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
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}
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
Characteristic Formulae for Liveness Properties of Non-terminating CakeML Programs

Oversimplifies the paper!
Two kinds of post-conditions

\{P\} e \{\text{Trm } Q\}

\{P\} e \{\text{Div } S\}
Two kinds of post-conditions

\{P\} \ e \ \{\text{Trm} \ Q\}

Bog-standard total correctness
Hoare triple!

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

\{P\} \ e \ \{\text{Div} \ S\}
Two kinds of post-conditions

\{P\} \ e \ \{\text{Trm} \ Q\} \\

Bog-standard total correctness Hoare triple!

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

\{P\} \ e \ \{\text{Div} \ S\} \\

“If P holds initially, then e runs forever, producing a trace satisfying S”
Two kinds of post-conditions

\[
\{P\} \ e \ \{\text{Trm} \ Q\}
\]

Bog-standard total correctness Hoare triple!
“If P holds initially, then e terminates in a state satisfying Q”

\[
\{P\} \ e \ \{\text{Div} \ S\}
\]

“If P holds initially, then e runs forever, producing a trace satisfying S”
Two kinds of post-conditions

Bog-standard total correctness Hoare triple!

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

“If P holds initially, then e runs forever, producing a trace satisfying Q”
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 $\lambda(s,h). h=[]$

\{h=[]\}

print(s)

\{Trm(h=[print s])\}
The WHILE rule

\[ P \Rightarrow I_0 \land h = h_0 \]
\[ \forall i. \{ I_i \land h = [] \} \text{ e } \{ \text{Trm}(I_{i+1} \land h = h_{i+1}) \} \]
\[ S(h_0 \cdot h_1 \cdot h_2 \ldots \ldots) \]

\[ \{ P \} \text{ while(true) do e } \{ \text{Div S} \} \]
The WHILE rule

\[
\begin{align*}
P & \Rightarrow I_0 \land h=h_0 \\
\forall i. \{I_i \land h=[]\} & \text{ e } \{\text{Trm}(I_{i+1} \land h=h_{i+1})\} \\
S(h_0 \cdot h_1 \cdot h_2 \ldots ) & \\
\{P\} & \text{ while(true) do e } \{\text{Div } S\}
\end{align*}
\]
The WHILE rule

\[ P \Rightarrow I_0 \land h=h_0 \]
\[ \forall i. \{ I_i \land h=\emptyset \} \vDash \{ \text{Trm}(I_{i+1} \land h=h_{i+1}) \} \]
\[ S(h_0 \cdot h_1 \cdot h_2 \ldots \ldots) \]

\[ \{ P \} \text{ while(true) do e } \{ \text{Div S} \} \]
The WHILE rule

\[ P \Rightarrow I_0 \land h=h_0 \]

\[ \forall i. \{ I_i \land h=[] \} \Leftrightarrow \{ T_{\text{rm}}(I_{i+1} \land h=h_{i+1}) \} \]

\[ S(h_0 \cdot h_1 \cdot h_2 \ldots) \]

\[ \{ P \} \text{ while(true) do e \{ Div S \} } \]
The WHILE rule

\[ P \Rightarrow I_0 \land h=h_0 \]

\[ \forall i. \{I_i \land h=[]\} \Rightarrow \{\text{Trm}(I_{i+1} \land h=h_{i+1})\} \]

\[ S(h_0 \cdot h_1 \cdot h_2 \cdots) \]

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

Actually more like

\[ S(\text{lflatten}((\lambda n. h_n))) \]
yes again

```plaintext
{h = []} 
while(true) do {
    print("y\n"); 
}
{Div(λt. t = [print("y\n")]^ω)}
```
yes again

\{h = []\}

while(true) do {
    print("y\n");
}

\{Div(\lambda t. t = [print("y\n")])^\omega}\}

Proof: apply the WHILE rule with

\[I_0 = \lambda x. \text{true}\]
\[h_0 = []\]
\[h_n = [\text{print}("y\n")]\] 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
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

\[ P \Rightarrow I_0 \land h = h_0 \]

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

\[ S(h_0 \cdot h_1 \cdot h_2 \ldots .) \]

\[ \{ P \} \text{ while(true) do e } \{ \text{Div } S \} \]
The WHILE rule reloaded

\[ P \Rightarrow I_0 \land h = h_0 \]
\[ \forall i. \{I_i \land h = []\} \land \{\text{Trm}(I_{i+1} \land h = h_{i+1})\} \]
\[ S(h_0 \cdot h_1 \cdot h_2 \ldots) \]

\{P\} \text{ while(true) do e \{Div S\}
The WHILE rule reloaded

\[ \text{fun repeat } f \ x = \text{repeat } f \ (f \ x) \]

**Expressions have return values**

\[ \text{P} \Rightarrow \text{I}_0 \ast h=h_0 \]
\[ \forall i. \ \{I_i \ast h=[]\} \ f(x_i) \ \{\text{Trm}(x_{i+1})(I_{i+1} \ast h=h_{i+1})\} \]
\[ S(h_0 \cdot h_1 \cdot h_2 \ldots) \]

\[ \{\text{P}\} \ \text{repeat } f \ x_0 \ \{\text{Div } S\} \]
The WHILE rule reloaded

\[ P \Rightarrow l_0 \ast h=h_0 \]
\[ \forall i. \{ l_i \ast h=\_\} \ f(x_i) \ \{ \text{Trm}(x_{i+1})(l_{i+1} \ast h=h_{i+1}) \} \]
\[ S(h_0 \cdot h_1 \cdot h_2 \ldots \ldots) \]

\[ \{ P \} \ \text{repeat} \ f \ x_0 \ \{ \text{Div} \ S \} \]

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:

\[
\text{cf} : \text{exp} \Rightarrow (\text{state} \Rightarrow \text{bool}) \Rightarrow (\text{result} \Rightarrow \text{state} \Rightarrow \text{bool}) \Rightarrow \text{bool}
\]
CF (Characteristic formulae)

Verification condition generator for impure functional programs [Charguéraud 2010]

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

Workhorse function:

\[
\text{cf} : \text{exp} \rightarrow (\text{state} \rightarrow \text{bool}) \rightarrow \\
(\text{result} \rightarrow \text{state} \rightarrow \text{bool}) \rightarrow \text{bool}
\]

Intuition: to prove a Hoare triple

\[
\{P\} e \{Q\}
\]

it suffices to show

\[
\text{cf} e P Q
\]
CF (before)

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

CF (after)

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

\[
\text{cf} \ (e_1; \ e_2) \ P \ Q = \\
\text{local}(\exists R. \ \text{cf} \ e_1 \ P \ R \land \\
\quad \forall \text{res}. \ \text{is\_value}(\text{res}) \Rightarrow \text{cf} \ e_2 \ (R \ \text{res}) \ Q \land \\
\quad \neg \text{is\_value}(\text{res}) \Rightarrow R(\text{res}) \Rightarrow Q(\text{res})
\]
No difference whatsoever!*
Theorem (soundness)

If

\[ \text{cf } e \ P \ Q \]

then

\[ \{P\} \ e \ \{Q\} \]
Reconciling repeat

```plaintext
fun yes() = 
  repeat (fn _ => print("y\n")) ()
```
Reconciling repeat

😊 fun yes() =
  repeat (fn _ => print("y\n")(())

😄 fun yes() =
  (print("y\n"); yes())
Theorem (folk)

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

\[
\text{repeat } g \ x
\]

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
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!