Verified Characteristic Formulae for CakeML

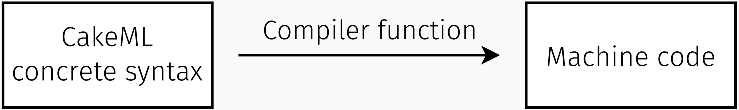
Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, Michael Norrish

April 18, 2017

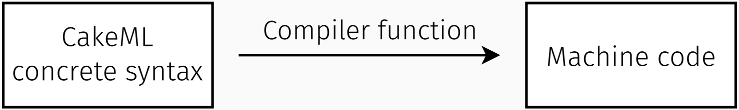
CAKEML

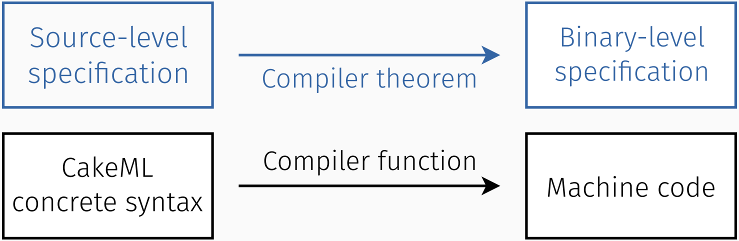
A Verified Implementation of ML

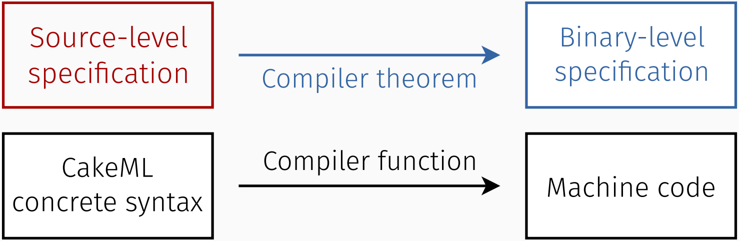
- Has: references, modules, datatypes, exceptions, a FFI, ...
- Doesn't have: functors, module nesting, let-polymorphism


CakeML
concrete syntax
Compiler function
Machine code

Compiler theorem




Source-level specification
Compiler theorem
Binary-level specification
CakeML concrete syntax
Compiler function
Machine code

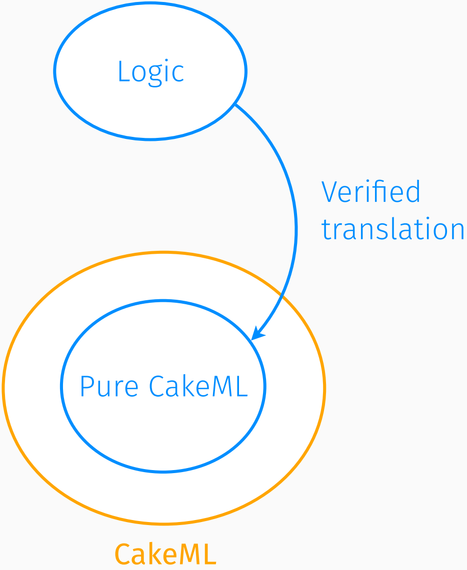


How do we get verified CakeML programs?

Verified translation: Myreen & Owens, ICFP'12

- Define and verify the program as a function in the logic.
- The translator automatically produces CakeML code...
- ...and a certificate theorem.

- Used to verify most of the compiler
- Drawback: it can only produce pure CakeML programs


Logic
Verified
translation
Pure CakeML
CakeML

Characteristic Formulae for CakeML: this work

A program logic for CakeML, based on the Characteristic Formulae approach.

- Based on the work of Arthur Charguéraud
- Handles all CakeML features, including I/O and exceptions
- Formally proved sound
- Interoperates with the proof-producing translator

Main contributions

New verification framework for CakeML.
We developed a significant addition to the CakeML ecosystem of verification tools.

Validating the CF approach.
Arthur Charguéraud's work on CF was only partly proved in Coq. We showed that CF can be proved completely sound in a theorem prover, and one can extend the framework to do more, exceptions and I/O.

Outline

Background on CF

Soundness theorem: connecting CF to CakeML semantics

Sound extensions of CF

- Support for I/O through the CakeML FFI
- Support for exceptions

Interoperating with the proof-producing translator

Background on CF

Program verification using Characteristic Formulae

CFML: program verification framework based on CF (Charguéraud, ICFP'11)

- for OCaml programs
- using the Coq proof assistant

This work: "CFML for CakeML" (and the HOL4 proof assistant)

How does a CF framework works?

The main workhorse: the `cf` function.

- Source-level expression e $\rightarrow$ its characteristic formula (`cf` e)

(`cf` e):

- logical formula that doesn't mention the syntax of e
- abstracted away from the details of the semantics
- akin to a total correctness Hoare triple

CFML: `cf` defined outside the logic; our framework: `cf` defined and proved in the logic.

How does a CF framework works? (2)

$(\texttt{cf}\ e)\ \mathit{env}\ H\ Q$:

- "*e* can have *H* as pre-condition and *Q* as post-condition in environment *env*"
- H, Q: heap predicates (separation logic assertions)
- $H : \texttt{heap} \rightarrow \texttt{bool}$
- $Q : \texttt{v} \rightarrow \texttt{heap} \rightarrow \texttt{bool}$

Examples

$$
\begin{array}{l}
\texttt{cf}\ (\texttt{Var}\ \mathit{name})\ \mathit{env} = \texttt{local}\ (\lambda H\ Q.\\
\quad \exists\, v.\ \texttt{lookup\_var\_id}\ \mathit{name}\ \mathit{env} = \texttt{Some}\ v \wedge\\
\qquad\quad H \triangleright Q\ v)
\end{array}
$$

$$
\begin{array}{l}
\texttt{cf}\ (\texttt{Let}\ (\texttt{Some}\ x)\ e_1\ e_2)\ \mathit{env} = \texttt{local}\ (\lambda H\ Q.\\
\quad \exists\, Q'.\\
\qquad \texttt{cf}\ e_1\ H\ Q' \wedge\\
\qquad \forall\, \mathit{xv}.\ \texttt{cf}\ e_2\ ((x, \mathit{xv}) :: \mathit{env})\ (Q'\ \mathit{xv})\ Q)
\end{array}
$$

$$
\begin{array}{l}
\texttt{cf}\ (\texttt{If}\ \mathit{cond}\ e_1\ e_2)\ \mathit{env}\ = \texttt{local}\ (\lambda\, H\ Q.\\
\quad \exists\, \mathit{condv}\ b.\\
\qquad \texttt{exp\_is\_val}\ \mathit{env}\ \mathit{cond}\ = \ \texttt{Some}\ \mathit{condv}\ \wedge\ \texttt{BOOL}\ b\ \mathit{condv}\ \wedge\\
\qquad ((b \iff \texttt{T}) \Rightarrow \texttt{cf}\ e_1\ \mathit{env}\ H\ Q)\ \wedge\\
\qquad ((b \iff \texttt{F}) \Rightarrow \texttt{cf}\ e_2\ \mathit{env}\ H\ Q))
\end{array}
$$

Program specifications

Specifications:

- Stated using app: Hoare-triple for functional applications
- Written $\{|H|\}\ f \cdot \mathit{args}\ \{|Q|\}$
- Related to cf via a consequence of the soundness theorem:

$$
\begin{array}{l}
\vdash\ \mathit{ns} \neq [] \Rightarrow \\
\quad \texttt{length}\ \mathit{xvs} = \texttt{length}\ \mathit{ns} \Rightarrow \\
\quad\ \ \texttt{cf}\ \mathit{body}\ (\texttt{extend\_env}\ \mathit{ns}\ \mathit{xvs}\ \mathit{env})\ H\ Q \Rightarrow \\
\quad\quad \{|H|\}\ \texttt{naryClosure}\ \mathit{env}\ \mathit{ns}\ \mathit{body} \cdot \mathit{xvs}\ \{|Q|\}
\end{array}
$$

Example: a specification for cat

```
fun do_onefile fname =
  let
    val fd = CharIO.openIn fname
    fun recurse () =
      case CharIO.fgetc fd of
          NONE ⇒ ()
        | SOME c ⇒
          CharIO.write c;
          recurse ()
  in recurse ();
     CharIO.close fd
  end

fun cat fnames =
  case fnames of
    [] ⇒ ()
  | f::fs ⇒ do_onefile f; cat fs
```

$$
\begin{array}{l}
\vdash \text{LIST FILENAME } \mathit{fns}\ \mathit{fnsv}\ \wedge \\
\ \text{every } (\lambda\, \mathit{fnm}.\ \text{inFS\_fname } \mathit{fnm}\ \mathit{fs})\ \mathit{fns}\ \wedge \\
\ \text{numOpenFDs } \mathit{fs} < 255 \Rightarrow \\
\quad \{\!|\text{CATFS } \mathit{fs} * \text{STDOUT } \mathit{out}|\!\} \\
\quad\ \text{cat\_v} \cdot [\mathit{fnsv}] \\
\quad \{\!|\lambda\, u. \\
\qquad \langle \text{UNIT } ()\ u \rangle * \text{CATFS } \mathit{fs} * \\
\qquad \text{STDOUT } (\mathit{out}\ @\ \text{catfiles\_string } \mathit{fs}\ \mathit{fns})|\!\}
\end{array}
$$

Soundness theorem: connecting CF to CakeML semantics

Soundness of a CF framework

"Proving properties on a characteristic formula gives equivalent properties about the program itself"

CFML:

- no formal semantics of OCaml
- assumes idealized semantics
- some parts are axiomatized

CF for CakeML:

- Re-implement CF generation as in CFML
- Realize CFML axioms wrt. CakeML semantics
- Prove an end-to-end correctness theorem

Connecting CF to CakeML semantics

Heap predicates and semantic store

"(`cf` e) env H Q": H and Q are assertions about the memory heap

Examples:

- $(r \rightsquigarrow v)$: heap containing one reference cell r pointing to value v
- $(r_1 \rightsquigarrow v_1 * r_2 \rightsquigarrow v_2)$: heap containing two *distinct* reference cells ("$*$": separating conjunction of separation logic)

Connecting CF to CakeML semantics

Heap predicates and semantic store

CakeML semantics describe the memory heap in the `state` record:

```
state =
  <| clock : num
   ; refs : v store_v list
   ; ffi : θ ffi_state
   ; defined_types : tid_or_exn set
   ; defined_mods : (modN list) set
   |>
```

```
'a store_v =
 (* A ref cell *)
   Refv of 'a
 (* A byte array *)
 | W8array of word8 list
 (* An array of values *)
 | Varray of 'a list
```

Connecting CF to CakeML semantics

Heap predicates and semantic store

Define heaps holding CakeML values:

$$\texttt{heap} = (\texttt{num} \times \texttt{v}\ \texttt{store\_v})\ \mathsf{set}$$

$$\begin{aligned} r \rightsquigarrow v \ &= \ (\lambda\, h.\ \exists\, loc.\ r = \texttt{Loc}\ loc \ \wedge\ h = \{\ (loc,\ \texttt{Refv}\ v)\ \}\,) \\ p\ *\ q \ &= \ (\lambda\, h.\ \exists\, u\ v.\ \texttt{split}\ h\,(u, v)\ \wedge\ p\ u\ \wedge\ q\ v) \end{aligned}$$

Define $\texttt{state\_to\_set} : \texttt{state} \rightarrow \texttt{heap}$.

For a state st with $st.refs = [\texttt{Refv}\ v_1;\ \texttt{Refv}\ v_2]$:

- $\texttt{state\_to\_set}\ st = \{(0, v_1);\ (1, v_2)\}$
- $(\texttt{Loc}\ 0 \rightsquigarrow v_1 * \texttt{Loc}\ 1 \rightsquigarrow v_2)\ (\texttt{state\_to\_set}\ st)$

Connecting CF to CakeML semantics

Logical values and deep-embedded values

CakeML values:

```
v =
    Litv lit
  | Conv ((conN × tid_or_exn) option) (v list)
  | Closure (v sem_env) string exp
  | Recclosure (v sem_env) ((string × string × exp) list) string
  | Loc num
  | Vectorv (v list)
```

We reuse the *refinement invariants* used by the translator:

$$\texttt{INT}\ i\ =\ (\lambda\, v.\ v\ =\ \texttt{Litv}\ (\texttt{IntLit}\ i))$$
$$\texttt{BOOL}\ \texttt{T}\ =\ (\lambda\, v.\ v\ =\ \texttt{Conv}\ (\texttt{Some}\ (\text{“true”}, \texttt{TypeId}\ (\texttt{Short}\ \text{“bool”})))\ [])$$

$$\vdash\ \texttt{INT}\ x_0\ v_0\ \wedge\ \texttt{INT}\ x_1\ v_1\ \Rightarrow$$
$$\{\!|\texttt{emp}|\!\}\ \texttt{plus\_v} \cdot [v_0;\ v_1]\ \{\!|\lambda\, v.\ \langle \texttt{INT}\ (x_0\ +\ x_1)\ v\rangle|\!\}$$

Realizing CFML axioms: `app`

Give an implementation for app, written "$\{|H|\}\ f \cdot \mathit{args}\ \{|Q|\}$", which is axiomatized in CFML.

Extract from CakeML big-step semantics:

```
evaluate st env [Lit l] = (st, Rval [Litv l])
evaluate st env [Var n] =
 case lookup_var_id n env of
  None ⇒ (st, Rerr (Rabort Rtype_error))
 | Some v ⇒ (st, Rval [v])
evaluate st env [Fun x e] = (st, Rval [Closure env x e])
evaluate st env [App Opapp [f; v]] =
 case evaluate st env [v; f] of
  (st′, Rval [v; f]) ⇒
     case do_opapp [f; v] of
      None ⇒ (st′, Rerr (Rabort Rtype_error))
     | Some (env′, e) ⇒
         if st′.clock = 0 then
          (st′, Rerr (Rabort Rtimeout_error))
         else evaluate (dec_clock st′) env′ [e]
 | res ⇒ res

do_opapp vs =
 case vs of
  [Closure env n e; v] ⇒ Some ((n, v) :: env, e)
 | [Recclosure env funs n; v] ⇒ . . .
 | _ ⇒ None
```

```
evaluate :
  state →
  v sem_env →
  exp list →
  state × (v list, v) result
```

Realizing CFML axioms: `app`

Semantics of Hoare-triples for expressions

Hoare-triple for an expression e in environment env:
"$env \vdash \{\!| H |\!\}\ e\ \{\!| Q |\!\}$"

$$
\begin{array}{l}
env \vdash \{\!| H |\!\}\ e\ \{\!| Q |\!\} \iff \\
\quad \forall\, st\ h_i\ h_k. \\
\quad\quad \texttt{split}\ (\texttt{state\_to\_set}\ st)\ (h_i, h_k) \Rightarrow \\
\quad\quad\quad H\ h_i \Rightarrow \\
\quad\quad\quad\quad \exists\, v\ st'\ h_f\ h_g\ ck. \\
\quad\quad\quad\quad\quad \texttt{evaluate}\ (st \text{ with clock } := ck)\ env\ [e] = (st', \texttt{Rval}\ [v]) \wedge \\
\quad\quad\quad\quad\quad \texttt{split3}\ (\texttt{state\_to\_set}\ st')\ (h_f, h_k, h_g) \wedge Q\ v\ h_f
\end{array}
$$

Integrates the frame rule with GC: h_k is the frame, h_g is the garbage

Realizing CFML axioms: `app`

Semantics of Hoare-triples for unary application

Hoare-triple for the application of a closure to a single argument: "$\{\!|H|\!\}\ f \cdot x\ \{\!|Q|\!\}$"

$$\{\!|H|\!\}\ f \cdot x\ \{\!|Q|\!\} \iff$$
$$\text{case } \mathtt{do\_opapp}\ [f;\ x] \text{ of}$$
$$\quad \mathtt{None} \Rightarrow \forall\, st\ h_1\ h_2.\ \mathtt{split}\ (\mathtt{state\_to\_set}\ p\ st)\ (h_1, h_2) \Rightarrow \neg H\ h_1$$
$$\mid \mathtt{Some}\ (env, exp) \Rightarrow env \vdash \{\!|H|\!\}\ exp\ \{\!|Q|\!\}$$

Realizing CFML axioms: `app`

Semantics of Hoare-triples for n-ary application

Hoare-triple for the application of a closure to multiple arguments:
"$\{\!|H|\!\}\ f \cdot args\ \{\!|Q|\!\}$"

$$\begin{aligned}
&\{\!|H|\!\}\ f \cdot []\ \{\!|Q|\!\} \iff \mathtt{F} \\
&\{\!|H|\!\}\ f \cdot [x]\ \{\!|Q|\!\} \iff \{\!|H|\!\}\ f \cdot x\ \{\!|Q|\!\} \\
&\{\!|H|\!\}\ f \cdot x :: x' :: xs\ \{\!|Q|\!\} \iff \\
&\quad \{\!|H|\!\}\ f \cdot x\ \{\!|\lambda\, g.\ \exists\ H'.\ H' \ast \langle \{\!|H'|\!\}\ g \cdot x' :: xs\ \{\!|Q|\!\} \rangle |\!\}
\end{aligned}$$

Specifications are modular: `app` integrates the frame rule

Proving CF soundness

Soundness for an arbitrary formula F:

$$\texttt{sound}\ e\ F \iff \forall\, env\ H\ Q.\ F\ env\ H\ Q \Rightarrow env \vdash \{\!|H|\!\}\ e\ \{\!|Q|\!\}$$

Theorem (CF are sound wrt. CakeML semantics):

$$\vdash\ \texttt{sound}\ e\ (\texttt{cf}\ e)$$

Proof: by induction on the size of e.

Sound extensions of CF

Sound extensions of CF

Support for I/O through the CakeML FFI

Performing I/O in CakeML

CakeML programs do I/O using a byte-array-based foreign-function interface (FFI).

- "`App` (`FFI` *name*) [*array*]": a CakeML expression
- Calls the external function "*name*" (typically implemented in C) with "*array*" as a parameter
- Reads back the result in "*array*"

For example: read a character from `stdin`, open a file, ...

CakeML I/O semantics

- The state of the "external world" is modeled by the semantics FFI state (what has been printed to `stdout`, which files are open, ...)
- Executing an FFI operation updates the state of the FFI
- FFI state changes are modeled by an oracle function

```
state =
  <| clock : num
   ; refs : v store_v list
   ; ffi : θ ffi_state
   ; defined_types :
       tid_or_exn set
   ; defined_mods :
       (modN list) set
   |>
```

```
θ ffi_state =
<| oracle :
     string → θ → byte list →
     θ oracle_result
 ; ffi_state : θ
 ; final_event : final_event option
 ; io_events : io_event list
 |>
```

Reasoning about I/O in CF

- Modify (`state_to_set` : `state` → `heap`) to expose the FFI to pre- and post-conditions
- Modular proofs: need to be able to split the FFI state using "∗" (proofs about `stdout` should be independent from proofs about the file-system...)

```
θ ffi_state =
<| oracle :
     string → θ → byte list →
     θ oracle_result
 ; ffi_state : θ
 ; final_event : final_event option
 ; io_events : io_event list
 |>
```

Problem: we know nothing about the type variable θ!

Splitting the FFI state

Solution: parametrize `state_to_set` with information on how to split the FFI state into "parts".

- A *part* represents an independent bit of the external world
- Several external functions can update the same part
- The FFI state θ can be split into separated parts
- "`stdout`" would be a part, "`stdin`" an other, the filesystem a third one...

Splitting the FFI state (2)

We parametrize `state_to_set` with:

- A projection function *proj* : $\theta \to$ (`string` $\mapsto$ `ffi`)
- A list of FFI *parts* : (`string list` $\times$ `ffi_next`) `list`

`ffi`: low-level generic model for the state of a FFI part

`ffi_next`: "next-state function", a part of the oracle

```
ffi =
  Str string
| Num num
| Cons ffi ffi
| List (ffi list)
| Stream (num stream)

ffi_next =
  string → byte list → ffi →
  (byte list × ffi) option
```

Splitting the FFI state (3)

Finally, we define a generic IO heap assertion:

$$\texttt{IO} : \texttt{ffi} \to \texttt{ffi\_next} \to \texttt{string list} \to \texttt{heap} \to \texttt{bool}$$
$$\texttt{IO}\ \mathit{st}\ \mathit{u}\ \mathit{ns}\ =\ (\lambda\, s.\ \exists\, \mathit{ts}.\ s\ =\ \{\ \texttt{FFI\_part}\ \mathit{st}\ \mathit{u}\ \mathit{ns}\ \mathit{ts}\ \})$$

Pre- and post-conditions can now make assertions about I/O. Users typically define more specialized assertions on top of IO.

Not described in this presentation:

- Characteristic formula for "`App` (`FFI` *name*) [*array*]"
- How the soundness proof was updated
- How CF is used in the bootstrapped CakeML compiler to verify the I/O part

Sound extensions of CF

Support for exceptions

Exception-aware post-conditions

Without support for exceptions:

- An expression must reduce to a value
- Post-conditions have type $\texttt{v} \to \texttt{heap} \to \texttt{bool}$

We now allow expressions to raise an exception:

- Define datatype $\texttt{res} = \texttt{Val v} \mid \texttt{Exn v}$
- Post-conditions have type $\texttt{res} \to \texttt{heap} \to \texttt{bool}$
- Define wrappers for common cases:

$$
\begin{aligned}
(\texttt{POSTv})\ Q_v \ &= \ (\lambda\, r.\ \text{case } r \text{ of } \texttt{Val}\ v \ \Rightarrow\ Q_v\ v \mid \texttt{Exn}\ e \ \Rightarrow\ \langle \texttt{F} \rangle) \\
(\texttt{POSTe})\ Q_e \ &= \ (\lambda\, r.\ \text{case } r \text{ of } \texttt{Val}\ v \ \Rightarrow\ \langle \texttt{F} \rangle \mid \texttt{Exn}\ e \ \Rightarrow\ Q_e\ e) \\
\texttt{POST}\ Q_v\ Q_e \ &= \ (\lambda\, r.\ \text{case } r \text{ of } \texttt{Val}\ v \ \Rightarrow\ Q_v\ v \mid \texttt{Exn}\ e \ \Rightarrow\ Q_e\ e)
\end{aligned}
$$

Example: a more general specification for `cat1`

We can remove the precondition that the input file must exist:

$$
\begin{array}{l}
\vdash \texttt{FILENAME}\ \mathit{fnm}\ \mathit{fnv}\ \wedge\ \texttt{numOpenFDs}\ \mathit{fs}\ <\ 255\ \Rightarrow \\
\quad \{\!|\texttt{CATFS}\ \mathit{fs}\ *\ \texttt{STDOUT}\ \mathit{out}|\!\} \\
\quad\ \ \texttt{cat1\_v}\cdot[\mathit{fnv}] \\
\quad \{\!|\texttt{POST} \\
\qquad (\lambda\, u. \\
\qquad\quad \exists\ \mathit{content}. \\
\qquad\qquad \langle\texttt{UNIT}\ ()\ u\rangle\ *\ \langle\texttt{alist\_lookup}\ \mathit{fs}.\mathrm{files}\ \mathit{fnm}\ =\ \texttt{Some}\ \mathit{content}\rangle\ * \\
\qquad\qquad \texttt{CATFS}\ \mathit{fs}\ *\ \texttt{STDOUT}\ (\mathit{out}\ @\ \mathit{content})) \\
\qquad (\lambda\, e. \\
\qquad\quad \langle\texttt{BadFileName\_exn}\ e\rangle\ *\ \langle\neg\texttt{inFS\_fname}\ \mathit{fnm}\ \mathit{fs}\rangle\ *\ \texttt{CATFS}\ \mathit{fs}\ * \\
\qquad\quad \texttt{STDOUT}\ \mathit{out})|\!\}
\end{array}
$$

Exception-aware Hoare-triples

Hoare-triple validity "$env \vdash \{|H|\}\ e\ \{|Q|\}$" becomes:

$$
\begin{array}{l}
env \vdash \{|H|\}\ e\ \{|Q|\} \iff \\
\quad \forall\, st\ h_i\ h_k. \\
\quad\quad \texttt{split}\ (\texttt{state\_to\_set}\ p\ st)\ (h_i, h_k) \Rightarrow \\
\quad\quad\quad H\ h_i \Rightarrow \\
\quad\quad\quad\quad \exists\, r\ st'\ h_f\ h_g\ ck. \\
\quad\quad\quad\quad\quad \texttt{split3}\ (\texttt{state\_to\_set}\ p\ st')\ (h_f, h_k, h_g) \wedge Q\ r\ h_f \wedge \\
\quad\quad\quad\quad\quad \textsf{case}\ r\ \textsf{of} \\
\quad\quad\quad\quad\quad\quad \texttt{Val}\ v \Rightarrow \texttt{evaluate}\ (st\ \textsf{with clock} := ck)\ env\ [e] = (st', \texttt{Rval}\ [v]) \\
\quad\quad\quad\quad\quad \mid \texttt{Exn}\ v \Rightarrow \texttt{evaluate}\ (st\ \textsf{with clock} := ck)\ env\ [e] = (st', \texttt{Rerr}\ (\texttt{Rraise}\ v))
\end{array}
$$

Note: we still rule out actual failures, where `evaluate` returns "`Rerr (Rabort abort)`".

Updating cf

Add side-conditions to characteristic formulae, to deal with exceptions:

$$\begin{array}{l}\texttt{cf}\ p\ (\texttt{Var}\ \mathit{name})\ \mathit{env}\ =\ \texttt{local}\ (\lambda\, H\ Q.\\ \quad (\exists\, v.\ \texttt{lookup\_var\_id}\ \mathit{name}\ \mathit{env}\ =\ \texttt{Some}\ v\ \wedge\ H\ \rhd\ Q\ (\texttt{Val}\ v))\ \wedge\\ \quad Q\ \blacktriangleright_{\mathrm{e}}\ \mathbf{F})\end{array}$$

$$\begin{array}{l}\texttt{cf}\ p\ (\texttt{Let}\ (\texttt{Some}\ x)\ e_1\ e_2)\ \mathit{env}\ =\ \texttt{local}\ (\lambda\, H\ Q.\\ \quad \exists\, Q'.\\ \qquad \texttt{cf}\ p\ e_1\ \mathit{env}\ H\ Q'\ \wedge\ Q'\ \blacktriangleright_{\mathrm{e}}\ Q\ \wedge\\ \qquad \forall\, xv.\ \texttt{cf}\ p\ e_2\ ((x, xv) :: \mathit{env})\ (Q'\ (\texttt{Val}\ xv))\ Q)\end{array}$$

$$Q_1\ \blacktriangleright_{\mathrm{e}}\ Q_2 \iff \forall e.\ Q_1\ (\texttt{Exn}\ e)\ \rhd\ Q_2\ (\texttt{Exn}\ e)$$

CFs for `raise` and `handle`

Define `cf` for `Raise` and `Handle`: similar to the `Var` and `Let` cases

$$\begin{array}{l}\texttt{cf}\ p\ (\texttt{Raise}\ e)\ env\ = \texttt{local}\ (\lambda\, H\ Q.\\ \quad \exists\, v.\ \texttt{exp\_is\_val}\ env\ e\ =\ \texttt{Some}\ v\ \wedge\ H\ \rhd\ Q\ (\texttt{Exn}\ v)\ \wedge\ Q\ \blacktriangleright_{\mathrm{v}}\ \mathbf{F})\end{array}$$

$$\begin{array}{l}\texttt{cf}\ p\ (\texttt{Handle}\ e\ rows)\ env\ = \texttt{local}\ (\lambda\, H\ Q.\\ \quad \exists\, Q'.\\ \qquad \texttt{cf}\ p\ e\ env\ H\ Q'\ \wedge\ Q'\ \blacktriangleright_{\mathrm{v}}\ Q\ \wedge\\ \qquad \forall\, ev.\\ \qquad\quad \texttt{cf\_cases}\ ev\ ev\ (\texttt{map}\ (\texttt{I}\ \#\#\ \texttt{cf}\ p)\ rows)\ env\ (Q'\ (\texttt{Exn}\ ev))\ Q)\end{array}$$

$$Q_1\ \blacktriangleright_{\mathrm{v}}\ Q_2 \iff \forall e.\ Q_1\ (\texttt{Val}\ e)\ \rhd\ Q_2\ (\texttt{Val}\ e)$$

- Only basic automation is required (rewriting `POSTv` Q (`Exn` e) $\Leftrightarrow$ `F`, `POSTv` Q (`Val` v) $\Leftrightarrow$ $Q\ v$, ...)
- No additional proof effort for verifying programs that do not involve exceptions

Interoperating with the proof-producing translator

Verified translation from HOL to CakeML: ICFP'12

- Define and verify the program in HOL4:

$$(\text{length } [] = 0) \land$$
$$(\text{length } (h :: t) = 1 + \text{length } t)$$

$$\vdash \forall x\, y.\, \text{length } (x +\!\!+\, y) = \text{length } x + \text{length } y$$

- The translator automatically produces CakeML code ...

```
fun length_ml x =
  case x of
    | [] ⇒ 0
    | (h::t) ⇒ 1 + length t
```

- ...and the certificate theorems

$$\vdash \texttt{run\_prog length\_ml length\_env}$$
$$\vdash \texttt{lookup\_var}\ \text{“length”}\ \texttt{length\_env} = \texttt{Some length\_v}$$
$$\vdash (\texttt{a LIST} \longrightarrow \texttt{NUM})\ \texttt{length length\_v}$$

Translator-generated functional specifications

$$\vdash \ (\mathsf{a}\ \texttt{LIST} \longrightarrow \ \texttt{NUM})\ \texttt{length}\ \texttt{length\_v}$$

- Relates the HOL function length to the closure value `length_v`
- Uses the "arrow" predicate "($a \longrightarrow b$) f fv"
  "For xv satisfying (a x), evaluating the closure with xv produces a value satisfying b (f x)"
- This gives a specification for `length_v`

Relating translator specifications and CF specifications

We prove equivalence between "arrow" specifications and a particular shape of CF specifications.

This allows:

- Using translated functions in CF-verified programs, and get a specification "for free"
- Provide programs certified using CF as drop-in replacements for translated functions

Relating translator specifications and CF specifications (2)

Formally, we prove:

$$\vdash \; (a \longrightarrow b)\; f\; fv \iff \\ \forall x\; xv.\; a\; x\; xv \;\Rightarrow\; \{\!|\mathtt{emp}|\!\}\; fv \cdot xv\; \{\!|\mathtt{POSTv}\; v.\; \langle b\,(f\;x)\;v\rangle|\!\}$$

A function satisfying such a spec:

- Can be called on any heap
- Cannot assume anything about the heap or access it
- Can still allocate heap objects (references, arrays,...) for internal use

Translator-CF interoperability applications

- Used to connect the purely functional part and the I/O part of the CakeML compiler
- “Translator $\to$ CF” direction is used pervasively in any non-trivial CF-verified program, e.g. for basic functions like $+$
- Future work: use “CF $\to$ translator” to implement more efficiently parts of the compiler e.g. the register allocator

Conclusion

- Verification framework for CakeML, with support for all language features
- Formal proof of characteristic formulae soundness