Characteristic Formulae for Liveness Properties of Non-terminating CakeML Programs

Johannes Åman Pohjola*
Henrik Rostedt‡
Magnus O. Myreen‡

*Data61/CSIRO, Sydney
*University of New South Wales, Sydney
‡Chalmers University of Technology, Gothenburg

ITP, Portland OR, September 12th 2019

y\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\n….

Johannes Åman Pohjola*
Henrik Rostedt‡
Magnus O. Myreen‡

*Data61/CSIRO, Sydney
*University of New South Wales, Sydney
‡Chalmers University of Technology, Gothenburg

ITP, Portland OR, September 12th 2019

```
while(true) do {
 print(“y\n”);
}
```

```
while(true) do {
 print(“y\n”);
}
```

Safety: yes never prints anything but y\n
Liveness: yes always eventually prints another y\n

Smashed together:

yes diverges, printing exactly an infinite stream of y\n

{true}

```
while(true) do {
  print(“y\n”);
}
```

{false}

Hoare Logic ~~Characteristic Formulae~~ for Liveness Properties of Non-terminating ~~CakeML~~ WHILE Programs

Johannes Åman Pohjola*
Henrik Rostedt‡
Magnus O. Myreen‡

*Data61/CSIRO, Sydney
*University of New South Wales, Sydney
‡Chalmers University of Technology, Gothenburg

ITP, Portland OR, September 12th 2019

Hoare Logic

~~Characteristic Formulae~~ for Liveness Properties of Non-terminating ~~CakeML~~

WHILE Programs

Oversimplifies the paper!

Two kinds of post-conditions

```
{P} e {Trm Q}

{P} e {Div S}
```

Two kinds of post-conditions

Bog-standard total correctness Hoare triple!

"If P holds initially, then e terminates in a state satisfying Q"

```
{P} e {Trm Q}
```

```
{P} e {Div S}
```

Two kinds of post-conditions

`{P} e {Trm Q}`

Bog-standard total correctness Hoare triple!

"If P holds initially, then e terminates in a state satisfying Q"

`{P} e {Div S}`

"If P holds initially, then e runs forever, producing a trace satisfying S"

Two kinds of post-conditions

`{P} e {Trm Q}`

Bog-standard total correctness Hoare triple!

“If P holds initially, then e terminates in a state satisfying Q”

State predicates

`{P} e {Div S}`

“If P holds initially, then e runs forever, producing a trace satisfying S”

Trace predicate

Two kinds of post-conditions

Bog-standard total correctness Hoare triple!

“If P holds initially, then e terminates in a state satisfying Q”

```
{P} e {Trm Q}
```

state => bool

```
{P} e {Div S}
```

“If P holds initially, then e runs forever, producing a trace satisfying Q”

io_event llist => bool

States

```
type state = heap x io_event list
```

State includes I/O history

{h=[]}

```
print(s)
```

{Trm(h=[print s])}

States

```
type state = heap x io_event list
```

Abbreviates λ(s,h). h=[]

{h=[]}

```
print(s)
```

{Trm(h=[print s])}

The WHILE rule

$$\frac{\begin{array}{l} P \Rightarrow I_0 \wedge h{=}h_0 \\ \forall i.\ \{I_i \wedge h{=}[]\}\ e\ \{\mathrm{Trm}(I_{i+1} \wedge h{=}h_{i+1})\} \\ S(h_o \cdot h_1 \cdot h_2 \cdot \ldots\ldots) \end{array}}{\{P\}\ \texttt{while(true) do e}\ \{\mathrm{Div}\ S\}}$$

The WHILE rule

Witnesses:

h_n I/O trace from the n:th loop iteration

I_n (internal) state after the n:th loop iteration

$$P \Rightarrow I_0 \wedge h=h_0$$

$$\forall i.\ \{I_i \wedge h=[]\}\ e\ \{\mathrm{Trm}(I_{i+1} \wedge h=h_{i+1})\}$$

$$S(h_0 \cdot h_1 \cdot h_2 \cdot \ldots\ldots)$$

$$\{P\}\ \texttt{while(true) do e}\ \{\mathrm{Div}\ S\}$$

The WHILE rule

$$\frac{\begin{array}{l} P \Rightarrow I_0 \wedge h{=}h_0 \\ \forall i.\ \{I_i \wedge h{=}[]\}\ e\ \{Trm(I_{i+1} \wedge h{=}h_{i+1})\} \\ S(h_o \cdot h_1 \cdot h_2 \cdot \ldots\ldots) \end{array}}{\{P\}\ \texttt{while(true) do e}\ \{Div\ S\}}$$

The WHILE rule

$$\frac{\begin{array}{l} P \Rightarrow I_0 \wedge h=h_0 \\ \forall i.\ \{I_i \wedge h=[]\}\ e\ \{\mathrm{Trm}(I_{i+1} \wedge h=h_{i+1})\} \\ S(h_o \cdot h_1 \cdot h_2 \cdot \ldots\ldots) \end{array}}{\{P\}\ \texttt{while(true) do e}\ \{\mathrm{Div}\ S\}}$$

The WHILE rule

$P \Rightarrow I_0 \wedge h{=}h_0$

$\forall i.\ \{I_i \wedge h{=}[]\}\ e\ \{Trm(I_{i+1} \wedge h{=}h_{i+1})\}$

$S(h_o{\cdot}h_1{\cdot}h_2{\cdot}\ldots..)$

$\{P\}$ `while(true) do e` $\{Div\ S\}$

Actually more like

$S(lflatten(lgenerate\ (\lambda n.\ h_n)))$

yes again

{h = []}

```
while(true) do {
  print(“y\n”);
}
```

{Div(λt. t = [print(“y\n”)]$^{\omega}$}

yes again

```
{h = []}
  while(true) do {
    print("y\n");
  }
```

$\{\mathrm{Div}(\lambda t.\ t = [\mathrm{print}(\text{“y\n”})]^{\omega}\}$

Proof: apply the WHILE rule with

$I_n = \lambda x.\ \mathrm{true}$

$h_0 = []$

$h_n = [\mathrm{print}(\text{“y\n”})]$ otherwise

In summary

😄 Conservative extension

😄 Existing proofs can be replayed verbatim

😄 Supports silent divergence

😄 No clocks, program counters or special silent actions

😱 Divergence and termination not treated uniformly

😱 No good story for internal non-determinism

😱 Incomplete (without extra structural rules)

c.f. [Nakata and Uustalu 2010]

Characteristic Formulae for Liveness Properties of Non-terminating CakeML Programs

Johannes Åman Pohjola*
Henrik Rostedt‡
Magnus O. Myreen‡

*Data61/CSIRO, Sydney
*University of New South Wales, Sydney
‡Chalmers University of Technology, Gothenburg

ITP, Portland OR, September 12th 2019

Yes, but...

\begin{itemize}

\item We don’t actually use Hoare Logic...

\item …but characteristic formulae (CF), and

\item CakeML doesn’t even have while!

\end{itemize}

The WHILE rule reloaded

$$\frac{\begin{array}{l} P \Rightarrow I_0 \land h{=}h_0 \\ \forall i.\ \{I_i \land h{=}[]\}\ e\ \{\mathrm{Trm}(I_{i+1} \land h{=}h_{i+1})\} \\ S(h_o \cdot h_1 \cdot h_2 \cdot \ldots\ldots) \end{array}}{\{P\}\ \texttt{while(true) do e}\ \{\mathrm{Div}\ S\}}$$

The WHILE rule reloaded

Separating conjunction

$$\frac{P \Rightarrow I_0 * h{=}h_0 \qquad \forall i.\ \{I_i * h{=}[]\}\ e\ \{\mathrm{Trm}(I_{i+1} * h{=}h_{i+1})\} \qquad S(h_o \cdot h_1 \cdot h_2 \cdot \ldots\ldots)}{\{P\}\ \texttt{while(true) do e}\ \{\mathrm{Div}\ S\}}$$

The WHILE rule reloaded

Expressions have return values

$$\frac{P \Rightarrow I_0 * h{=}h_0 \qquad \forall i.\ \{I_i * h{=}[]\}\ f(x_i)\ \{\mathrm{Trm}(x_{i+1})(I_{i+1} * h{=}h_{i+1})\} \qquad S(h_o \cdot h_1 \cdot h_2 \cdot \ldots\ldots)}{\{P\}\ \texttt{repeat f x}_o\ \{\mathrm{Div}\ S\}}$$

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

The WHILE rule reloaded

$$\frac{\begin{array}{l} P \Rightarrow I_0 * h{=}h_0 \\ \forall i.\ \{I_i * h{=}[]\}\ f(x_i)\ \{Trm(x_{i+1})(I_{i+1} * h{=}h_{i+1})\} \\ S(h_o \cdot h_1 \cdot h_2 \cdot \ldots\ldots) \end{array}}{\{P\}\ \texttt{repeat f x}_o\ \{Div\ S\}}$$

This is a theorem about Hoare triples, not a postulate.

Hoare triples are shallowly embedded,
defined in terms of CakeML's operational semantics

CF (Characteristic formulae)

Verification condition generator for impure functional programs [Charguéraud 2010]

Adapted to CakeML in previous work [Guéneau et al. 2016]

Workhorse function:

```
cf : exp ⇒ (state ⇒ bool) ⇒
     (result ⇒ state ⇒ bool) ⇒ bool
```

CF (Characteristic formulae)

Verification condition generator for impure functional programs [Charguéraud 2010]

Adapted to CakeML in previous work [Guéneau et al. 2016]

Workhorse function:

```
cf : exp ⇒ (state ⇒ bool) ⇒
     (result ⇒ state ⇒ bool) ⇒ bool
```

Intuition: to prove a Hoare triple

{P} e {Q}

it suffices to show

cf e P Q

CF (before)

```
Datatype result =
  Val v
| Exn v
```

CF (after)

```
Datatype result =
  Val v
| Exn v
| Div (io_event llist)
```

CF (before)

```
cf (e1; e2) P Q =
  local(∃R. cf e1 P R ∧
           ∀res. is_value(res) ⇒ cf e2 (R res) Q  ∧
                ¬is_value(res) ⇒  R(res) ⇒ Q(res)
       )
```

CF (after)

```
cf (e1; e2) P Q =
  local(∃R. cf e1 P R ∧
           ∀res. is_value(res) ⇒ cf e2 (R res) Q  ∧
                ¬is_value(res) ⇒  R(res) ⇒ Q(res)
       )
```

CF (before)

```
cf (e1; e2) P Q =
  local(∃R. cf e1 P R ∧
            ∀res. is_value(res) ⇒ cf e2 (R res) Q  ∧
                  ¬is_value(res) ⇒  R(res) ⇒ Q(res)
       )
```

No difference whatsoever!*

CF (after)

```
cf (e1; e2) P Q =
  local(∃R. cf e1 P R ∧
            ∀res. is_value(res) ⇒ cf e2 (R res) Q  ∧
                  ¬is_value(res) ⇒  R(res) ⇒ Q(res)
       )
```

CF

Theorem (soundness)

If

`cf` e P Q

then

{P} e {Q}

Reconciling repeat

😱

```
fun yes() =
  repeat (fn _ => print(“y\n”)) ()
```

Reconciling repeat

😱
```
fun yes() =
  repeat (fn _ => print(“y\n”)) ()
```

😄
```
fun yes() =
  (print(“y\n”); yes())
```

Reconciling repeat

Theorem (folk)

For every tail-recursive function `f`
there exists a non-recursive function `g`,
such that whenever `f(x)` diverges,

```
repeat g x
```

diverges in an observationally equivalent way.

Reconciling repeat

Theorem (folk?)

For every tail-recursive function `f`
there exists a non-recursive function `g`,
such that whenever `f(x)` diverges,

```
repeat g x
```

diverges in an observationally equivalent way.

A verified program transformation automatically produces such a g before the WHILE rule is applied

Limitations of repeat transformation

😱 Only supports tail recursion

Not a problem in practice:

```
fun f(x) = f(x) + 1
```

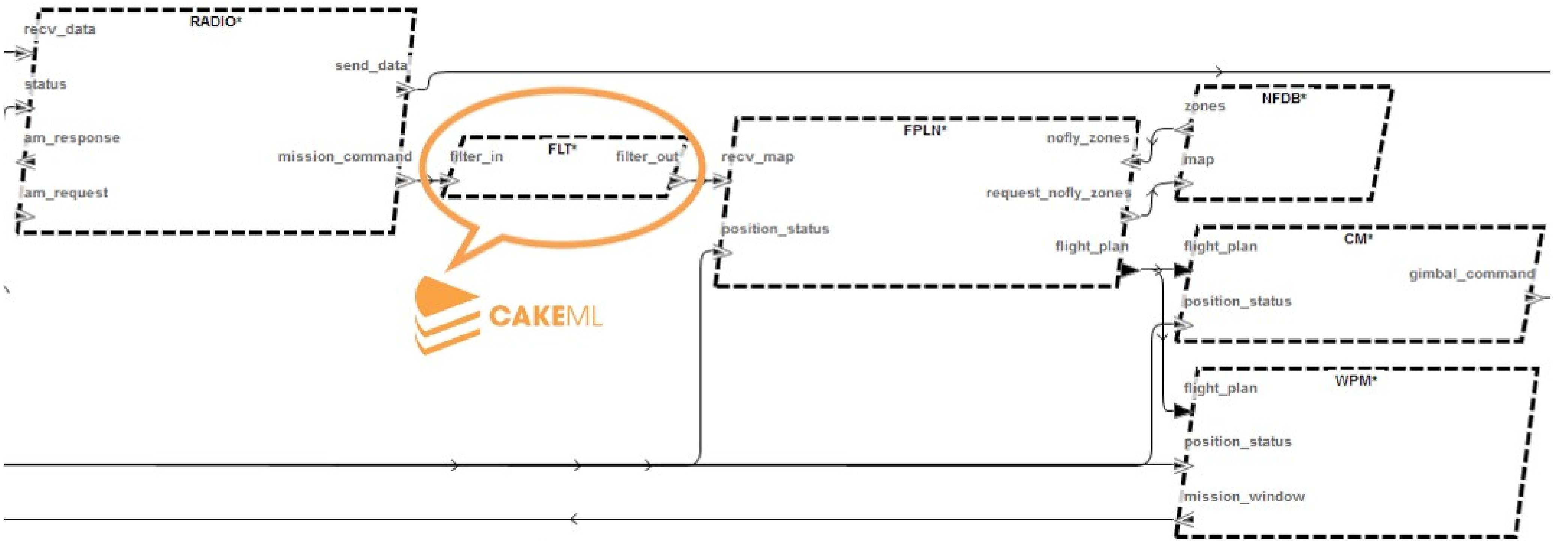
diverges in theory, but overflows the stack in practice.

😱 No support for mutual recursion (yet)

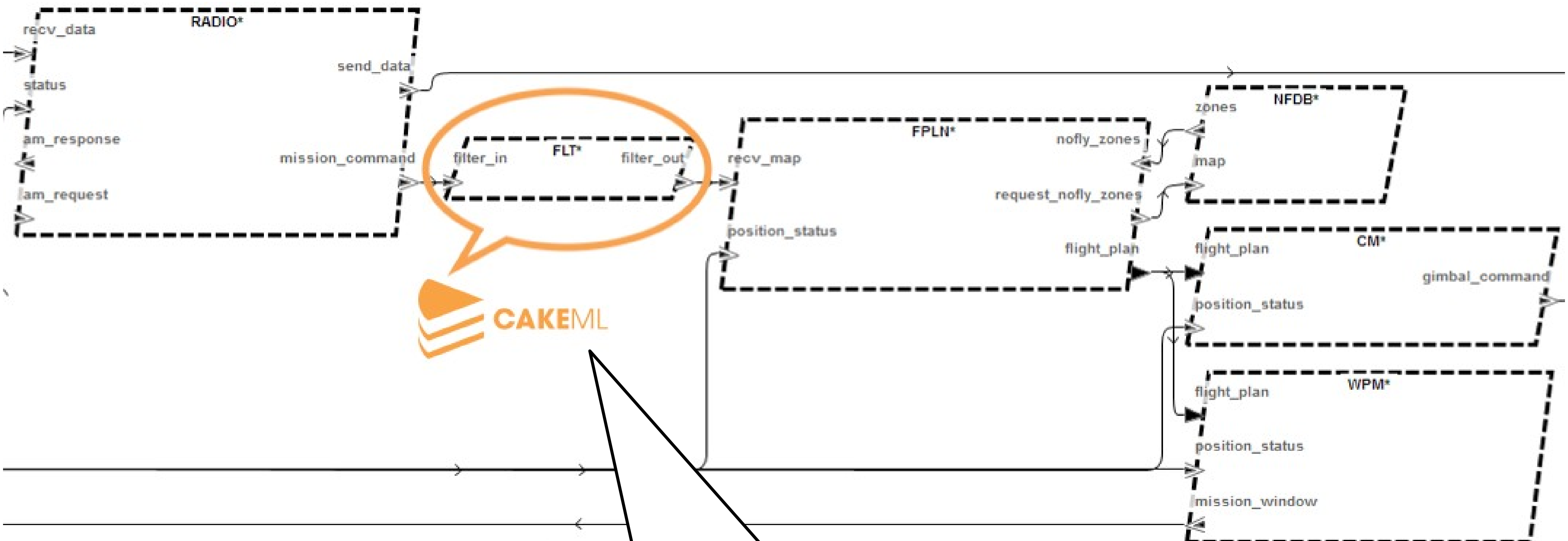
😱 No support for weird recursion through the store

```
val f = ref (fn () => ());
val g = (fn () => !f ());
val _ = (f := g; g());
```

Case study: verified filters



Case study: verified filters



The filter *cannot* terminate
(If it does, the UAV can't be contacted)

Conclusion

- 😄 Conservative extension of CF to support liveness proofs for non-terminating programs.
- 😄 Formalised in HOL4, integrated into CakeML ecosystem

  https://code.cakeml.org

- 😄 Thanks for listening!