Verified Characteristic Formulae for CakeML

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

April 27, 2017

Goal: write programs in a **high-level (ML-style) language**, **prove** them **correct interactively**, and compile them using a **fully verified compilation chain**.

Main building blocks:

- The CakeML compiler (POPL'14, ICFP'16)
- Characteristic Formulae for ML (ICFP'11)

CakeML

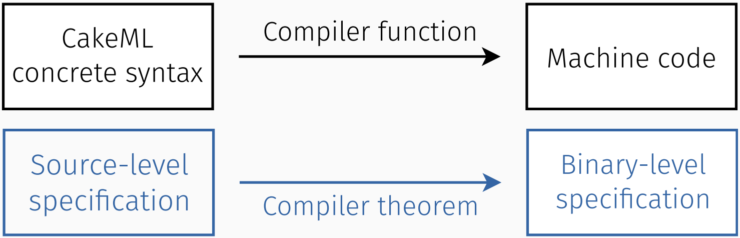
CAKEML
A Verified Implementation of ML

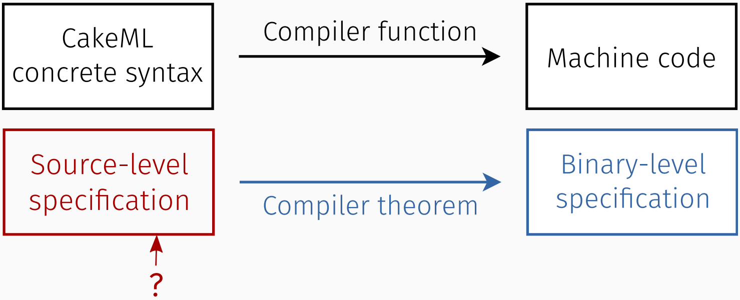
- Features: references, modules, datatypes, exceptions, a FFI, ...
- Missing features: functors, module nesting, records

CakeML

The CakeML compiler:

- Optimizing compiler
- Verified compiler: “Compcert for ML”
- Small trusted base: HOL kernel & machine axiomatisation
- Bootstraps (compiles itself)


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

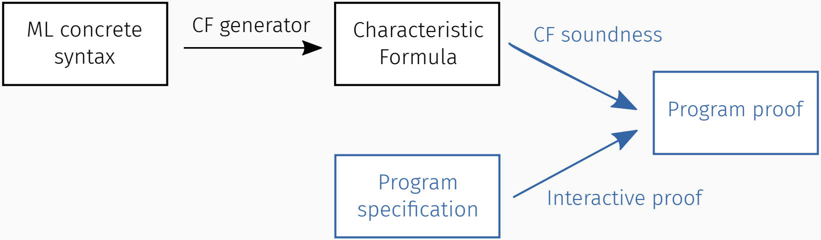

CakeML
concrete syntax
Compiler function
Machine code
Source-level
specification
Compiler theorem
Binary-level
specification
?

Characteristic Formulae for ML (CFML)

CFML:

- A tool to reason about ML programs. . .
- . . . using Separation Logic
- . . . in an interactive proof assistant (Coq).
- Smoothly integrate the SL reasoning rules into the higher-order logic, by turning all program variables into logical variables in one step
- Used to verify numerous non-trivial data-structures and algorithms (Union-Find, Dijkstra, Binary trees, Vectors, Hashtables...)

Characteristic Formulae for ML (CFML)



Connecting CakeML and Characteristic Formulae

What are the missing bits?

End-to-end verification of ML programs: the missing bits

Main challenge: realize CF axioms against the source code semantics

Other challenges:

- Add support for exceptions
- Add support for I/O
- Adapt from Coq to HOL
- Adapt CakeML translator to integrate into CF the connection between program values and logical values

Our Contribution

A program logic for CakeML

- State modular specifications about ML programs
- Prove them in a covenient way (using Separation Logic, following their CF)
- Theorem: the toplevel specification carries to the machine code produced by the CakeML compiler

Trusted codebase: HOL kernel, machine axiomatization

Outline

Background on CF

Soundness theorem: connecting CF to CakeML semantics

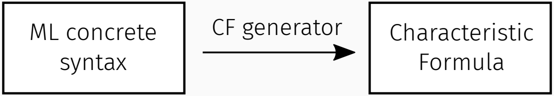
Extensions of CF

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

Interoperating with the proof-producing translator

Background on CF

How does the CF framework work?



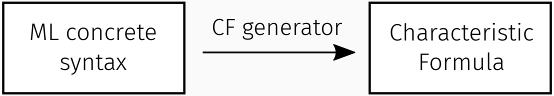
Main workhorse: the CF generator, "`cf`".

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

(`cf` e):

- logical formula; doesn't mention the syntax of e
- akin to a total correctness Hoare triple

How does the CF framework work? (2)



(`cf` e) 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)
- $H : \texttt{heap} \rightarrow \texttt{bool}$
- $Q : \texttt{v} \rightarrow \texttt{heap} \rightarrow \texttt{bool}$

Example: "let $x = e_1$ in e_2"

Hoare-logic rule:

$$\frac{env \vdash \{H\}\ e_1\ \{Q'\} \qquad \forall X.\ ((x, X) :: env) \vdash \{Q'\ X\}\ e_2\ \{Q\}}{env \vdash \{H\}\ (\texttt{Let}\ x\ e_1\ e_2)\ \{Q\}}$$

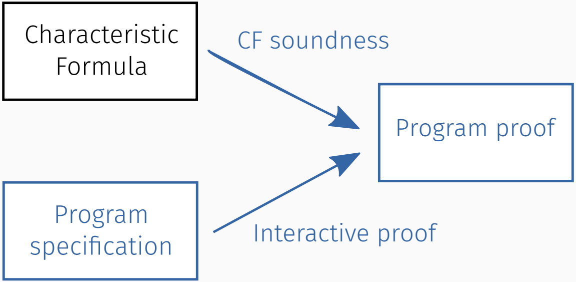
Characteristic Formula:

$$\begin{aligned} \texttt{cf}\ (\texttt{Let}\ x\ e_1\ e_2)\ env = \ & \texttt{local}\ (\lambda H\ Q. \\ & \ \exists\, Q'. \\ & \quad \texttt{cf}\ e_1\ H\ Q' \wedge \\ & \quad \forall\, X.\ \texttt{cf}\ e_2\ ((x, X) :: env)\ (Q'\ X)\ Q) \end{aligned}$$

Note: in practice, tactics are provided and the definition of `cf` is not shown to the user

Soundness theorem: connecting CF to CakeML semantics

Soundness of a CF framework



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

Connecting CF and CakeML views of the heap

CakeML view of the heap: list of store values.

```
state =
  <| clock : num
   ; refs : v store_v list
   ; ... |>

'a store_v =
    Refv of 'a
  | W8array of word8 list
  | Varray of 'a list
```

CF view of the heap: Separation Logic heap assertions.

Example: $(r_1 \rightsquigarrow v_1 * r_2 \rightsquigarrow v_2)$

Define heaps and heap predicates specialized for CakeML values:

- Type $\texttt{heap} = (\texttt{num} \mapsto \texttt{v store\_v})$
- Projection $\texttt{state\_to\_heap} : \texttt{state} \to \texttt{heap}$
- $r \rightsquigarrow v \;=\; (\lambda\, h.\, \exists\, loc.\, r = \mathrm{Loc}\ loc \wedge\ h = \{\ (loc,\ \texttt{Refv}\ v)\ \}\,)$

Connecting logical values to CakeML 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 re-use CakeML *refinement invariants*:

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

$$\vdash\; \mathtt{INT}\; i_0\; v_0 \;\wedge\; \mathtt{INT}\; i_1\; v_1 \;\Rightarrow$$
$$\{\mathtt{emp}\}\; \mathtt{plus\_v} \cdot [v_0;\; v_1]\; \{\lambda\, v.\; \langle \mathtt{INT}\,(i_0 \;+\; i_1)\; v\rangle\}$$

Realizing CFML axioms: Hoare-triple semantics

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
...
```

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

Realizing CFML axioms: Hoare-triple semantics

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

$$
\begin{array}{l}
env \vdash \{H\}\ e\ \{Q\} \iff \\
\quad \forall\, st\ H'. \\
\quad\quad (H * H')\ (\texttt{state\_to\_heap}\ st) \Rightarrow \\
\quad\quad\quad \exists\, v\ st'\ ck. \\
\quad\quad\quad\quad \texttt{evaluate}\ (st\ \text{with clock} := ck)\ env\ [e] = (st', \texttt{Rval}\ [v]) \wedge \\
\quad\quad\quad\quad (Q\ v * H' * true)\ (\texttt{state\_to\_heap}\ st')
\end{array}
$$

- H' accounts for the framed heap
- ck accounts for termination
- $true$ accounts for discarded memory cells

Proving CF soundness

Bridging the gap between:

Characteristic formulae

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

Big-step semantics

$$\begin{array}{l} \texttt{evaluate}\ \mathit{st}\ \mathit{env}\ [\texttt{Let}\ x\ \mathit{e1}\ \mathit{e2}] = \\ \quad \text{case}\ \texttt{evaluate}\ \mathit{st}\ \mathit{env}\ [\mathit{e1}]\ \text{of} \\ \quad\quad (\mathit{st}',\ \texttt{Rval}\ v) \Rightarrow \\ \quad\quad\quad \texttt{evaluate}\ \mathit{st}'\ ((x,\ \texttt{HD}\ v) :: \mathit{env})\ [\mathit{e2}] \\ \quad \mid\ (\mathit{st}',\ \texttt{Rerr}\ v) \Rightarrow (\mathit{st}',\ \texttt{Rerr}\ v) \end{array}$$

Proving CF soundness

Theorem (CF are sound wrt. CakeML semantics):

$$\vdash \ (\texttt{cf}\ e)\ \mathit{env}\ H\ Q \quad \Rightarrow \quad \mathit{env} \vdash \{H\}\ e\ \{Q\}$$

Proof: by induction on the size of e.

Corollary:

If "($\texttt{cf}$ e) *env* H Q" holds, then starting from a state satisfying H, evaluating e terminates with a value v and a new state satisfying $Q\ v$.

Extensions of CF

Extensions of CF

Support for I/O through the CakeML FFI

Performing I/O in CakeML

- Needed to interact with the external world (printing to `stdout`, opening files...)
- Done by calling external (C) code

CakeML I/O semantics

- State of the "external world" 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
- Modular proofs: need to be able to split the FFI state using "$*$" (proofs about `stdout` may be independent from proofs about the file-system...)

```
state =
  <| clock : num
   ; refs : v store_v list
   ; ffi : θ ffi_state
   ; ... |>
```

```
θ ffi_state =
<| oracle :
     string → θ → byte list →
     θ oracle_result
 ; ffi_state : θ
 ; ... |>
```

Problem: we know nothing about the type variable θ!

Splitting the FFI state

Solution:

- User provides: how to split the FFI state into *independent parts* (e.g. one part about `stdout`, one part about the file-system...)
- Eath part models a fraction of the external world
- Several external functions can update the same part
- FFI parts exposed in the heap, and are $*$-separated

$\Rightarrow$ Programs compose as long as their specifications agree on parts for external functions they both use

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
```

$\vdash$ LIST FILENAME *fns fnsv* $\wedge$
every (λ *fnm*. inFS_fname *fnm fs*) *fns* $\wedge$
numOpenFDs *fs* $< 255 \Rightarrow$
{CATFS *fs* $*$ STDOUT *out*}
cat_v $\cdot$ [*fnsv*]
{λ *u*.
$\langle$UNIT () *u*$\rangle$ $*$ CATFS *fs* $*$
STDOUT (*out* @ catfiles_string *fs fns*)}

Extensions of CF

Support for exceptions

Without support for exceptions:

- An expression must reduce to a value
- Post-conditions have type `v` $\rightarrow$ `heap` $\rightarrow$ `bool`

We now allow expressions to raise an exception:

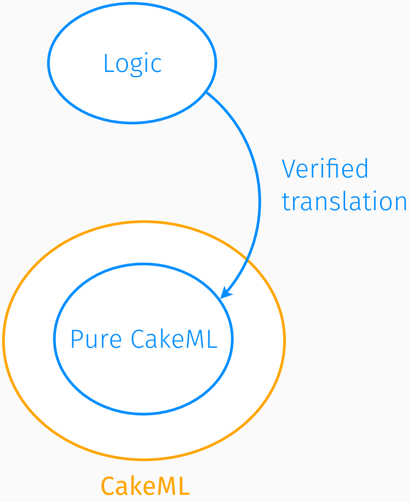
- Define datatype `res` = `Val v | Exn v`
- Post-conditions have type `res` $\rightarrow$ `heap` $\rightarrow$ `bool`
- Update the semantics and `cf` definitions accordingly...

Example: prove a more general specification for `cat`, which doesn't require that the input files exist.

Thanks to our tactics, programs without exceptions can be verified without additional effort.

Interoperating with the proof-producing translator

Existing tool: verified translation from HOL to CakeML



Produces theorems relating a (purely functional) piece of CakeML code with the corresponding HOL (pure) function.

Used to verify most of the compiler

Example: verified translation of length

- Define and verify the program in HOL4:

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

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

- The translator automatically produces CakeML code . . .

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

- . . . and the theorems

$\vdash$ run_prog length_ml length_env
$\vdash$ lookup_var "length_ml" length_env = Some length_v
$\vdash$ (a LIST $\longrightarrow$ NUM) length length_v

Relating translator specifications and CF specifications

We prove equivalence between translator specifications and a particular shape of CF specifications:

$$\vdash\ (a \longrightarrow b)\ f\ fv \iff \forall x\ xv.\ a\ x\ xv\ \Rightarrow\ \{\texttt{emp}\}\ fv \cdot xv\ \{\lambda\ v.\ \langle b\ (f\ x)\ v\rangle\}$$

Makes translated programs and **imperative** code **with empty input and output heap** interchangeable.

Interest:

- Get specifications "for free" for pure (translated) functions
- Allows for efficient imperative implementations of algorithms that realize logical functions

Conclusion

- End-to-end verification framework for CakeML
- With a CF generator that supports all the features of CakeML
- With support for formally relating a logical function with an efficient imperative implementation

Future work:

- Enhance tactics to further improve proof automation
- Prove correct more data-structures and algorithms (e.g. replace bits of the CakeML compiler with more efficient, imperative code)

$$\texttt{cf}\ (\texttt{Var}\ x)\ env = \quad \texttt{local}\ (\lambda H\ Q.\ \exists X.\ \texttt{lookup\_var\_id}\ x\ env = \texttt{Some}\ X \wedge H \rhd Q\ X)$$

$$\texttt{cf}\ (\texttt{Let}\ x\ e_1\ e_2)\ env = \quad \texttt{local}\ (\lambda H\ Q.\ \exists Q'.\ \texttt{cf}\ e_1\ H\ Q' \wedge \forall X.\ \texttt{cf}\ e_2\ ((x, X) :: env)\ (Q'\ X)\ Q)$$

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

Program specifications

Specifications:

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

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

Connecting CF and CakeML visions of the heap

Define heaps holding CakeML values:

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

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

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

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

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

Realizing CFML axioms: `app`

Semantics of Hoare-triples for unary application

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

$$
\begin{aligned}
&\{H\}\ f \cdot x\ \{Q\} \iff \\
&\quad \text{case } \texttt{do\_opapp}\ [f;\ x] \text{ of} \\
&\quad\ \ \texttt{None} \Rightarrow \forall\, st\ h_1\ h_2.\ \texttt{split}\ (\texttt{state\_to\_heap}\ p\ st)\ (h_1, h_2) \Rightarrow \neg H\ h_1 \\
&\quad \mid \texttt{Some}\ (env, exp) \Rightarrow env \vdash \{H\}\ exp\ \{Q\}
\end{aligned}
$$

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

$$
\begin{array}{l}
(\texttt{cf}\ e)\ \mathit{env}\ H\ Q\ \Rightarrow \\
\forall\, \mathit{st}. \\
\quad H\ (\texttt{state\_to\_heap}\ \mathit{st})\ \Rightarrow \\
\qquad \exists\, v\ \mathit{st}'\ \mathit{ck}. \\
\qquad\quad \texttt{evaluate}\ (\mathit{st}\ \text{with clock}\ :=\ \mathit{ck})\ \mathit{env}\ [e]\ =\ (\mathit{st}', \texttt{Rval}\ [v])\ \wedge \\
\qquad\quad (Q\ v * \mathit{true})\ (\texttt{state\_to\_heap}\ \mathit{st}')
\end{array}
$$

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, ...

Splitting the FFI state

Solution: parametrize `state_to_heap` 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_heap` 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:

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

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

Example: a more general specification for `cat1`

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

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

Exception-aware Hoare-triples

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

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

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:

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

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

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

CFs for `raise` and `handle`

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

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

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

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