :: xs)$ *refs seen* $= (n, refs', seen')$

Intuitively, blocks whose timestamps have not been "seen" (i.e., $ts \notin seen$) are always counted (B1). Moreover, blocks with already "seen" timestamps are ignored (B2), as this hints at the block being present in previous values (B3) at the back of the list — since `size_of` operates from the back.

The definition of `size_of` succeeds at providing tight bounds, mitigating the effects of aliasing, and traversing only 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 back of the argument list directly affect subsequent ones through pointers or timestamps — as in example (B3). Thus, conceptually simple properties (e.g., the reordering of values) are hard to prove and apply. The shortcomings of `size_of` are explained in more detail in Section 4.

3 A NEW FLAT REACHABILITY-BASED MEASUREMENT

This section shows the definition of our new heap cost measuring function, `flat_size_of`, which improves on 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 (Section 3.1), and then sums the sizes of all heap elements at those addresses (Section 3.2). We refer to this new formulation as "flat" because operations occur mostly over sets and avoid recursing into the structure of values. The rest of this section goes into the details of the definition of `flat_size_of`.

3.1 The set of all reachable addresses

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. Intuitively, an address is meant to represent any value that might contain or reference other values. Therefore, we represent an address as either the timestamp (`TStamp`) of a Block (remember each block has a unique timestamp) or the pointer to a reference (`RStamp`).

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

From a list of DataLang values, we can compute, using `to_addrs`, a set of corresponding addresses. Note that `to_addrs` does not recurse into `Block` values, because it only wants to collect the immediately reachable addresses of the given values.

```
to_addrs [] ≝ ∅
to_addrs (Block ts tag [] :: xs) ≝ to_addrs xs
to_addrs (Block ts tag (v :: vs) :: xs) ≝
  { BlockAddr ts } ∪ to_addrs xs
to_addrs (RefPtr ref :: xs) ≝
  { RefAddr ref } ∪ to_addrs xs
```

Omitted value kinds in the definition of `to_addrs` do not have addresses in this representation; in those cases, recursion directly continues through the list.

As a precursor to reachability, we define the `next` relation which consider pairs of addresses that are one-step reachable. When

provided with value mappings for pointers (*refs*) and timestamps (*blocks*), the relation `next` *refs blocks a b* holds only if *b* is immediately reachable from *a* using one of such mappings.

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

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

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

Therefore, from an initial set of addresses we can neatly describe all reachable addresses using the reflexive transitive closure (*) of `next`. Note that this approach implicitly handles aliasing by declaratively defining the set of reachable addresses; avoiding non-termination concerns and ruling out duplicated values.

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

With these functions we can state the set of all reachable addresses from a list of root values as follows.

```
reachable_v refs blocks (to_addrs roots)
```

Crucially, the result of `reachable_v` is only finite if the initial set of roots is finite; a requirement to iterate on the resulting set in subsequent functions. Fortunately, the result of `to_addrs` *roots* is know to be finite as it only turns the finitely many elements of *roots* into addresses. Thus, one can prove the following.

```
⊢ FINITE (reachable_v refs blocks (to_addrs roots))
```

3.2 Adding it all up

In order to sum the sizes of all the reachable values, we need a function that can compute the heap space consumed by a heap element at a specific address. For this purpose, we define a function `size_of_addr` which given an address returns the size of that heap element.

```
flat_measure lims (Word64 v0) ≝ 3
flat_measure lims (Number i) ≝
  if small_num lims.arch_64_bit i then 0
  else bignum_size lims.arch_64_bit i
flat_measure lims (Block v5 v6 v7) ≝ 0
flat_measure lims (CodePtr v8) ≝ 0
flat_measure lims (RefPtr v9) ≝ 0
```

Figure 6: The definition of `flat_measure`

```
size_of_addr lims refs blocks (TStamp ts) ≝
  case lookup ts blocks of
    Some (Block _ _ vs) ⇒
      1 + |vs| + sum(map (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| + sum(map (flat_measure lims) vs)
  | Some (ByteArray _ bs) ⇒
      |bs| div (arch_size lims div 8) + 2
```

In the definition above, we see that an address of a `Block` *t n vs* has size `1 + |vs| + sum (map (flat_measure lims) vs)`. Here 1 is the space 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 a way to compute the size of a heap element at each address. Our final definition makes use of $\sum$ which sums the application of a given function f to all elements of a finite set s.

$$\sum f\ s \stackrel{\text{def}}{=} \texttt{fold\_set}\ (\lambda\, e\ acc.\ f\ e\ +\ acc)\ s\ 0$$

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 $\sum$ of `size_of_addr` applied to every reachable address in the heap.

```
flat_size_of lims refs blocks roots ≝
  sum (map (flat_measure lims) roots) +
  ∑ (size_of_addr lims refs blocks)
    (reachable_v refs blocks (to_addrs roots))
```

Even though this definition is very different in formulation from the original `size_of`, shown in Figure 5, it computes the same number while providing various advantages. Aliasing is implicitly handled and there is no need for book-keeping of pointers and timestamps. Moreover, the clear separation between the gathering (`reachable_v`) and measuring (`size_of_addr`,`flat_measure`) of

heap elements makes for a more concise definition than combining both operations in a single recursive descent. More generally, the main advantage of the `flat_size_of` approach is that it abstracts the structure of the heap into a model (the set of all reachable addresses) that is considerably easier to operate over; this is in stark contrast of `size_of`, which operates directly on the structure of the heap and therefore must deal with its associated complexity.

3.3 Requirements

In order for `flat_size_of` to be a viable replacement of `size_of`, some support in DataLang's semantics is required. Specifically, the semantic state must provide suitable values for the auxiliary arguments *lims*, *refs*, and *blocks*. However, in the current semantics, only s.limits (*lims*) and s.refs (*refs*) are available.

To add support for `flat_size_of` to the semantics, we extended the semantic state to include a mapping from timestamps to blocks: s.all_blocks. This field is updated every time a block is created, adding a mapping between the block's timestamp and the block itself (i.e., *ts* $\mapsto$ `Block` *ts tag l*). Since timestamps uniquely identify blocks, the mapping in s.all_blocks is always consistent with all blocks in the heap, and by extension, all addresses derived by `reachable_v`.

Given this set up, one can define the top-level cost measuring function `flat_size_of_heap` in a way similar to `size_of_heap`.

```
flat_size_of_heap s =def
  flat_size_of s.limits s.refs s.all_blocks (stack_to_vs s)
```

4 FLAT_SIZE_OF IS BETTER THAN SIZE_OF

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

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

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:

$$\begin{array}{l}\texttt{size\_of}\ [y]\ \mathit{refs}\ \emptyset\ =\ (n_{y1}, \mathit{refs}_{y1}, \mathit{seen}_{y1})\ \wedge\\ \texttt{size\_of}\ [x]\ \mathit{refs}\ \emptyset\ =\ (n_{x1}, \mathit{refs}_{x1}, \mathit{seen}_{x1})\ \wedge\\ \texttt{size\_of}\ [y]\ \mathit{refs}_{x1}\ \mathit{seen}_{x1}\ =\ (n_{y2}, \mathit{refs}_{y2}, \mathit{seen}_{y2})\ \wedge\\ \texttt{size\_of}\ [x]\ \mathit{refs}_{y1}\ \mathit{seen}_{y1}\ =\ (n_{x2}, \mathit{refs}_{x2}, \mathit{seen}_{x2})\ \Rightarrow\\ (n_{y1} + n_{x2}, \mathit{refs}_{x2}, \mathit{seen}_{x2})\ =\ (n_{x1} + n_{y2}, \mathit{refs}_{y2}, \mathit{seen}_{y2})\end{array}$$

This mismatch 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 the new `flat_size_of` function. 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 $\sum$ 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 space consumption before the function call is below the limits or roughly `size_of_heap` s + $M \leq$ `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:

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

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 most likely to become tricky:

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

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:

$$\begin{array}{l}\vdash\ \ldots\ \Rightarrow\\ \quad \texttt{size\_of\_heap}\ s'\ +\ M\ s'\ \leq\ \texttt{size\_of\_heap}\ s\ +\ M\ s\end{array}$$

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 hypothetical 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 want 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:

$$
\begin{array}{l}
\vdash \texttt{size\_of\_heap}\ s\ +\ C\ \leq\ s.\texttt{limits.heap\_limit}\ \wedge \\
\quad \texttt{lookup}\ ''\texttt{ftail}''\ s.\texttt{code} = \texttt{Some}\ \texttt{ftail\_body}\ \wedge \\
\quad s.\texttt{locals} = \\
\qquad \{\ 0\ \mapsto\ \texttt{Block 8 cons\_tag [Number 1,} \\
\qquad\qquad \texttt{Block 7 cons\_tag [Number 2,...]]}\}\ \wedge \\
\quad s.\texttt{safe\_for\_space}\ \wedge \\
\quad \texttt{evaluate}\ (\texttt{ftail\_body}, s) = (\mathit{res}, s')\ \Rightarrow \\
\qquad s'.\texttt{safe\_for\_space}
\end{array}
$$

Above, 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, leading to an inequality of the form:

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

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

$$
\begin{array}{l}
\texttt{size\_of}\ (\texttt{[Block 7 cons\_tag [Number 2,...]]} +\!+ \mathit{rest}) \\
\qquad s.\texttt{refs}\ \emptyset \\
\leq \\
\texttt{size\_of}\ (\texttt{[Block 8 cons\_tag [Number 1,} \\
\qquad\quad \texttt{Block 7 cons\_tag [Number 2,...]]} +\!+ \mathit{rest}) \\
\qquad s.\texttt{refs}\ \emptyset
\end{array}
$$

Moreover, since `size_of` operates from the back of the list, we can abstract away the common measurement of *rest* at both sides as `size_of` *rest* s`.refs` $\emptyset = (n, \mathit{refs}, \mathit{seen})$, and rewritten to:

$$
\begin{array}{l}
\texttt{size\_of [Block 7 cons\_tag ...]}\ \mathit{refs}\ \mathit{seen}\ \leq \\
\quad \texttt{size\_of [Block 8 cons\_tag ...]}\ \mathit{refs}\ \mathit{seen}
\end{array}
$$

At this point, it would appear that 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]`), a rather intuitive claim. 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 only 8 is in *seen* and no other timestamps in the block is in *seen*. Such a situation will result in the measurement being 0 at the right of the inequality and 4 on the left, a clear falsehood.

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

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 block are also in *seen*.

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

By switching our reasoning to `flat_size_of`, our proof goal is greatly simplified:

$$
\begin{array}{l}
\texttt{flat\_size\_of}\ s.\texttt{refs}\ s.\texttt{all\_blocks}\ (\texttt{[Block 7 ...]} +\!+ \mathit{rest}) \\
\leq \\
\texttt{flat\_size\_of}\ s.\texttt{refs}\ s.\texttt{all\_blocks}\ (\texttt{[Block 8 ...]} +\!+ \mathit{rest})
\end{array}
$$

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, which states that if the reachable set of addresses from two roots x and y are subsets, and `flat_measure` then the space measurement of x done by `flat_size_of` must be less than that of y.

$$
\begin{array}{l}
\texttt{flat\_measure}\ \mathit{lims}\ x \leq \texttt{flat\_measure}\ \mathit{lims}\ y\ \wedge \\
\texttt{reachable\_v}\ \mathit{refs}\ \mathit{blocks}\ (\texttt{to\_addrs}\ x) \subseteq \\
\quad \texttt{reachable\_v}\ \mathit{refs}\ \mathit{blocks}\ (\texttt{to\_addrs}\ y)\ \Rightarrow \\
\quad \texttt{flat\_size\_of}\ \mathit{lims}\ \mathit{refs}\ \mathit{blocks}\ x\ \leq \\
\qquad \texttt{flat\_size\_of}\ \mathit{lims}\ \mathit{refs}\ \mathit{blocks}\ y
\end{array}
$$

Using this lemma the proof goal becomes trivial:

$$
\begin{array}{l}
\{\texttt{TStamp 7,...}\} \cup \texttt{reachable\_v ...}\ (\texttt{to\_addrs}\ \mathit{rest}) \\
\quad \subseteq \{\texttt{TStamp 8,TStamp 7,...}\} \cup \\
\qquad \texttt{reachable\_v ...}\ (\texttt{to\_addrs}\ \mathit{rest})
\end{array}
$$

One can then conclude the proof using basic set reasoning.

It is this ease of reasoning in the presence of (possibly) aliased values that 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 in the heap structure a value is located, and focus solely on its effect on the space measurement. In contrast, reasoning about `size_of` constantly requires additional safeguards and guarantees on the heap structure to be able to relate two measurements, as seen in our previous example.

4.3 A concrete tail-recursive example

Consider the CakeML function `sum` defined below:

```
fun sum xs = foldl (+) 0 xs
```

Where `xs` is a list of (unbounded) integers and `(+)` is integer addition with support for bignum arithmetic. As expected, a call to `sum` computes the addition of all the elements of `xs`.

The space safety of `sum` follows from a similar intuition as the one presented for `ftail` (Section 4.2) even after considering the space consumption of bignum arithmetic (+) and accumulator arguments (`foldl`). This relation is made evident by the space safety proof of `sum` currently available in the CakeML project, which shares `ftail`'s proof structure, and thus, its issues regarding the use of `size_of`. Specifically, the proof requires additional assumptions and theorems to enforce the timestamps in `xs`'s representation are correctly traversed — i.e., it is never the case that a timestamp at the head of the list has been "seen" and one in the tail has not.

Fortunately, as with `ftail`m the space safety proof of `sum` can also be improved by switching to `flat_size_of`. As an experiment we updated the proof of the `sum` example to use `flat_size_of` and the following quantitative improvements (in LOC) where archived:

- Assumptions outside of the scope of (A1), (A2), and (A3) were removed.
- 14 auxiliary lemmas and definitions were removed.
- The section of the proof dedicated to re-establishing (A1) shrunk by 32%.
- The proof of space safety shrunk by 13%.
- The entire file for this proof and all auxiliary lemmas shrunk by 28%.

Furthermore, the new proof text for `flat_size_of` only utilized definitions and standard set reasoning leading to a nicer proof.

In summary, this updated space safety proof demonstrates the advantages of `flat_size_of` over `size_of`.

5 SOUNDNESS

We have proved `flat_size_of_heap` sound. More specifically, we have proved that, under reasonable assumptions `size_inv` s, the number computed by `flat_size_of_heap` is equal to the number computed by `size_of_heap`.

$$\vdash \texttt{size\_inv}\ s \implies \texttt{size\_of\_heap}\ s = \texttt{flat\_size\_of\_heap}\ s$$

The `size_inv` assumption ensures that the values in $s.$all_blocks are consistent with those in the heap (i.e,. $s.$refs and `stack_to_vs`). Specifically, that for any `Block` *ts tag l* reachable in the heap, there is an entry $ts \mapsto$ `Block` *ts tag l* in $s.$all_blocks. Our proof of soundness requires `size_inv` because `flat_size_of`, unlike `size_of`, does not recurse over block values and instead must rely on an accurate block mapping to obtain the same results.

Using the equality that we have proved between `size_of_heap` and `flat_size_of_heap` one can rephrase any space safety proof, previously involving `size_of`, to be in terms of `flat_size_of`.

5.1 Updates to CakeML's cost semantics

The CakeML's cost semantics was updated to facilitate the usage of `flat_size_of` in space safety proofs.

The main hurdle when switching to `flat_size_of` is establishing `size_inv` (so the soundness theorem can apply). To address this, we extended (at the DataLang level) how $s.$safe_for_space is updated to include `size_inv` as an antecedent.

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

With this change, if one starts a typical space safety proof (Section 4.1) and uses `flat_size_of_heap`, instead of `size_of_heap`, for the (A1) assumption (i.e., heap measurement are within the limits), then, whenever $s.$safe_for_space needs to be re-established `size_inv` will be available as an assumption.

The addition of `size_inv` to the cost semantics was proven sound w.r.t. the rest of the compiler, as an update to `size_of_heap`'s original soundness proof. Informally, if `size_inv` holds for the initial semantic state and is preserved by the semantic as an invariant, then, its addition as an antecedent in $s.$safe_for_space does not affect the field's value. Therefore, the addition of `size_inv` makes proofs more convenient while keeping the semantics essentially unchanged.

6 RELATED WORK

Verified cost semantics are available for the CompCert [12] and CakeML [11] verified compilers. Carbonneaux et al. [5] 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 [2–4], which allows for memory usage estimates of C functions that are proven to be bounds of the compiled code.

In recent work, Madiot and Pottier [13] develop a separation logic for conveniently reasoning about heap space usage in the presence of garbage collection. However, their cost semantics is not proved correct w.r.t. a concrete compiler.

There have been many other approaches to source-level analysis of space cost. For example, resource-aware type systems based on refinement types [6, 7, 10] can be used to obtain bounds for source programs. Moreover, a program's resource usage can be directly encoded as a refinement type in compilers with support for such type systems [9]. Time-complexity annotations and indexes in types [16] can also be used to express costs. Another approach is for proof-carrying code to be equipped with a resource usage proof w.r.t. a resource-aware program logic [1]. In general, theses approaches provide formal estimates of costs for source-level programs, however, they forgo the effects compilation and program transformation can have on resource consumption. Source-level cost analysis techniques could be used on DataLang programs to facilitate reasoning further, however, we have not yet investigated this approach.

7 CONCLUSION

In this paper we have proposed a new reachability-based measure for CakeML's verified cost semantics. The examples explored here suggest that the new formulation is better suited for space safety proofs. We found that the need for extra assumptions and auxiliary lemmas has been greatly reduced and, as a consequence, proof scripts are more concise and easy to read, making the whole proving process more scalable. Overall, we hope that by making space safety

reasoning easier, more ambitious verification projects that prevent out-of-memory errors can be undertaken.

REFERENCES

[1] David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. 2007. A program logic for resources. *Theor. Comput. Sci.* 389, 3 (2007), 411–445. https://doi.org/10.1016/j.tcs.2007.09.003

[2] 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.

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

[4] 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.

[5] 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.

[6] Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational cost analysis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 316–329. https://doi.org/10.1145/3009837.3009858

[7] Ezgi Çiçek, Deepak Garg, and Umut A. Acar. 2015. Refinement Types for Incremental Computational Complexity. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer, 406–431. https://doi.org/10.1007/978-3-662-46669-8_17

[8] 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

[9] Martin A. T. Handley, Niki Vazou, and Graham Hutton. 2020. Liquidate your assets: reasoning about resource usage in liquid Haskell. *Proc. ACM Program. Lang.* 4, POPL (2020), 24:1–24:27. https://doi.org/10.1145/3371092

[10] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource Aware ML. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings (Lecture Notes in Computer Science, Vol. 7358)*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer, 781–786. https://doi.org/10.1007/978-3-642-31424-7_64

[11] 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

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

[13] Jean-Marie Madiot and François Pottier. 2022. A Separation Logic for Heap Space under Garbage Collection. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022). http://cambium.inria.fr/~fpottier/publis/madiot-pottier-diamonds-2022.pdf to appear.

[14] 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.

[15] 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

[16] Peng Wang, Di Wang, and Adam Chlipala. 2017. TiML: a functional language for practical complexity analysis with invariants. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 79:1–79:26. https://doi.org/10.1145/3133903