Characteristic Formulae for Liveness Properties of Non-Terminating CakeML Programs

Johannes Åman Pohjola
Data61/CSIRO, Sydney, Australia
University of New South Wales, Sydney, Australia
johannes.amanpohjola@data61.csiro.au

Henrik Rostedt
Chalmers University of Technology, Gothenburg, Sweden

Magnus O. Myreen
Chalmers University of Technology, Gothenburg, Sweden

Abstract

There are useful programs that do not terminate, and yet standard Hoare logics are not able to prove liveness properties about non-terminating programs. This paper shows how a Hoare-like programming logic framework (characteristic formulae) can be extended to enable reasoning about the I/O behaviour of programs that do not terminate. The approach is inspired by transfinite induction rather than coinduction, and does not require non-terminating loops to be productive. This work has been developed in the HOL4 theorem prover and has been integrated into the ecosystem of proof tools surrounding the CakeML programming language.

2012 ACM Subject Classification Software and its engineering → Software verification; Theory of computation → Higher order logic; Theory of computation → Separation logic

Keywords and phrases Program verification, non-termination, liveness, Hoare logic

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.32

Supplement Material This work has been developed in HOL4; the sources are at `https://code.cakeml.org`

Funding *Johannes Åman Pohjola*: This work was sponsored in part by the U.S. Defense Advanced Research Projects Agency (DARPA). The views expressed are those of the authors and do not reflect the official policy or position of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government. Approved for Public Release, Distribution Unlimited.
Magnus O. Myreen: This work was supported by the Swedish Foundation for Strategic Research.

Acknowledgements We are grateful to Robert Sison and the anonymous reviewers for many constructive and insightful comments.

1 Introduction

Consider the following non-terminating ML program that prints the letter `y` forever.

```
fun yes() = (put_line "y"; yes());
val () = yes();
```

This program has the same behaviour as the default configuration of the Unix tool `yes`.

The `yes` program highlights a peculiar omission in Hoare-style programming logics to date: with only a few exceptions (Section 7), Hoare-like logics have only focused on reasoning about terminating programs or proving absence of bad behaviours. Few Hoare logics can state (let alone prove) that the `yes` program (1) will not terminate and (2) will produce a never-ending stream of `y` characters as output. Note that (2) is a liveness property.

One can argue that correctness is not important for toy examples such as `yes`. However, there are real-world programs that are both non-terminating and where correctness is important. Examples include embedded controllers, web servers, network filters and other software that is part of some device or infrastructure.

The fact that non-terminating behaviours are important is acknowledged in compiler verification where it is expected/common to prove that compilation preserves both terminating and non-terminating behaviours of the compiled programs; the CompCert [25] and CakeML [36] compilers are proved to preserve both types of behaviours.

In this paper we present how reasoning about total correctness of non-terminating programs can be integrated into and used in the context of a Hoare-like programming logic. Specifically, we describe how a proved-to-be-sound Hoare-style programming logic framework (characteristic formulae for CakeML) has been extended to enable reasoning about the I/O behaviour of non-terminating programs, thus enabling proofs of correctness properties such as (1) and (2).

For the `yes` program, our extended framework allows us to prove the following correctness theorem. The theorem is stated as a Hoare triple, where the precondition assumes that an I/O stream exists and nothing has been written to it, and the code is the application of function yes to an arbitrary argument *arg*. The postcondition is the interesting part here: we specify with POSTd that the program does not terminate. Furthermore, we assert that the trace of *io* produced by the diverging (i.e. non-terminating) execution is the infinite lazy list obtained by repeating the I/O event put_str_event "y\n" forever.

$$
\begin{array}{l}
\vdash \{|\mathsf{io\_events}\ []|\} \\
\quad \mathsf{yes} \cdot [\mathit{arg}] \\
\quad \{|\mathsf{POSTd}\ \mathit{io}.\ \mathit{io} = \mathsf{lrepeat}\ [\mathsf{put\_str\_event}\ \texttt{"y\textbackslash n"}]|\}
\end{array}
$$

In conventional Hoare logics, postconditions make a statement about final program states. However, non-terminating programs do not have final states, and the only interesting observation that can be made is what I/O they produce. Here the POSTd-postconditions make a statement about a possibly infinite trace of I/O events. One can think of this trace as the I/O events produced by an infinite execution of the program. As we will see later, formally POSTd-postconditions make a statement about *the least upper bound* of all I/O traces the program produces when allowed to run for different lengths of time.

Contributions

This paper makes the following contributions:

- It shows how a Hoare-like logic, characteristic formulae (CF) for CakeML, can be extended to enable correctness proofs for non-terminating programs. The approach is inspired by transfinite induction rather than coinduction, and does not require non-terminating loops to be productive. Our extension to CF enables reasoning about non-terminating programs in the same setting as terminating programs. We support postconditions with conditional (non-)termination, including conditions on external state such as the length or contents of input streams.
- Proofs of non-termination are, in two steps, turned into proofs about terminating functions. The first automatically step transforms the function under consideration into a repeat combinator applied to a terminating step function. The second step is to interactively prove that each execution of the (terminating) step function has behaviours that can

be composed to describe the infinite behaviour of the original function. Currently, this approach is limited to tail-recursive functions and does not consider mutual or higher-order recursion.[1]

- We demonstrate the use of CF on examples of non-terminating programs. The most complex example is a filter component for systems built on the verified seL4 microkernel. This filter component had an unwieldy and overly complicated proof before CF could be used, but now has a manageable proof that avoids reasoning directly at the level of the operational semantics.

2 Bird's-eye view of yes verification

We start with a high-level summary of the user experience when verifying the `yes` program. Subsequent sections explain the technical setup and several more interesting examples.

To prove the `yes` program, the user of the proof tools first applies a tactic that runs a verified source-to-source transformation on the recursive function `yes`. The transformation converts the goal we want (i.e. the theorem statement about `yes` from above) to a goal that talks about an application of repeat to a non-recursive function. Here `repeat` is defined as `fun repeat f x = repeat f (f x)`. This helps isolate the behaviour of each iteration of `yes`. The new goal statement is roughly:

$$\{|\mathsf{io\_events}\ []|\}$$
```
 repeat (fn () => put_line "y"; ()) ()
```
$$\{|\mathsf{POSTd}\ io.\ io = \mathsf{lrepeat}\ [\mathsf{put\_str\_event}\ \texttt{"y\textbackslash n"}]|\}$$

The tactic then invokes a general theorem that can reduce any such goal about `repeat` and POSTd into a goal about the terminating executions of its function argument. At this point, the proof goal splits into *two subgoals* and the user needs to instantiate three existentially quantified variables: *events*, *Hs* and *vs* (which will be explained below).

The first subgoal is a Hoare triple which asserts that each execution of the loop body respects *events*, *Hs* and *vs*. The Hoare triple is roughly the following. Think of *Hs i* as the precondition for the *i*th iteration of the loop. Here *events i* is a list of I/O events produced on the *i*th iteration; and *vs i* is a predicate that the argument given as input on iteration *i* satisfies. The POSTv-postcondition requires that the function returns normally and *vs* $(i+1)$ *retv* requires that the function produces the next argument.

$$\forall i.\ \{|Hs\ i * \mathsf{io\_events}\ []|\}$$
```
   (fn () => put_line "y"; ()) · [vs i]
```
$$\{|\mathsf{POSTv}\ retv.\ Hs\ (i+1) * \mathsf{io\_events}\ (events\ i) * \langle vs\ (i+1)\ retv\rangle|\}$$

The second subgoal requires the user to prove that the infinite concatenation lflatten of the list consisting of all I/O event lists, i.e. lgenlist *events* None, satisfies the desired postcondition:

$$\mathsf{lflatten}\ (\mathsf{lgenlist}\ events\ \mathsf{None}) = \mathsf{lrepeat}\ [\mathsf{put\_str\_event}\ \texttt{"y\textbackslash n"}]$$

Since each loop iteration behaves the same, we can instantiate *events*, *Hs*, *vs* with constant functions that return [put_str_event "y\n"], the empty heap predicate emp, and equality with the unit value (), respectively. The two subgoals are then easy to prove. Note that the proof of the first subgoal uses standard CF methods because it is about a terminating program.

[1] Note that this is not a significant restriction in practice, because most non-tail-recursive functions that have diverging semantics will actually terminate with an out-of-stack error message.

3 Background and new technical setup

3.1 Heaps and characteristic formulae

Characteristic formulae [5] (CF for short) is a technique for program verification that is based around a function that given a program p produces a predicate $\mathsf{cf}\ p$ called the *characteristic formula* of p. The idea is that in order to prove validity of the Hoare triple $\{\!|P|\!\}\ p\ \{\!|Q|\!\}$, it suffices to prove $\mathsf{cf}\ p\ P\ Q$. This helps because $\mathsf{cf}\ p$ is a higher-order logic formula and not a program; that is, we reduce reasoning about programs to shallowly embedded formulae in the meta-logic[2] that make no direct reference to the program source code. These formulae are typically much easier to reason about in a proof assistant. The present paper extends the work of Guéneau et al. [15] on characteristic formulae for CakeML, to which we refer for more background and details.

A *heap* is a set of *heap parts*; a heap part is either a memory cell $\mathsf{Mem}\ l\ v$, meaning that the value v is at memory location l, or an external resource $\mathsf{FFI\_part}\ st\ f\ ps\ e$, which describes a part of the world outside the CakeML runtime that can be affected by foreign function calls such as I/O operations. It records the state of the outside world (st), an oracle that models the effects of invoking foreign functions (f), the list of FFI calls it can handle (ps), and a list of the events (e) observed so far. We define the lifting of a boolean c to a heap predicate $\langle c \rangle$ as $\lambda\, s.\ s = \emptyset \wedge c$. Let $l \mapsto v$ denote the heap predicate $\lambda\, h.\ h = \{\ \mathsf{Mem}\ l\ v\ \}$.

The *result* of executing a program is modelled by an element of the datatype res:

$$\mathsf{res}\ =\ \mathsf{Val\ v}\ |\ \mathsf{Exn\ v}\ |\ \mathsf{FFIDiv\ string\ (word8\ list)\ (word8\ list)}\ |\ \mathsf{Div\ (io\_event\ llist)}$$

Programs can return a value (Val), raise an exception (Exn), invoke a foreign function that never returns control to CakeML (FFIDiv), or diverge (Div). When a program diverges, it exhibits a possibly infinite trace of I/O events, represented as a lazy list. Div and FFIDiv are where we extend previous work [15], which only considered values and exceptions. We will focus the presentation on the case of Div.

When we write a Hoare triple $\{\!|P|\!\}\ p\ \{\!|Q|\!\}$, the precondition P is a heap predicate, and the postcondition Q is a function from results to heap predicates. The following abbreviations are convenient for writing postconditions:

$$(\mathsf{POSTv}\ v.\ Q\ v) \stackrel{\mathsf{def}}{=} (\lambda\, res.\ \mathsf{case}\ res\ \mathsf{of}\ \mathsf{Val}\ v\ \Rightarrow\ Q\ v\ |\ \_\ \Rightarrow\ \langle \mathsf{F} \rangle)$$
$$(\mathsf{POSTd}\ io.\ Q\ io) \stackrel{\mathsf{def}}{=} (\lambda\, res.\ \mathsf{case}\ res\ \mathsf{of}\ \mathsf{Div}\ io\ \Rightarrow\ \langle Q\ io \rangle\ |\ \_\ \Rightarrow\ \langle \mathsf{F} \rangle)$$

For example, the Hoare triple $\{\!|\langle \mathsf{T} \rangle|\!\}\ p\ \{\!|\mathsf{POSTv}\ v.\ \langle \mathsf{int}\ 5\ v \rangle * l \mapsto v|\!\}$ is true if from every initial state the program p returns 5 and, moreover, the memory location l contains 5. Here $*$ denotes separating conjunction. Note that in POSTd, $Q\ io$ is a predicate and not a heap predicate; this reflects the fact that divergent programs have no final state.

The characteristic formula is generated by a straightforward recursion on the syntactic structure of the program. To give the flavour, we show the characteristic formula of a sequential composition $e_1\ ;\ e_2$:

$$\begin{array}{l}\mathsf{cf}\ (e_1\ ;\ e_2) \stackrel{\mathsf{def}}{=} \\ \quad \mathsf{local} \\ \quad\quad (\lambda\, H\ Q. \\ \quad\quad\quad \exists\, Q'. \\ \quad\quad\quad\quad (\mathsf{cf}\ e_1\ H\ Q' \wedge Q' \Rightarrow_{\neg v} Q) \wedge \\ \quad\quad\quad\quad \forall\, xv.\ \mathsf{cf}\ e_2\ (Q'\ (\mathsf{Val}\ xv))\ Q)\end{array}$$

[2] In our case, the meta-logic is higher-order logic.

local is, intuitively, the closure of a predicate under separating conjunction. This mimics the frame rule of separation logic, allowing us to disregard irrelevant parts of the heap in subproofs. The remainder of the formula says that there must be an intermediate postcondition Q' admitted by the characteristic formula of e_1 such that (1) Q' *res* implies Q *res* if *res* is not a value ($\Rightarrow_{\neg v}$), and (2) the characteristic formula of e_2 admits Q' (Val xv) as precondition and Q as postcondition for all values xv. To see how this formula copes with divergence, note that if Q' is a POSTd then conjunct (2) is vacuous because of the precondition Q' (Val xv) = $\langle$F$\rangle$. Thus, if e_1 diverges then cf (e_1 ; e_2) does not depend on e_2.

Perhaps surprisingly, virtually no changes to the definition of cf are needed to support divergence. This is for two reasons. First, CF for CakeML already supports reasoning about exceptions. Once obtained, the way a Div result propagates through a characteristic formula is exactly analogous to an exception that can never be handled; the reader may check this for the case when e_1 raises an exception in the above sequential composition. The second reason is that cf e does not unfold the definition of functions that are called within e. Instead, the cf of a function application falls back to a Hoare triple about the program:

$$\mathsf{cf}\ (f \cdot v) \stackrel{\text{def}}{=} \mathsf{local}\ (\lambda H\ Q.\ \{\!|H|\!\}\ f \cdot v\ \{\!|Q|\!\})$$

Thus there is no need to accommodate infinite recursion in e by, say, making cf clocked or corecursive. This design means that the CF logic has no native proof rule to deal with recursive calls, terminating or not. This aspect is handled entirely by the meta-logic, which, being higher-order, offers excellent support for induction.

In the above presentation, we have taken the liberty of abstracting away from certain details that are not germane to the issue at hand. In particular, the definition of cf in the formalisation is parameterised by a binding environment mapping variables to values. We will continue to ignore binding environments in the remainder of this presentation. This is because they are mainly a matter of plumbing: the CF user never sees or manipulates binding environments. Readers interested in the gory details may peruse [15] or the formalisation.

3.2 Semantics and soundness

In this section, we will define how we give meaning to Hoare triples, and prove that characteristic formulae are sound with respect to Hoare triples.

The semantics of CakeML is defined in the style of *functional big-step semantics* [30]. That is, the workhorse is a function evaluate which given an initial state and a program returns a final state and a result. It is structured much like an interpreter, but is not necessarily executable. Since evaluate is a function, we need to make sure it is terminating, and since we also wish to give semantics to non-terminating programs, evaluate is *clocked*: it is parameterised on a natural number *ck* that is decremented whenever evaluate consumes a function call, and if *ck* is 0, evaluate terminates with a special timeout result.[3] The top-level semantics of a program is defined in terms of evaluate by quantifying over possible clock values: a diverging program is one that times out for every *ck*. A terminating program is one where for some *ck*, evaluate terminates with a non-timeout result. This intuition is formalised in the definition of evaluate_to_heap:

[3] In the CakeML language, (recursive) function calls are the only language constructs that can lead to divergence. There are no while loops or similar constructs.

$$\begin{array}{l}\mathsf{evaluate\_to\_heap}\ st\ exp\ heap\ (\mathsf{Val}\ v) \stackrel{\mathsf{def}}{=} \\ \quad \exists\, ck\ st'.\ \mathsf{evaluate}\ ck\ st\ [exp] = (st', \mathsf{Rval}\ [v]) \wedge \mathsf{st2heap}\ st' = heap\end{array}$$

$$\begin{array}{l}\mathsf{evaluate\_to\_heap}\ st\ exp\ heap\ (\mathsf{Div}\ io) \stackrel{\mathsf{def}}{=} \\ \quad (\forall\, ck.\ \exists\, st'.\ \mathsf{evaluate}\ ck\ st\ [exp] = (st', \mathsf{Rerr}\ (\mathsf{Rabort}\ \mathsf{Rtimeout\_error}))) \wedge \\ \quad \mathsf{sup}\ \{\ io\ |\ \exists\, ck.\ io = \mathsf{fromList}\ (\mathsf{fst}\ (\mathsf{evaluate}\ ck\ st\ [exp])).\mathsf{ffi.io\_events}\ \}\ = io\end{array}$$

As a technical detail, evaluate_to_heap also mediates between evaluate's concrete notion of state, and the heap abstraction that our Hoare triples use, via st2heap. Two noteworthy things are happening in the divergence case. First we require that io is the supremum (ordered by prefix inclusion) of the I/O events that the program emits for every clock value. Thus, a Div result represents the limit behaviour of a program as time goes to infinity. Second, the value of *heap* is ignored, because a divergent execution has no final state.

We now have all the machinery required to define Hoare triples in terms of this semantics:

$$\begin{array}{l}\{\!|H|\!\}\ e\ \{\!|Q|\!\} \stackrel{\mathsf{def}}{=} \\ \quad \forall\, st\ h\_i\ h\_k. \\ \qquad \mathsf{split}\ (\mathsf{st2heap}\ st)\ (h\_i, h\_k) \Rightarrow \\ \qquad H\ h\_i \Rightarrow \\ \qquad \exists\, r\ h\_f\ h\_g\ heap.\ \mathsf{split3}\ heap\ (h\_f, h\_k, h\_g) \wedge Q\ r\ h\_f \wedge \mathsf{evaluate\_to\_heap}\ st\ e\ heap\ r\end{array}$$

In words, the Hoare triple $\{\!|H|\!\}\ e\ \{\!|Q|\!\}$ is true if, starting from any initial state which is the disjoint union (split) of a heap h_k and some heap that satisfies the precondition H, the result of evaluating e from this initial state is a heap *heap* and result r such that some subset of the *heap* which is disjoint from h_k satisfies $Q\ r$. Note that h_k recurs in both the pre- and postconditions – this, along with the disjoint unions before and after evaluation, are necessary to make local reasoning sound.

Soundness: the main result that validates the use of characteristic formulae for verification of CakeML programs is:

$$\vdash \mathsf{cf}\ e\ H\ Q \Rightarrow \{\!|H|\!\}\ e\ \{\!|Q|\!\}$$

This extends the soundness result of Guéneau et al. [15] to total-correctness Hoare triples about divergent programs. The main complications were the shifted clocks in the sampling of sup as used in the definition of evaluate_to_heap. The interesting cases to update for Div were `let`, `handle`, `andalso`/`orelse` and function application. Otherwise, the structure of the soundness proof needed some refactoring but did not fundamentally change.

4 Reasoning about divergent programs

When faced with programs that run forever, the traditional techniques for reasoning about loops no longer apply: induction fails because there is no base case, and the While rule of total correctness Hoare logic fails because there is no loop variant. Both of these approaches have the very important practical benefit that they are syntax-directed: they reduce reasoning about loops to reasoning about a single iteration of the loop body.

In this section, we develop the reasoning principles and tools necessary to support such syntax-directed proofs about divergent programs. In doing so, we have two conflicting goals that we must balance:

1. We want to support reasoning about silent loops that don't produce I/O.
2. The user should never see the semantic clock from Section 3.2 when proving a specification, and postconditions should describe no behaviour except the observable I/O behaviour.

The challenge is to meet both while avoiding unsoundness due to circularity. For example, the WHILE rule of Nakata and Uustalu [28] meets the first goal by sacrificing the second: their postconditions describe traces that include internal computation steps such as the evaluation of loop guards. Programming with corecursive functions in proof assistants or total programming languages [37] requires productivity, thus sacrificing the first goal.

Our solution is based on the insight that we can avoid circularity by exploiting the fact that in the evaluation semantics of Section 3.2, silent loops produce a clock tick every iteration. We can hide this tick from the user by considering programs encapsulated in a context that causes clock ticks to happen. Hence we consider programs of the form `repeat f x`, where

```
fun repeat f x = repeat f (f x);
```

We derive a reasoning principle, akin to transfinite induction, for proving POSTd specifications about such calls to `repeat`. This reasoning principle is syntax-directed in the sense that the premises talk only about the behaviour of the function argument `f`. In context, each invocation of `f` is interleaved with a recursive call to `repeat`, which produces the required clock tick.

The `repeat` form allows us to derive a sound and usable reasoning principle, but we do not want the straitjacket of having to write all our code in `repeat` form. Fortunately, there is no need. The key insight is that for every divergent tail-recursive function `f`, there exists a function `g` such that `f` and `repeat g` are semantically equivalent. For example, `yes` can be expressed in `repeat` form as follows:

```
fun yes() = repeat (fn x => put_line "y"; ()) ();
```

We implement and verify a program transformation that given a tail-recursive function produces a function in `repeat` form that satisfies the same POSTd specification. Thus, we can reduce reasoning about arbitrary divergent function calls to calls on `repeat` form, for which we have a sound reasoning principle. The reason we restrict attention to tail-recursive functions is that only functions consisting of tail-calls can truly diverge: any other program will eventually run out of stack space. Tail-calling programs, on the other hand, can run forever without exhausting the stack.

All these concerns are hidden from the user by a custom tactic that evaluates the program transformation in-logic and applies the transfinite induction principle to the resulting `repeat` program. The user may go about her business of verifying divergent programs without ever being exposed to the semantic clock, the `repeat` function, or the program transformation into `repeat` form.

In the remainder of this section, we will describe the induction principle and the program transformation in more detail.

4.1 An induction principle for divergence

Our reasoning principle for programs in `repeat` form is shown in Figure 1. In order to conclude that executing `repeat fv xv` from an initial state satisfying H results in a stream of I/O events satisfying Q using this rule, we must perform an argument by transfinite induction: we must discharge a base case, a successor case and a limit case.

The first conjunct limited_parts ns is a side condition which means, roughly, that ns is the list of all FFI calls that can occur in the program under consideration. Without this restriction, we could make only very limited predictions about the final I/O stream: since separation logic pre- and postconditions are local, we would have to account for the possibility that the frame includes others FFI_parts with other (possibly infinite) event streams that we have no information about.

$$
\begin{array}{l}
\vdash \mathsf{limited\_parts}\ ns \wedge \\
\quad (\exists\, Hs\ events\ vs\ ss\ u. \\
\qquad vs\ 0\ xv \wedge H \Rightarrow Hs\ 0 * \mathsf{one}\ (\mathsf{FFI\_part}\ (ss\ 0)\ u\ ns\ (events\ 0)) \wedge \\
\qquad (\forall\, i\ xv. \\
\qquad\quad vs\ i\ xv \Rightarrow \\
\qquad\quad \{\!|Hs\ i * \mathsf{one}\ (\mathsf{FFI\_part}\ (ss\ i)\ u\ ns\ [])|\!\} \\
\qquad\quad\ fv \cdot [xv] \\
\qquad\quad \{\!|\mathsf{POSTv}\ v'. \\
\qquad\quad\quad \langle vs\ (i+1)\ v'\rangle * Hs\ (i+1) * \\
\qquad\quad\quad \mathsf{one}\ (\mathsf{FFI\_part}\ (ss\ (i+1))\ u\ ns\ (events\ (i+1)))|\!\}) \wedge \\
\qquad Q\ (\mathsf{lflatten}\ (\mathsf{lgenlist}\ (\mathsf{fromList} \circ events)\ \mathsf{None}))) \Rightarrow \\
\{\!|H|\!\}\ \mathsf{repeat} \cdot [fv;\ xv]\ \{\!|\mathsf{POSTd}\ Q|\!\}
\end{array}
$$

Figure 1 Transfinite induction principle for proving POSTd specifications.

The base and successor cases require the user to exhibit a number of streams (represented as functions with domain num), where the i:th element of the streams describe the state after executing the function fv i times. Hs i is a heap predicate that holds after i iterations, $events$ i is the list of I/O produced by the i:th loop iteration, ss i the state of the FFI interface, and vs i a value predicate that $fv^i x$ satisfies.

In the base case, we must show that vs and Hs are true initially; this corresponds to the conjuncts $vs\ 0\ xv$ and $H \Rightarrow Hs\ 0$.

In the successor case, we must discharge a Hoare triple which intuitively states that doing one more iteration of fv respects the streams. Specifically, if we invoke fv with value and heap respectively satisfying vs i and Hs i, and with initial FFI state ss i, then fv terminates with a value and heap satisfying $vs\ (i+1)$ and $Hs\ (i+1)$, producing the I/O events $events\ (i+1)$ and reaching the FFI state ss i. Note that the event list starts out empty: this allows reasoning about each loop iteration that is independent of the I/O history from previous loop iterations.

In the limit case, we need to show that the least upper bound of $events$ – or in other words, the I/O events after infinitely many iterations of fv – satisfies Q. This upper bound has an explicit characterisation, namely the infinite concatenation of $events\ 0$, $events\ 1$, and so on, which is expressed by lflatten (lgenlist ... ...).

The intermediate heaps and values are not used in the limit case: the only relevant aspect is the I/O events. Since Q is a predicate on lazy lists, and since in HOL4 equality on lazy lists coincides with list bisimilarity, discharging this case tends to involve coinductive proofs via list bisimilarity.Hence our technique for verifying diverging programs uses a mix of transfinite induction and coinduction. The historically-minded reader may note that our limit case is similar to the admissibility side condition of Scott induction [34], where the predicate being proved must be closed under supremum.

4.2 Program transformation

To use the induction principles discussed in the previous section to verify a function f, we must first rewrite f into `repeat` form. That is, we must exhibit a function g such that if `repeat` g diverges, f diverges with the same result. In this section, we describe the program transformation we use to produce this g.

```
make_single_app fname allow_fname (e1 ; e2) ≝
 do
  e1' ← make_single_app fname F e1;
  e2' ← make_single_app fname allow_fname e2;
  Some (e1' ; e2')
 od

make_single_app fname allow_fname (f · x) ≝
 if Some f = fname then
  do assert allow_fname; make_single_app fname F x od
 else
  do
   x' ← make_single_app fname F x;
   if allow_fname then Some (then_tyerr (f · x'))
   else Some (f · x')
  od
```

Figure 2 Excerpts from the definition of the repeat program transformation.

We restrict attention to tail-recursive functions f which take a single input argument. The basic idea for how to produce g is simple: the body of g should be the body of f, but with every recursive call $f\ x$ replaced with x. To make the transformation sound, we need to muddy the basic idea with two minor complications. The first is to deal with shadowing carefully: if the function's name is shadowed by `let` bindings, occurrences of the function's name in this scope should obviously not be treated as recursive calls. Second, and more interestingly, what if f terminates? Consider this function:

```
fun condLoop x = if x = 0 then 0 else condLoop (x - 1);
```

If we were to naively rewrite its body as

```
fun condLoop' x = if x = 0 then 0 else x - 1;
```

we would lose soundness: it is easy to see that `repeat condLoop' 0` diverges but `condLoop 0` terminates. To avoid this problem, the transformation makes sure that whenever an expression is encountered in tail position that is *not* a tail call – like the expression `0` in the `if`-branch above – it is replaced with an expression that causes a runtime error.[4] With this modification, evaluation of `repeat condLoop' 0` gets stuck rather than diverges. This preserves soundness, because every Hoare triple is false for a program that gets stuck. It is true that this is not the same behaviour as `condLoop 0`, but that's fine: the transformation is only ever used to prove POSTd specifications, so the two programs only need to agree on divergent behaviour. Thus, to show that `condLoop (~1)` diverges it suffices to show that `repeat condLoop' (~1)` diverges: the `else` branch is always taken, so the runtime error never happens.

While the full definition is too big to show, Figure 2 shows representative extracts from make_single_app, the workhorse of the transformation. Given an expression e corresponding to the body of a function named *fname*, it produces the body of the transformed function.

[4] The expression we use is `ord 0`, which is not type correct because `ord` expects a character as input.

It is written in the option monad because the transformation may fail if e.g. the function is not tail-recursive. To deal with variable capture, *fname* is an option; the idea is that if the name of the function is shadowed, *fname* is None. *allow_fname* is a flag which is T if the expression under consideration is in tail position; this is used to determine whether to inject runtime errors or not. then_tyerr adds an expression which causes runtime errors to another expression.

The main result of this section is that the above is a sound technique for establishing POSTd specifications:

$$\vdash \mathsf{make\_repeat\_closure}\ fv = \mathsf{Some}\ gv \wedge \mathsf{wellformed}\ fv \wedge \\ \{\!|H|\!\}\ gv \cdot x\ \{\!|\mathsf{POSTd}\ Q|\!\} \Rightarrow \\ \{\!|H|\!\}\ fv \cdot x\ \{\!|\mathsf{POSTd}\ Q|\!\}$$

Here make_repeat_closure is the main entry point for the transformation, which lifts make_single_app from function bodies to function closures. It returns a new closure value gv, which is a function of the form `repeat g` for some `g`. A closure value is wellformed if it is not mutually recursive and the function name is distinct from the argument name (this precludes eg. `fun f f = f`).

The proof is tedious and ugly because it is done directly in terms of the CakeML semantics and not in CF. Large parts of it are focused on massaging binding environments and semantic clocks to line up in highly specific ways. To put it another way, the proof consists of exactly the kind of low-level reasoning that we want CF to abstract away from. Doing it here, once and for all, means that when a CF user verifies a diverging program, she won't have to.

We conclude this section by discussing some limitations of our program transformation. Recall that we restrict ourselves to tail-recursion. We do not consider functions with multiple (curried) arguments, nor do we consider mutual recursion. Extensions to handle both should be straightforward, if tedious, to implement; we have not yet done so because for the programs we are interested in verifying, the need has not arisen. One possibility is to add further program transformations on top, encoding curried arguments as tupled arguments and mutual recursion as direct recursion over sum types. It is also worth noting that the proof rule from Section 4.1 is not built into the CF infrastructure, but derived from it. Hence a possible direction for future work is to derive further proof rules covering more exotic forms of recursion, such as recursion through the store.

5 Examples

In this section, we present a number of example program verifications with the intention to showcase various features of our program logic.

Silent loop

Our first example is a function that just calls itself:

```
fun pureLoop x = pureLoop x;
```

This example illustrates that we can reason about loops without I/O, and that the shortest possible divergent program is trivial to verify – the proof script is four lines. The specification we prove is the following:

$$\vdash \mathsf{limited\_parts}\ ns \Rightarrow \\ \{\!|\mathsf{one}\ (\mathsf{FFI\_part}\ s\ u\ ns\ [])|\!\}\ \mathsf{pureLoop} \cdot [xv]\ \{\!|\mathsf{POSTd}\ io.\ io = [\!|\,|\!]\}$$

After applying the tactic described in Section 4, the user must exhibit streams of heap predicates, value predicates and events that describe the state at the nth iteration of the loop body, which in this case is `fn x => x`. We instantiate these variables with the constant functions that return, respectively, emp, $\lambda\, x.$ T and []. The remaining two lines are to prove that `fn x => x` does nothing, and that flattening the infinite list of empty lists is $[\![]\!]$.

Conditional divergence

In this section, we revisit the following example from Section 4.2:

```
fun condLoop x = if x = 0 then 0 else condLoop (x - 1);
```

The point here is to illustrate how to prove specifications about programs that may either terminate or diverge. In this case, the specification is the following:

$$
\begin{array}{l}
\vdash \mathsf{limited\_parts}\ ns \wedge \mathsf{int}\ x\ xv \Rightarrow \\
\quad \{|\mathsf{one}\ (\mathsf{FFI\_part}\ s\ u\ ns\ [])|\} \\
\quad\ \mathsf{condLoop} \cdot [xv] \\
\quad \{|\mathsf{POSTvd} \\
\quad\ (\lambda\, v.\ \langle 0 \leq x \wedge \mathsf{int}\ 0\ v\rangle * \mathsf{one}\ (\mathsf{FFI\_part}\ s\ u\ ns\ [])) \\
\quad\ (\lambda\, io.\ x < 0 \wedge io = [\![]\!])|\}
\end{array}
$$

where POSTvd Q_1 Q_2 abbreviates the disjunction of POSTv Q_1 and POSTd Q_2. Note that the POSTv includes the conditions under which the program terminates ($0 \leq x$), and vice versa for the POSTd part.

The proof proceeds by a case split on whether x is negative. If it is, the POSTvd condition is equivalent to POSTd $io.\ io = [\![]\!]$. From there, the proof is similar to `pureLoop`, with one added step: we must show that the loop maintains the invariant that x is negative.

If x is non-negative, the POSTvd condition is equivalent to the POSTv part. The rest of the proof proceeds by induction on x.

This proof strategy – case splitting on the conditions under which divergence or termination holds – is usually a forced move on the part of the user. An unfortunate side-effect of this is situations where reasoning about the loop body may be duplicated in the POSTv and POSTd cases, but not reusable across them. In practice, this issue is mostly obviated by factoring out code into auxiliary functions, whose specifications will be automatically applied in both the POSTv and POSTd cases. A pragmatic reason for preferring this state of affairs is backwards compatibility: there are already substantial case studies and infrastructure built on top of CF for terminating CakeML programs [12, 17], and since we keep reasoning about divergence separate, there is no need for them to change.

Input and output

In Guéneau et al. [15], CF for CakeML was used to develop and verify an implementation of the Unix `cat` utility; this was later extended to a more efficient implementation on a more realistic file system model [12]. Both developments share a limitation: they use a file system model where the contents of every file and standard stream (eg. `stdin`) can only be finite. Thus the theorems about them are not meaningful in situations with infinite input, such as `cat /dev/zero`, or `yes | cat`, or even just Unix `cat`.[5]

[5] `/dev/zero` is an infinite stream of null characters. Unix `cat` with no arguments will read from `stdin`.

In this section, we will show how to lift this limitation. File system modelling is not the topic of the present paper, so in order to avoid getting lost in file system details, we consider only the case where we read from `stdin` and write to `stdout`. Our example is:

```
fun catLoop (u:unit) = case get_char () of
    None   => ()
  | Some c => (put_char c; catLoop ());
```

In the following Hoare triple, SIO *input events* abbreviates a heap predicate which states that an FFI_part that can read from `stdin` and write to `stdout` is present. Here *input* is a lazy list of characters yet to be read from `stdin`, and *events* is the list of I/O events so far. This allows *input* to be infinite, which lifts the aforementioned limitation of the previous work [15, 12]. Interaction with the standard streams is encapsulated by `get_char` and `put_char`, which are (verified) CakeML library functions that make FFI calls to the corresponding `stdlib` functions, and do the necessary marshalling and unmarshalling.

The function cat abbreviates the I/O we expect to see for a character stream ll:

$$\mathsf{cat}\ ll \stackrel{\mathsf{def}}{=} \mathsf{lflatten}\ (\mathsf{lmap}\ (\lambda\, c.\ [\![\mathsf{get\_char\_event}\ c;\ \mathsf{put\_char\_event}\ c]\!])\ ll)$$

The Hoare triple which specifies the whole function is this:

$$\begin{array}{l}
\vdash \mathsf{limited_parts\ names} \Rightarrow \\
\quad \{|\mathsf{SIO}\ \mathit{input}\ []|\} \\
\quad\ \mathsf{catLoop} \cdot [\mathit{uv}] \\
\quad \{|\mathsf{POSTvd} \\
\quad\ (\lambda\, v. \\
\qquad \langle \mathsf{finite}\ \mathit{input} \wedge \mathsf{unit_type}\ ()\ v\rangle * \\
\qquad \mathsf{SIO}\ [\![]\!]\ (\mathsf{snoc\ get_char_eof_event}\ (\mathsf{the}\ (\mathsf{toList}\ (\mathsf{cat}\ \mathit{input}))))) \\
\quad\ (\lambda\, \mathit{io}.\ \neg\mathsf{finite}\ \mathit{input} \wedge \mathit{io} = \mathsf{cat}\ \mathit{input})|\}
\end{array}$$

As with the `condLoop` example, the postcondition is in POSTvd form. It will either terminate or diverge, depending on whether *input* is finite or not. If it is finite, we return unit, consume all pending inputs from SIO, and produce the expected sequence of I/O events, with a final EOF event corresponding to the failed `get_char` when *input* is empty. If *input* is infinite, the I/O events are cat *input*. The whole proof is around 90 lines of HOL script.

Traversing cyclic pointer structures

We now consider an example that combines divergence with separation logic-style reasoning about the shape of memory. Here we will traverse a cycle of cons cells containing characters on the heap, and print each character we encounter. The code is as follows:

```
fun pointerLoop c =
  case !c of (a,b) =>
    (put_char a; pointerLoop b);
```

As an aside, the reader may notice that, in standard Hindley–Milner type systems, this program has no type: it requires `c` to have a type `'a` such that `'a = (char * 'a) ref`. That's fine since CakeML's raw evaluation semantics is untyped, and so the only purpose of the type system is to establish the absence of a certain class of runtime errors. Here, we establish this absence by proving Hoare triples instead.

We use the heap predicate ref_list to describe pointer cycles:

$$\mathsf{ref\_list}\ rv\ []\ A\ [] \stackrel{\mathsf{def}}{=} \exists\ loc.\ \langle rv = \mathsf{Loc}\ loc\rangle$$

$$\mathsf{ref\_list}\ rv\ (rv_2{::}rvs)\ A\ (x{::}l) \stackrel{\mathsf{def}}{=}$$
$$\exists\ loc\ v_1.\ \langle rv = \mathsf{Loc}\ loc\rangle * loc \mapsto (v_1,\ rv_2) * \langle A\ x\ v_1\rangle * \mathsf{ref\_list}\ rv_2\ rvs\ A\ l$$

The idea is that ref_list rv rvs A xs describes an encoding of a list segment with elements xs of type A, where rv is a pointer to the memory location where this encoding resides, and rvs are pointers to the encodings of the tails. Note that there is no indication on the heap of where the segment ends; rather, the last pointer of rvs is left dangling. A cyclic lazy list, whose elements are those of xs over and over, is represented by a heap predicate ref_list rv (snoc rv rvs) A xs where the last pointer points back to the beginning. This predicate allows a concise specification of `pointerLoop`:

$$\begin{array}{l}
\vdash \mathsf{limited_parts\ names} \Rightarrow \\
\quad \{|\mathsf{SIO}\ [|]\ []\ *\ \mathsf{ref_list}\ rv\ (\mathsf{snoc}\ rv\ rvs)\ \mathsf{char}\ l|\} \\
\quad\ \mathsf{pointerLoop} \cdot [rv] \\
\quad \{|\mathsf{POSTd}\ io.\ io = \mathsf{lmap\ put_char_event}\ (\mathsf{lrepeat}\ l)|\}
\end{array}$$

In the successor case, the proof uses the fact that the ref_list predicate satisfies a kind of rotational symmetry – intuitively, any tail of a cyclic list with cycle xs is a cyclic list whose cycle is a rotation of xs. In the limit case, we use bisimulation up-to context [33] to reduce the size of the candidate relation to one pair only.

Verifying repeat with repeat

We now turn to a question of meta-verification: can the `repeat` construct described in Section 4.2 be used to verify `repeat` itself? For trivial syntactic reasons, the immediate answer is no: `repeat` is curried, and the transformation only considers one-argument functions. However, the answer changes if we allow ourselves to consider an uncurried version:

```
fun myRepeat (f,r) = myRepeat(f,f(r))
```

For such a function, we can easily (in just 8 lines) prove the following specification.

$$\begin{array}{l}
\vdash \mathsf{limited_parts}\ ns \Rightarrow \\
\quad \{|H\ * \\
\quad \langle vs\ 0\ xv \wedge H \Rightarrow Hs\ 0\ *\ \mathsf{one}\ (\mathsf{FFI_part}\ (ss\ 0)\ u\ ns\ (events\ 0)) \wedge \\
\quad (\forall\ i\ xv. \\
\qquad vs\ i\ xv \Rightarrow \\
\qquad \{|Hs\ i\ *\ \mathsf{one}\ (\mathsf{FFI_part}\ (ss\ i)\ u\ ns\ [])|\} \\
\qquad\ fv \cdot [xv] \\
\qquad \{|\mathsf{POSTv}\ v'. \\
\qquad\quad \langle vs\ (i+1)\ v'\rangle * Hs\ (i+1)\ * \\
\qquad\quad \mathsf{one}\ (\mathsf{FFI_part}\ (ss\ (i+1))\ u\ ns\ (events\ (i+1)))|\}) \wedge \\
\quad Q\ (\mathsf{lflatten}\ (\mathsf{lgenlist}\ (\mathsf{fromList} \circ events)\ \mathsf{None}))\rangle|\} \\
\quad\ \mathsf{myRepeat} \cdot [(fv,\ xv)] \\
\quad \{|\mathsf{POSTd}\ io.\ Q\ io|\}
\end{array}$$

Note that the preconditions of the Hoare triple above are essentially the same as the assumptions of the induction principle from Figure 1. In other words, we have given `repeat` a CF specification by applying the `repeat` transformation to `repeat` itself (modulo currying).

Listing 1 Excerpts from the filter source code.

```
fun forward_loop inputarr =
    (#(accept_call) "" inputarr;
     let val ln = Word8Array.substring inputarr 0 256;
         val ln' = cut_at_null ln;
     in
       if match_string ln' then
         #(emit_string) ln' dummyarr
       else ()
     end;
     forward_loop inputarr);

fun forward_matching_lines u =
    let val inputarr = Word8Array.array 256 (Word8.fromInt 0);
    in
      forward_loop inputarr
    end
```

6 Case study: verified filter components

In this section we describe the application of the techniques developed in this paper to a case study: the development of verified architectural components for systems built on the formally verified seL4 microkernel [21]. The particular domain we consider is unmanned aerial vehicles (UAVs), but the techniques can be applied to other systems too. The case study itself is the topic of another paper [35].

The particular component we consider is a filter. Architecturally, the filter sits between a radio driver, which receives commands from a ground station, and the rest of the UAV's flight control subsystem. Its purpose is (a) to protect the rest of the flight control subsystem from cyber-attacks based on malformed command messages, and to achieve this in a way that (b) does not require changing legacy components, (c) does not increase the attack surface of the overall system, and (d) does not prevent the rest of the system from fulfilling its mission.

Note that while (a) is a safety property, (d) is a liveness property: it requires that beyond rejecting malformed messages, the filter must never cause a well-formed message to be dropped. The precise definition of "well-formed" will of course vary; here we are interested in properties that can be decided by checking membership in a regular language $\mathcal{L}$.

An excerpt of the filter implementation is shown in Listing 1. Here `match_string` is a CakeML function that decides membership in $\mathcal{L}$. The function `forward_loop` will repeatedly invoke (via FFI) `accept_call`, which will receive a message from the radio driver via remote procedure call and write it to the buffer `inputarr`. If the contents of `inputarr` up until the first null terminator satisfies $\mathcal{L}$, we forward it to the flight controller, again via FFI (`emit_string`). The FFI calls are connected to seL4's RPC mechanism.

The theorem that states the desired liveness property is the following. In words, if *input* is an infinite stream of null-terminated strings of at most 256 characters[6], then `forward_matching_lines` will not terminate or abort (POSTd) and the messages it sends are precisely the inputs filtered by the language $\mathcal{L}$.

[6] The requirements on null-termination and message length show up as assumptions in this proof, but in practice they do not constitute attack vectors because they are enforced by our communication backend, namely the CAmkES component platform for seL4 [22].

```
⊢ llimited_parts ["accept_call"; "emit_string"] ∧ llength input = None ∧
  every null_terminated_w input ∧ every ((≥) 256 ∘ length) input ⇒
  {|seL4_IO input [] ∗ w8array dummyarr_loc []|}
   forward_matching_lines · [rv]
  {|POSTd io.
     lfilter is_emit io =
     lmap (output_event_of ∘ cut_at_null_w) (lfilter (ℒ ∘ cut_at_null_w) input)|}
```

Prior to this paper, the same liveness property was proved by Slind et al. [35] for the same program; the painful nature of those proofs was part of our motivation for extending CF with divergence. Having no verification framework at hand with support for divergence, the proofs were done directly in terms of the operational semantics (see Section 3.2). The result is proofs that spend inordinate amounts of energy massaging clocks and environments while carefully stepping through the interpretation of the program, e.g., unfold the definition of evaluate 11 times, then unfold some auxiliary definitions to find a particular value in the binding environment, then case split on whether we ran out of clock or not, then unfold evaluate 5 times, et cetera ad nauseam.

Redoing these proofs in CF, the results are more pleasant. At no point do clocks or binding environments enter into the proofs: instead, the granularity of proof steps is about the granularity of statements in the source program, with intermediate verification conditions generated at each step. Moreover, before deriving the equation about outputs in the POSTd condition above, Slind et al. [35] expend significant energy proving an explicit characterisation for the supremum of the I/O events. By using the induction principle from Figure 1 we get an explicit characterisation for free, so this effort is no longer necessary.

Besides the higher abstraction level, the proofs are shorter: the CF version of the theory that performs filter synthesis and verification comprises 1479 lines of HOL4, while the non-CF version is 1971 lines long. The former line count also includes infrastructure for lifting the filter's FFI model to CF's FFI abstraction.

In the CF version, we also derive a theorem from the specification above that gives the same liveness property directly in terms of the operational semantics, with no reference to CF abstractions such as heaps, FFI parts or Hoare triples:

```
⊢ llength input = None ∧ every null_terminated_w input ∧ every ((≥) 256 ∘ length) input ⇒
  ∃ events.
    semantics_prog … … [val () = forward_matching_lines ()] (Diverge events) ∧
    lfilter is_emit events =
    lmap (output_event_of ∘ cut_at_null_w) (lfilter (ℒ ∘ cut_at_null_w) input)
```

Here the elided arguments to semantics_prog are the program's initial state and environment.

The fact that we can prove theorems such as the one above means that our use of CF does not increase the trusted computing base. More importantly, it means our POSTd specification can be fed through CakeML's compiler correctness theorem [36] to obtain corresponding liveness theorems about the resulting binary (with the current caveat that the compiler correctness theorem allows the binary to exit early with an out-of-memory error, see Section 8).

7 Related work

The historical roots of characteristic formulae go back to the modal logic characterisation of bisimilarity by Hennessy and Milner [16]. Charguéraud's CFML work [5, 6] builds on this idea to develop a verification framework for impure functional programs. The CakeML CF framework [15] adapts these ideas for CakeML, and adds a mechanised soundness proof as well as support for exceptions and I/O [12]. Characteristic formulae have also been used to reason about complexity [9, 14], higher-order representation predicates [8], and read-only permissions [10].

Transfinite models have been used in program analysis in areas such as term rewriting [20] and program slicing [13]. In these cases program flow is modelled to continue after infinite loops, for the purpose of investigating how the loop affects succeeding computations. In our setting we are not interested in considering computations beyond infinite loops. As a result, we only need to consider the smallest infinite ordinal ω in our transfinite induction.

There is a large body of work on non-termination; most relevant to us are works that consider Hoare-like logics [18, 19, 11, 23, 24], coinduction [26, 1, 7, 4, 32], and interactive theorem proving [26]. We will focus the discussion on work that also treat I/O behaviour.

Nakata and Uustalu [27] introduce coinductive big-step semantics for a simple While language, formalised in Coq. They use coinductively defined state traces to reason uniformly about both termination and non-termination. In a follow-up paper [28] they define a Hoare logic for their big-step semantics, where postconditions describe state traces rather than a single final state. In another paper [29], they extend their semantics to handle I/O using resumptions. Resumptions can be thought of as coinductive trees that describe the I/O behaviour of all possible runs of a program. They do not extend their Hoare logic to this resumption semantics. In contrast, CakeML gives semantics to divergent programs not by coinduction, but by taking the limit of a clocked inductive semantics. An advantage of Nakata and Uustalu's approach is that it treats termination and non-termination uniformly, while we need to treat the two cases separately. On the other hand, this necessitates the introduction of silent actions (that do not correspond to I/O) into their traces, so that termination and silent divergence can be distinguished. The presence of silent actions lead, in turn, to observationally equivalent programs potentially exhibiting different traces. To recover observational equivalence, they can either consider traces up to termination-sensitive weak bisimilarity on the meta-level, or use one of two alternative semantics – one constructive and one classical – that do not produce silent actions. However, the constructive semantics fails to account for silent divergence, and the classical version does not treat termination and divergence uniformly. A more practical difference is that our Hoare logic considers I/O behaviour, and that CakeML is a much richer language than While.

Penninckx et al. [31] define a program logic for reasoning about I/O, where I/O events occur in the preconditions rather than the postconditions. These can be thought of as permission to do these events. Their assertion language is a separation logic where the heaps are Petri nets: the transitions are I/O events, and the nodes are analogous to our FFI states. For terminating programs, a Hoare triple can express that the right I/O events were performed in the right order by specifying which nodes have tokens in the postcondition. For a non-terminating program, the preconditions express an upper bound on the I/O events, but unlike our work, not necessarily a *least* upper bound. Hence they can prove safety but not liveness for non-terminating programs.

Ancona et al. [2, 3] have recently explored using corules and coaxioms – intuitively, auxiliary rules used to filter out judgements with undesired conclusions from infinite proof trees – to give semantics in terms of I/O traces for divergent executions in a lambda calculus

and a small Java-like language. The authors focus on semantics and do not develop a program logic, but they present an example verification similar to our `cat` example, albeit directly in terms of the operational semantics and with a more abstract treatment of I/O. Their work is not formalised in a proof assistant.

8 Conclusion

We have seen how characteristic formulae for CakeML, an existing verification framework for total correctness of impure terminating programs with I/O, can be extended to support liveness of non-terminating programs. The extension is non-invasive: existing proofs about terminating programs need not change at all. We support syntax-directed reasoning about loops, that reduces proofs about loops to proofs about the loop body. We support silent divergence without the need to involve clocks or special silent actions.

The framework is proven sound with respect to the CakeML semantics and thus integrated into the wider CakeML ecosystem, including in particular a verified optimising compiler [36]. Thus we can verify real programs, and reify our specifications to the machine code that runs them. Currently this comes with a caveat: liveness properties carry over to the binary only under the assumption that we do not run out of memory. The missing puzzle piece for unconditional liveness at the binary level is a means to discharge this assumption, which we are working towards by developing a verified space-cost semantics.

References

1. Davide Ancona. Soundness of Object-Oriented Languages with Coinductive Big-Step Semantics. In James Noble, editor, *Object-Oriented Programming (ECOOP)*. Springer, 2012. doi:10.1007/978-3-642-31057-7_21.
2. Davide Ancona, Francesco Dagnino, and Elena Zucca. Reasoning on divergent computations with coaxioms. *PACMPL*, 1(OOPSLA), 2017. doi:10.1145/3133905.
3. Davide Ancona, Francesco Dagnino, and Elena Zucca. Modeling Infinite Behaviour by Corules. In *Object-Oriented Programming (ECOOP)*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.ECOOP.2018.21.
4. Richard Bubel, Crystal Chang Din, Reiner Hähnle, and Keiko Nakata. A Dynamic Logic with Traces and Coinduction. In *Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*. Springer, 2015. doi:10.1007/978-3-319-24312-2_21.
5. Arthur Charguéraud. Program verification through characteristic formulae. In Paul Hudak and Stephanie Weirich, editors, *International Conference on Functional Programming (ICFP)*. ACM, 2010.
6. Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In *International Conference on Functional Programming (ICFP)*. ACM, 2011. doi:10.1145/2034773.2034828.
7. Arthur Charguéraud. Pretty-Big-Step Semantics. In *European Symposium on Programming (ESOP)*. Springer, 2013. doi:10.1007/978-3-642-37036-6_3.
8. Arthur Charguéraud. Higher-order representation predicates in separation logic. In *Certified Programs and Proofs (CPP)*, 2016. doi:10.1145/2854065.2854068.
9. Arthur Charguéraud and François Pottier. Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation. In *Interactive Theorem Proving (ITP)*, 2015. doi:10.1007/978-3-319-22102-1_9.
10. Arthur Charguéraud and François Pottier. Temporary Read-Only Permissions for Separation Logic. In *European Symposium on Programming (ESOP)*. Springer, 2017. doi:10.1007/978-3-662-54434-1_10.

11 Hong Yi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter W. O'Hearn. Proving Nontermination via Safety. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2014. doi:10.1007/978-3-642-54862-8_11.

12 Hugo Férée, Johannes Åman Pohjola, Ramana Kumar, Scott Owens, Magnus O. Myreen, and Son Ho. Program Verification in the Presence of I/O - Semantics, Verified Library Routines, and Verified Applications. In *Verified Software: Theories, Tools, Experiments (VSTTE)*, 2018. doi:10.1007/978-3-030-03592-1_6.

13 Roberto Giacobazzi and Isabella Mastroeni. Non-Standard Semantics for Program Slicing. *Higher-Order and Symbolic Computation*, 16(4), 2003. doi:10.1023/A:1025872819613.

14 Armaël Guéneau, Arthur Charguéraud, and François Pottier. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification. In *European Symposium on Programming (ESOP)*. Springer, 2018. doi:10.1007/978-3-319-89884-1_19.

15 Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. Verified Characteristic Formulae for CakeML. In Hongseok Yang, editor, *European Symposium on Programming (ESOP)*, volume 10201 of *LNCS*. Springer, 2017. doi:10.1007/978-3-662-54434-1_22.

16 Matthew Hennessy and Robin Milner. On Observing Nondeterminism and Concurrency. In J. W. de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming (ICALP)*, LNCS. Springer, 1980. doi:10.1007/3-540-10003-2_79.

17 Son Ho, Oskar Abrahamsson, Ramana Kumar, Magnus O. Myreen, Yong Kiam Tan, and Michael Norrish. Proof-Producing Synthesis of CakeML with I/O and Local State from Monadic HOL Functions. In *Automated Reasoning – International Joint Conference (IJCAR)*. Springer, 2018. doi:10.1007/978-3-319-94205-6_42.

18 Marieke Huisman and Bart Jacobs. Java Program Verification via a Hoare Logic with Abrupt Termination. In *Fundamental Approaches to Software Engineering (FASE)*, 2000. doi:10.1007/3-540-46428-X_20.

19 Bart Jacobs and Erik Poll. A Logic for the Java Modeling Language JML. In *Fundamental Approaches to Software Engineering (FASE)*, 2001. doi:10.1007/3-540-45314-8_21.

20 Richard Kennaway, Jan Willem Klop, M. Ronan Sleep, and Fer-Jan de Vries. Transfinite Reductions in Orthogonal Term Rewriting Systems. *Inf. Comput.*, 119(1), 1995. doi:10.1006/inco.1995.1075.

21 Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6), 2010. doi:10.1145/1743546.1743574.

22 Ihor Kuz, Yan Liu, Ian Gorton, and Gernot Heiser. CAmkES: A component model for secure microkernel-based embedded systems. *Journal of Systems and Software*, 80(5), 2007. doi:10.1016/j.jss.2006.08.039.

23 Ton Chanh Le, Cristian Gherghina, Aquinas Hobor, and Wei-Ngan Chin. A Resource-Based Logic for Termination and Non-termination Proofs. In *International Conference on Formal Engineering Methods (ICFEM)*, 2014. doi:10.1007/978-3-319-11737-9_18.

24 Ton Chanh Le, Shengchao Qin, and Wei-Ngan Chin. Termination and non-termination specification inference. In *Programming Language Design and Implementation (PLDI)*. ACM, 2015. doi:10.1145/2737924.2737993.

25 Xavier Leroy. A Formally Verified Compiler Back-end. *J. Autom. Reasoning*, 43(4), 2009. doi:10.1007/s10817-009-9155-4.

26 Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2), 2009. doi:10.1016/j.ic.2007.12.004.

27 Keiko Nakata and Tarmo Uustalu. Trace-Based Coinductive Operational Semantics for While. In *Theorem Proving in Higher Order Logics (TPHOLs)*. Springer, 2009. doi:10.1007/978-3-642-03359-9_26.

28 Keiko Nakata and Tarmo Uustalu. A Hoare Logic for the Coinductive Trace-Based Big-Step Semantics of While. In *European Symposium on Programming (ESOP)*. Springer, 2010. doi:10.1007/978-3-642-11957-6_26.

29 Keiko Nakata and Tarmo Uustalu. Resumptions, Weak Bisimilarity and Big-Step Semantics for While with Interactive I/O: An Exercise in Mixed Induction-Coinduction. In *Structural Operational Semantics (SOS)*, 2010. doi:10.4204/EPTCS.32.5.

30 Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional Big-Step Semantics. In Peter Thiemann, editor, *European Symposium on Programming (ESOP)*, LNCS. Springer, 2016. doi:10.1007/978-3-662-49498-1_23.

31 Willem Penninckx, Bart Jacobs, and Frank Piessens. Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs. In *European Symposium on Programming (ESOP)*. Springer, 2015. doi:10.1007/978-3-662-46669-8_7.

32 Casper Bach Poulsen and Peter D. Mosses. Flag-based big-step semantics. *J. Log. Algebr. Meth. Program.*, 88, 2017. doi:10.1016/j.jlamp.2016.05.001.

33 Davide Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8(5), October 1998. doi:10.1017/S0960129598002527.

34 Dana Scott and J.W. De Bakker. A Theory of Programs. Unpublished manuscript, IBM Vienna, 1969.

35 Konrad Slind, David S. Hardin, Johannes Åman Pohjola, and Michael Sproul. Synthesis of Verified Architectural Components for Autonomy Hosted on a Verified Microkernel. Draft, 2019.

36 Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29, 2019. doi:10.1017/S0956796818000229.

37 D. A. Turner. Total Functional Programming. *J. UCS*, 10(7), 2004. doi:10.3217/jucs-010-07-0751.