*Program logics*, seventh lecture

# Logics for functional, higher-order languages

Xavier Leroy

2021-04-15

Collège de France, chair of software sciences

`xavier.leroy@college-de-france.fr`

# Which program logics for functional languages?

# Do we need a program logic for a functional language?

**No**, if the functions that can be defined in the language are also functions of the ambient logic:

- total functions (no divergence, no errors)
- without effects (no imperative features).

Example: functions definable in Coq or in Agda are objects of the ambient logic (type theory).

In this case, propositions and proofs from the ambient logic work just as well as Hoare triples:

$$\forall x, P\, x \Rightarrow Q\, x\, (f\, x) \text{ instead of } \{\, P \,\}\, f\, x\, \{\, Q \,\}$$

# Do we need a program logic for a functional language?

**Probably yes**, if the functional language has effects:

- divergence;
- run-time errors;
- mutable state, input/output;
- exceptions, continuations, algebraic effects, …

We can reason "manually" on effectful functional programs, typically via a monadic translation back to a pure functional language.

However, an appropriate program logic provides higher-level, more convenient tools for specification and verification.

# Example: reasoning about mutable state

We can represent an imperative computation in Coq as a state transformer: a pure function

$$\text{state “before”} \to \text{value} \times \text{state “after”}$$

Stating and proving properties of these computations is painful:

```
forall x s, valid x s ->
let (y, s') := f x s in
~valid y s /\ valid y s' /\ s' x = 0 /\ s' y = s x /\
(forall l, l <> x -> l <> y -> s' l = s l).
```

In separation logic, it suffices to write

$$\forall x, \{\, x \mapsto n \,\}\ f\ x\ \{\, \lambda y.\ x \mapsto 0 \star y \mapsto n \,\}$$

# Example: reasoning about non-termination

Several possible representations for computations that may not terminate (e.g. with general recursion) (lecture of 2020-01-30).

For example: Capretta's partiality monad (2005)

```
CoInductive delay (A: Type) : Type :=
  | now: A -> delay A
  | later: delay A -> delay A.
```

The weak triple $\{\, P \,\}\ c\ \{\, Q \,\}$ becomes $P \to$ `safe` $c\ Q$, where `safe` is the following coinductive predicate:

```
CoInductive safe {A: Type}: delay A -> (A -> Prop) -> Prop :=
  | safe_now: forall a Q, Q a -> safe (now a) Q
  | safe_later: forall c Q, safe c Q -> safe (later c) Q
```

# Outline

Two courses of action in this lecture:

- How can we extend Hoare logic and separation logic to deal with functions, including higher-order functions and functions as first-class values?
  Example: Iris.
- How can we use higher-order functions and dependent types to express program logics?
  Examples: F*, CFML.

# First-order procedures and functions in Hoare logic and in separation logic

# Procedures in Hoare logic

An early extension of Hoare's original logic.

A practical motivation: verifying Quicksort. (Foley and Hoare, 1971)

A principle of modular reasoning:

> Procedures support reusing code in several call contexts.
> Can we reuse the verification of this code? (instead of re-verifying it at each call context)

Clarifying the semantics of procedures: variable bindings, parameter passing mechanisms, etc.

# A reverse chronological presentation

Hoare logic rules for procedures are complicated, because they must control mutations over variables.

We follow Parkinson, Bornat and Calcagno (2006):

- First, we add procedures and functions to the PTR language (where variables are immutable but can be references to mutable memory cells), and give them separation logic rules.
- Second, for reference, we extend IMP with procedures and outline the corresponding rules in Hoare logic.

## Functions in PTR

Commands: $c ::= \dots$

$$\begin{array}{lll} & \mid \texttt{let}\ f\ (\vec{x}) = c\ \texttt{in}\ c' & \text{function definition} \\ & \mid f\ (\vec{a}) & \text{function call} \end{array}$$

These are imperative functions, in the style of C or ML: they can modify the state before returning a value.

Example: the *minmaxplus* function.

$$\begin{array}{l} \texttt{let}\ \mathit{minmaxplus}\ (x, y, m, M) = \\ \quad \texttt{if}\ x < y\ \texttt{then}\ (\texttt{set}(m, x); \texttt{set}(M, y)) \\ \quad \qquad \qquad \texttt{else}\ (\texttt{set}(m, y); \texttt{set}(M, x)); \\ \quad x + y \end{array}$$

# Specifying a function

Specification of the form $\{ P \} \, f \, (\vec{x}) \, \{ Q \}$ where $P$ and $Q$ are separation logic assertions.

Example: the *minmaxplus* function.

$$\begin{gathered} \{ m \mapsto \_ \star M \mapsto \_ \} \\ \mathit{minmaxplus}\,(x, y, m, M) \\ \{ \lambda v. \langle v = x + y \rangle \star m \mapsto \min(x, y) \star M \mapsto \max(x, y) \} \end{gathered}$$

Example: a function $\mathit{incr}(d)$ that adds $d$ to a global counter $c$ and return the previous value of $c$.

$$\forall \alpha, \; \{ c \mapsto \alpha \} \, \mathit{incr}\,(d) \, \{ \lambda v. \, \langle v = \alpha \rangle \star c \mapsto \alpha + d \}$$

# Rules for functions

A context $\Gamma$ = a set of function specifications.

Function calls:

$$\frac{(\{\, P \,\}\, f\ (\vec{x})\ \{\, Q \,\}) \in \Gamma}{\Gamma \vdash \{\, P[\vec{x} \leftarrow [\![\vec{a}]\!]] \,\}\, f\ (\vec{a})\ \{\, Q[\vec{x} \leftarrow [\![\vec{a}]\!]] \,\}}$$

Function definitions:

$$\frac{\begin{array}{c} \Gamma' = \Gamma,\ \{\, P \,\}\, f\ (\vec{x})\ \{\, Q \,\} \\ \forall \vec{x},\ \Gamma' \vdash \{\, P \,\}\ c\ \{\, Q \,\} \\ \Gamma' \vdash \{\, P' \,\}\ c'\ \{\, Q' \,\} \end{array}}{\Gamma \vdash \{\, P' \,\}\ \mathtt{let}\ f\ (\vec{x}) = c\ \mathtt{in}\ c'\ \{\, Q' \,\}}$$

# Hoare's rule for recursion

$$\frac{\Gamma, \ \{P\}\, f\,()\,\{Q\} \vdash \{P\}\, c\, \{Q\}}{\Gamma \vdash \{P\}\, f\,()\,\{Q\}}$$

Coinductive viewpoint: we can use the conclusion as an hypothesis, provided it is guarded by at least one rule.

Step-indexing viewpoint: to prove that the triple $\{P\}\, f\,()\,\{Q\}$ is valid for $n$ steps of computation, we can assume it is valid for $j < n$ steps.

Modal viewpoint: this is Löb's rule for the $\triangleright$ modality

$$\frac{Q \wedge \triangleright P \vdash P}{Q \vdash P}$$

## An example of verification

Using the specification $\{\, x \mapsto \_ \,\}$ $slowset\,(x, n)$ $\{\, x \mapsto n \,\}$

$$
\begin{array}{ll}
\texttt{let}\ slowset\,(x, n) = & \{\, x \mapsto \_ \,\} \\
\quad \texttt{if}\ n = 0\ \texttt{then} & \\
\qquad \texttt{set}(x, 0) & \{\, x \mapsto 0 \,\} \\
\quad \texttt{else} & \\
\qquad slowset\,(x, n - 1); & \{\, x \mapsto n - 1 \,\} \\
\qquad \texttt{let}\ v = \texttt{get}(x)\ \texttt{in}\ \texttt{set}(v, x + 1) & \{\, x \mapsto n \,\} \\
\texttt{in} & \\
 & \{\, a \mapsto \_ \ast b \mapsto \_ \,\} \\
\quad slowset(a, 2); & \{\, a \mapsto 2 \ast b \mapsto \_ \,\} \\
\quad slowset(b, 3) & \{\, a \mapsto 2 \ast b \mapsto 3 \,\}
\end{array}
$$

## Back to IMP and Hoare logic

Commands:

$$
\begin{array}{rlr}
c ::= & \ldots & \\
& \mid \texttt{local}\ x\ \texttt{in}\ c & \text{local variable} \\
& \mid \texttt{let}\ f\ (\texttt{var}\ \vec{x}; \texttt{val}\ \vec{y}) = c\ \texttt{in}\ c' & \text{procedure definition} \\
& \mid f\ (\vec{x}, \vec{a}) & \text{procedure call}
\end{array}
$$

Parameters $\vec{x}$ are passed by reference.
The corresponding arguments are variables.

Parameters $\vec{y}$ are passed by value.
The corresponding arguments are expressions.

Example: minimum and maximum.

$$
\begin{array}{l}
\texttt{let}\ \mathit{minmax}\ (\texttt{var}\ m, M; \texttt{val}\ x, y) = \\
\quad \texttt{if}\ x < y\ \texttt{then}\ (m := x; M := y)\ \texttt{else}\ (m := y; M := x)
\end{array}
$$

# Rules for procedures with parameters

The specification of a procedure is a triple with additional information on the variables used:

$$\{\, P \,\}\, f(\texttt{var}\ \vec{x}, \texttt{val}\ \vec{y})\ [\texttt{uses}\ \bar{u}, \texttt{modifies}\ \ \bar{v}]\ \{\, Q \,\}$$

$\bar{u}$ is the set of non-local variables mentioned (free) in $f$.

$\bar{v}$ is the set of non-local variables modified by $f$.

Procedure calls:

$$\frac{\begin{array}{c}(\{\, P \,\}\, f(\texttt{var}\ \vec{x}, \texttt{val}\ \vec{y})\ [\texttt{uses}\ \bar{u}, \texttt{modifies}\ \ \bar{v}]\ \{\, Q \,\}) \in \Gamma \\ \vec{w} \cap (\bar{u} \cup \bar{v}) = \emptyset\end{array}}{\Gamma \vdash \{\, \vec{\alpha} = \vec{a} \wedge P[\vec{x} \leftarrow \vec{w}, \vec{y} \leftarrow \vec{\alpha}] \,\}\ f\ (\vec{w}, \vec{a})\ \{\, Q[\vec{x} \leftarrow \vec{w}, \vec{y} \leftarrow \vec{\alpha}] \,\}}$$

# Rules for procedures with parameters

Procedure definitions:

$$\frac{\begin{array}{c}\Gamma' = \Gamma, \{\, P \,\}\, f\ (\texttt{var}\ \vec{x}, \texttt{val}\ \vec{y})\ [\texttt{uses}\ \bar{u}, \texttt{modifies}\ \bar{v}]\ \{\, Q \,\} \\ \bar{u} = \mathit{free}_\Gamma(c) \setminus (\vec{x} \cup \vec{y}) \qquad \bar{v} = \mathit{mods}_\Gamma(c) \setminus (\vec{x} \cup \vec{y}) \\ \vec{z} \cap \mathit{free}(P, Q, c, \vec{x}, \vec{y}) = \emptyset \\ \Gamma' \vdash \{\, P \,\}\ \texttt{local}\ \vec{z}\ \texttt{in}\ \vec{z} := \vec{y}; c[\vec{y} \leftarrow \vec{z}]\ \{\, Q \,\} \\ \Gamma' \vdash \{\, P' \,\}\ c'\ \{\, Q' \,\}\end{array}}{\Gamma \vdash \{\, P' \,\}\ \texttt{let}\ f\ (\texttt{var}\ \vec{x}; \texttt{val}\ \vec{y}) = c\ \texttt{in}\ c'\ \{\, Q' \,\}}$$

# Rules for local variables

The correct rule (= static scoping discipline):

$$\frac{\{\, P \,\}\ c[x \leftarrow y]\ \{\, Q \,\} \qquad y \notin \mathit{free}(c, P, Q)}{\{\, P \,\}\ \texttt{local}\ x\ \texttt{in}\ c\ \{\, Q \,\}}$$

An appealing but wrong rule (= dynamic scoping):

$$\frac{\{\, P[x \leftarrow y] \,\}\ c\ \{\, Q[x \leftarrow y] \,\} \qquad y \notin \mathit{free}(c, P, Q)}{\{\, P \,\}\ \texttt{local}\ x\ \texttt{in}\ c\ \{\, Q \,\}}$$

# Functions as first-class values in separation logic

# PTR with first-class functions

Expressions: $a ::= \dots$

$\quad\mid \texttt{rec}\, f\ x = c$ function abstraction

Commands: $c ::= a \mid \dots$

$\quad\mid a_1\ a_2$ function application

A nonrecursive function $\lambda x.\ c$ is handled as a recursive function $\texttt{rec}\, f\ x = c$ with $f$ not free in $c$.

Semantics: the familiar $\beta$-reduction rule.

$$(\texttt{rec}\, f\ x = c)\ a/h\ \rightarrow\ c[x \leftarrow [\![a]\!], f \leftarrow \texttt{rec}\, f\ x = c]/h$$

## Hoare triples as assertions

Assertions, preconditions:

$$\begin{aligned} P \ ::= \ & \langle A \rangle \mid \texttt{emp} \mid \ell \mapsto v \mid P_1 \ast P_2 \mid \ldots \\ & \mid \{ P \}\, c\, \{ Q \} \end{aligned}$$

Hoare triple

Postconditions:

$$Q ::= \lambda v.\ P$$

Triple assertions can be duplicated:

$$\{ P \}\, c\, \{ Q \} = \{ P \}\, c\, \{ Q \} \ast \{ P \}\, c\, \{ Q \}$$

# Rules for functions

Recursive abstraction:

$$\frac{\begin{array}{l}\forall v, \{\, P \,\}\ (\texttt{rec}\ f\ x = c)\ v\ \{\, Q \,\} \Rightarrow \\ \qquad\qquad \forall v, \{\, P \,\}\ c[x \leftarrow v, f \leftarrow \texttt{rec}\ f\ x = c]\ \{\, Q \,\}\end{array}}{\forall v, \{\, P \,\}\ (\texttt{rec}\ f\ x = c)\ v\ \{\, Q \,\}}$$

Nonrecursive abstraction (derived rule):

$$\frac{\{\, P \,\}\ c[x \leftarrow v]\ \{\, Q \,\}}{\{\, P \,\}\ (\lambda x.c)\ v\ \{\, Q \,\}}$$

Moving the triple in / out of the precondition:

$$\frac{(\forall \vec{v},\ \{\, P_1 \,\}\ c_1\ \{\, Q_1 \,\}) \Rightarrow \{\, P_2 \,\}\ c_2\ \{\, Q_2 \,\}}{\{\, (\forall \vec{v},\ \{\, P_1 \,\}\ c_1\ \{\, Q_1 \,\}) \star P_2 \,\}\ c_2\ \{\, Q_2 \,\}}$$

# Specifying a higher-order function

Consider the function $app = \lambda f.\, f\ 0$.

We would like to give it the following specification:
"if $f$ is positive valued, $app\ f$ returns a positive number".

Writing $Q = \lambda x.\, \langle x > 0 \rangle$ the postcondition "returns a positive number", we can derive

$$\frac{(\forall v, \{\, \mathtt{emp} \,\}\ f\ v\ \{\, Q \,\}) \Rightarrow \{\, \mathtt{emp} \,\}\ f\ 0\ \{\, Q \,\}}{\{\, \forall v, \{\, \mathtt{emp} \,\}\ f\ v\ \{\, Q \,\} \,\}\ app\ f\ \{\, Q \,\}}$$

# Representing an object with an internal state

```
class Counter {
    private int val;
    Counter() { val = 0 }
    int curr() { return val; }
    void incr() { val += 1; }
}
```

An implementation in PTR:

$$
\begin{aligned}
&\texttt{let}\ \mathit{mkpair} = \lambda x.\ \lambda y. \\
&\qquad \texttt{let}\ p = \texttt{alloc}(2)\ \texttt{in}\ \texttt{set}(p, x); \texttt{set}(p + 1, y); p\ \texttt{in} \\
&\texttt{let}\ \mathit{counter} = \lambda\_. \\
&\qquad \texttt{let}\ \mathit{val} = \texttt{alloc}(1)\ \texttt{in} \\
&\qquad \mathit{mkpair}\ (\lambda\_.\ \texttt{get}(\mathit{val})) \\
&\qquad\qquad (\lambda\_.\ \texttt{let}\ n = \texttt{get}(\mathit{val})\ \texttt{in}\ \texttt{set}(\mathit{val}, n + 1))
\end{aligned}
$$

# Representation predicate

We define the predicate $\mathit{Counter}(p, n)$, "at location $p$ there is a counter whose current value is $n$", as follows:

$$\begin{aligned}\exists \mathit{curr}, \mathit{incr}, \mathit{val},\ & p \mapsto \mathit{curr} \ast p + 1 \mapsto \mathit{incr} \ast \mathit{val} \mapsto n \\ & \ast\ \{\, \mathit{val} \mapsto n \,\}\ \mathit{curr}\ ()\ \{\, \lambda v.\ \langle v = n \rangle \ast \mathit{val} \mapsto n \,\} \\ & \ast\ \{\, \mathit{val} \mapsto n \,\}\ \mathit{incr}\ ()\ \{\, \lambda\_.\ \mathit{val} \mapsto n + 1 \,\}\end{aligned}$$

We can then prove

$$\begin{array}{rcl} \{\, \mathtt{emp} \,\} & \mathit{counter}\ () & \{\, \lambda p.\ \mathit{Counter}(p, 0) \,\} \\ \{\, \mathit{Counter}(p, n) \,\} & \mathtt{get}(p)\ () & \{\, \lambda v.\ \langle v = n \rangle \ast \mathit{Counter}(p, n) \,\} \\ \{\, \mathit{Counter}(p, n) \,\} & \mathtt{get}(p + 1)\ () & \{\, \lambda\_.\ \mathit{Counter}(p, n + 1) \,\} \end{array}$$

# Semantic soundness of the rule for recursion

$$\frac{\begin{array}{l}\forall v, \{\,P\,\}\ (\mathtt{rec}\, f\ x = c)\ v\ \{\,Q\,\} \Rightarrow \\ \qquad\qquad \forall v, \{\,P\,\}\ c[x \leftarrow v, f \leftarrow \mathtt{rec}\, f\ x = c]\ \{\,Q\,\}\end{array}}{\forall v, \{\,P\,\}\ (\mathtt{rec}\, f\ x = c)\ v\ \{\,Q\,\}}$$

Following our usual semantic approach, to prove the conclusion, we study the reductions of the command:

$$(\mathtt{rec}\, f\ x = c)\ v/h \rightarrow c[x \leftarrow v, f \leftarrow \mathtt{rec}\, f\ x = c]$$

The premise gives us a semantic triple for $c[x \leftarrow v, f \leftarrow \mathtt{rec}\, f\ x = c]$, but only if we have already proved

$$\forall v, \{\,P\,\}\ (\mathtt{rec}\, f\ x = c)\ v\ \{\,Q\,\}$$

that is, the desired result! This is circular reasoning!

# Step-indexing to the rescue

Idea: in the definition of the semantic Hoare triple

$$\{\{\, P \,\}\}\ c\ \{\{\, Q \,\}\} = \forall n,\ h,\ P\ h \Rightarrow \mathtt{Safe}^n\ c\ h\ Q$$

a function call within $c$ consumes one reduction step. Therefore, the function being called needs to be safe for $n-1$ steps at most.

Consequently, Hoare triples appearing in precondition $P$ only need to be true "at depth $n-1$", not absolutely true.

# Step-indexing to the rescue

An implementation of this idea: we index assertions by a step count $n$. For the usual assertions, this count is ignored:

$$\langle A \rangle\ h\ n = Dom(h) = \emptyset \land A$$

$$(\ell \mapsto v)\ h\ n = Dom(h) = \{\ell\} \land h\ \ell = v$$

but it is taken into account for “triple” assertions

$$(\{\,P\,\}\ c\ \{\,Q\,\})\ h\ 0 = Dom(h) = \emptyset$$

$$(\{\,P\,\}\ c\ \{\,Q\,\})\ h\ (n+1) = Dom(h) = \emptyset \land \forall h',\ P\ h'\ n \Rightarrow \mathtt{Safe}^{n+1}\ c\ h'\ Q$$

The semantic triple, then, becomes

$$\{\{\,P\,\}\}\ c\ \{\{\,Q\,\}\} = \forall n > 0, \forall h,\ P\ h\ (n-1) \Rightarrow \mathtt{Safe}^{n}\ c\ h\ Q$$

# The $\triangleright$ modality to the rescue

An alternative to step-indexing is to use a modal logic with the $\triangleright$ modality (“later”).

This modality supports proofs by Löb induction:

$$\frac{Q \wedge \triangleright P \vdash P}{Q \vdash P}$$

It also supports the definition of recursive predicates of the form

$$P\,x = \ldots \triangleright P\,y \ldots \triangleright P\,z \ldots$$

# The $\triangleright$ modality to the rescue

In particular, we can define the predicate $\mathtt{Safe}\ c\ h\ Q$
("if $c/h$ terminates, the final state satisfies $Q$")
without step-indexing, simply as

$$\begin{aligned}\mathtt{Safe}\ c\ h\ Q = (&c = a \Rightarrow Q\ [\![a]\!]\ h)\\ &\wedge (c/h \not\to \mathtt{err})\\ &\wedge (\forall c', h', c/h \to c'/h' \Rightarrow \triangleright \mathtt{Safe}\ c'\ h'\ Q)\end{aligned}$$

This definition of $\mathtt{Safe}$ and of the semantic triple validates the rule for recursive functions $\mathtt{rec}\ f\ x = c$, by Löb induction.

# More powerful rules

In the rules that correspond to an actual computation step, we can weaken the precondition from $P$ to $\triangleright P$.
(This lets us prove more results by Löb induction.)

$$\frac{\forall v, \{\, P \,\}\ (\mathtt{rec}\, f\ x = c)\ v\ \{\, Q \,\} \Rightarrow \quad \forall v, \{\, P \,\}\ c[x \leftarrow v, f \leftarrow \mathtt{rec}\, f\ x = c]\ \{\, Q \,\}}{\forall v, \{\, \triangleright P \,\}\ (\mathtt{rec}\, f\ x = c)\ v\ \{\, Q \,\}}$$

$$\frac{\{\, P \,\}\ c[x \leftarrow v]\ \{\, Q \,\}}{\{\, \triangleright P \,\}\ (\lambda x.c)\ v\ \{\, Q \,\}}$$

$$\{\, \triangleright \ell \mapsto v \,\}\ \mathtt{get}(\ell)\ \{\, \lambda v'.\ \langle v' = v \rangle \ast \ell \mapsto v \,\}$$

$$\{\, \triangleright \ell \mapsto \_ \,\}\ \mathtt{set}(\ell, v)\ \{\, \lambda \_.\ \ell \mapsto v \,\}$$

# CFML: reasoning about ML programs using characteristic formulas

# Characteristic formulas for pure programs

The characteristic formula $[\![t]\!]$ of a term $t$ is its weakest precondition calculus: $[\![t]\!]\ Q = wp(t, Q)$.

$$[\![t]\!] : \underbrace{(\lceil\tau\rceil \rightarrow \texttt{Prop})}_{\text{postcondition}} \rightarrow \underbrace{\texttt{Prop}}_{\text{precondition}} \qquad \text{if } t : \tau$$

Some representative cases:

$$\begin{aligned}
[\![v]\!] &= \lambda Q.\ Q\ \lceil v\rceil \\
[\![\texttt{fail}]\!] &= \lambda Q.\ \bot \\
[\![\texttt{let}\ x = t\ \texttt{in}\ t']\!] &= \lambda Q.\ \exists R.\ [\![t]\!]\ R \wedge (\forall x, R\ x \Rightarrow [\![t']\!]\ Q) \\
[\![\texttt{if}\ v\ \texttt{then}\ t_1\ \texttt{else}\ t_2]\!] &= \lambda Q.\ (\lceil v\rceil \Rightarrow [\![t_1]\!]\ Q) \wedge (\neg\lceil v\rceil \Rightarrow [\![t_2]\!]\ Q)
\end{aligned}$$

# Characteristic formulas for pure programs

The actual definition uses combinators to reflect the program structure in the characteristic formula:

$$[\![v]\!] = \mathtt{Ret}\ \lceil v \rceil \qquad [\![f\ v]\!] = \mathtt{App}\ \lceil f \rceil\ \lceil v \rceil \qquad [\![\mathtt{fail}]\!] = \mathtt{Fail}$$

$$[\![\mathtt{let}\ x = t\ \mathtt{in}\ t']\!] = \mathtt{Let}\ x = [\![t]\!]\ \mathtt{In}\ [\![t']\!]$$

$$[\![\mathtt{if}\ v\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2]\!] = \mathtt{If}\ \lceil v \rceil\ \mathtt{Then}\ [\![t_1]\!]\ \mathtt{Else}\ [\![t_2]\!]$$

where the combinators are defined as

$$\mathtt{Ret}\ V = \lambda Q.\ Q\ V \qquad \mathtt{App}\ F\ V = \mathit{AppReturns}\ F\ V \qquad \mathtt{Fail} = \lambda Q.\ \bot$$

$$\mathtt{Let}\ x = F\ \mathtt{In}\ F' = \lambda Q.\ \exists R,\ F\ R \wedge (\forall x,\ R\ x \Rightarrow F'\ Q)$$

$$\mathtt{If}\ V\ \mathtt{Then}\ F\ \mathtt{Else}\ F' = \lambda Q.\ (V \Rightarrow F\ Q) \wedge (\neg V \Rightarrow F'\ Q)$$

# Example of characteristic formula

```
let rec half x =
  if x = 0 then 0 else if x = 1 then fail
  else let y = half (x - 2) in y + 1
```

The body of function `half` becomes

$$\begin{aligned}&\mathtt{If}\ x = 0\ \mathtt{Then}\ \mathtt{Ret}\ 0\ \mathtt{Else}\ \mathtt{If}\ x = 1\ \mathtt{Then}\ \mathtt{Fail}\\ &\mathtt{Else}\ \mathtt{Let}\ y = \mathtt{App}\ \mathit{half}\ (x - 2)\ \mathtt{In}\ \mathtt{Ret}\ (y + 1)\end{aligned}$$

that is,

$$\begin{aligned}&\lambda Q.\ (x = 0 \Rightarrow Q\ 0) \land (x \neq 0 \Rightarrow\\ &\quad (((x = 1) \Rightarrow \bot) \land (x \neq 1 \Rightarrow\\ &\qquad \exists R,\ \mathit{AppReturns}\ \mathit{half}\ (x - 2)\ R \land (\forall y, R\ y \Rightarrow Q(y + 1)))))\end{aligned}$$

# Representing functions

A function is represented by a value of the abstract type *Func*.

The *AppReturns* operator associates a characteristic formula to each function:

$$AppReturns : \forall A, B,\ Func \to A \to (B \to \texttt{Prop}) \to \texttt{Prop}$$

In other words, $AppReturns\, f\ v\ Q$ is the precondition of application $f\ v$ with postcondition $Q$.

Each global function definition $\texttt{let rec}\, f\ x = t$ introduces a fresh constant $f : Func$ and an axiom

$$\forall x, Q,\ [\![t]\!]\ Q \Rightarrow AppReturns\, f\ x\ Q$$

# Specifying functions

A function specification of the form $\{ P \}\, f\, x\, \{ Q \}$ is expressed as a lemma about $\mathit{AppReturns}\, f$:

$$\forall x, P\, x \Rightarrow \mathit{AppReturns}\, f\, x\, Q$$

In the previous example:

```
let rec half x =
  if x = 0 then 0 else if x = 1 then fail
  else let y = half (x - 2) in y + 1
```

Here are two plausible specifications:

$$\forall n, n \geq 0 \Rightarrow \mathit{AppReturns}\, f\, (2 \times n)\, (\lambda v.\ v = n)$$
$$\forall n, n \geq 0 \wedge \mathtt{even}(n) \Rightarrow \mathit{AppReturns}\, f\, n\, (\lambda v.\ v = n/2)$$

# Specifying higher-order functions

A parameter $f$ that is a function is specified via hypotheses on *AppReturns* $f$.

$$\texttt{let}\ app\ f = f\ 0$$

A specification: "if $f$ is positive valued, then $app\ f$ returns a positive number".

$$\forall f, (\forall x, AppReturns\, f\, x\, (\lambda v.\, v \geq 0)) \Rightarrow AppReturns\, app\, f\, (\lambda v.\, v \geq 0)$$

A more precise specification: "$app\ f$ satisfies all the postconditions that $f\ 0$ satisfies".

$$\forall f, Q,\ AppReturns\ f\ 0\ Q \Rightarrow AppReturns\ app\ f\ Q$$

# Characteristic formulas for imperative programs

The full CFML system also handles imperative ML programs (with references to mutable state).

Preconditions and posconditions use separation logic assertions $\mathit{heap} \to \mathtt{Prop}$ instead of propositions $\mathtt{Prop}$.

Characteristic formulas are no longer a weakest precondition calculus (functions postcondition $\to$ precondition), but relations between preconditions and postconditions:

$$[\![t]\!] : \underbrace{(\mathit{heap} \to \mathtt{Prop})}_{\text{precondition}} \to \underbrace{(\lceil\tau\rceil \to \mathit{heap} \to \mathtt{Prop})}_{\text{postcondition}} \to \mathtt{Prop} \qquad \text{if } t : \tau$$

# F*: dependent types and monads for verification

# Dependent types, preconditions, postconditions

In a dependently-typed functional language (such as Agda, Coq, F*), we can write types that express both value types and logical propositions:

| | |
|---|---|
| $\forall x : A.\ P(x) \rightarrow B$ | functions taking an $x : A$ and a proof of $P(x)$ |
| $\{\, y : A \mid Q(y) \,\}$ | pairs of a $y : A$ and a proof of $Q(y)$ |

**Example (a precise type for the "square root" function)**

$$\forall n : \mathbb{Z},\ n \geq 0 \rightarrow \{\, r : \mathbb{Z} \mid r \geq 0 \wedge r^2 \leq n < (r+1)^2 \,\}$$

# A type of Hoare triples

Idea: use dependent types to define a type $M\ P\ A\ Q$ of computations $c$ of type $A$ that satisfy the triple $\{\,P\,\}\ c\ \{\,Q\,\}$.

For pure computations, we take

$$M\,(P : \mathtt{Prop})\,(A : \mathtt{Type})\,(Q : A \to \mathtt{Prop}) : \mathtt{Type} := P \to \{\, a : A \mid Q\, a \,\}$$

This type is a monad, with the monadic operations

$$\begin{aligned} \mathtt{ret}\, v &= \lambda p.\ \langle v, p \rangle \\ \mathtt{bind}\, m\, f &= \lambda p.\ \mathtt{let}\ \langle v, q \rangle = m\ p\ \mathtt{in}\ f\ x\ q \end{aligned}$$

# Hoare monads

The interesting aspect of these monadic operations is their types:

$$\mathtt{ret} : \forall (A : \mathtt{Type})\ (a : A)(Q : A \to Prop),\ M\ (Q\ v)\ A\ Q$$

$$\begin{aligned}\mathtt{bind} : \forall (A\ B\ C : \mathtt{Type})\ (P : \mathtt{Prop})\ (Q : A \to \mathtt{Prop})\ (R : B \to \mathtt{Prop}),\\ M\ P\ A\ Q \to (\forall x : A, M\ (Q\ x)\ B\ R) \to M\ P\ A\ R\end{aligned}$$

These types correspond exactly to rules of Hoare logic
(in the style of the PTR language):

$$\{\, Q\ [\![a]\!]\, \}\ a\ \{\, Q \,\} \qquad \frac{\{\, P \,\}\ c\ \{\, Q \,\} \qquad \forall x,\ \{\, Q\ x \,\}\ c'\ \{\, R \,\}}{\{\, P \,\}\ \mathtt{let}\ x = c\ \mathtt{in}\ c'\ \{\, R \,\}}$$

# “The” Hoare monad: mutable state

(Nanevski *et al*, Hoare Type Theory (2006); Ynot (2008))

If *State* is the type of states, the usual state monad is

$ST\ A = State \to A \times State$ (state “before” $\to$ value, state “after”)

The corresponding Hoare monad is

$$ST\ P\ A\ Q = \forall s : State, P\ s \to \{\ (a, s') \mid Q\ a\ s'\ \}$$

with $P : State \to \texttt{Prop}$ and $Q : A \to State \to \texttt{Prop}$
(assertions about the state).

`ret` and `bind` have their usual types.

# Imperative operations

We can give types to mutable state operations that correspond to the "large rules" of separation logic:

$$
\begin{aligned}
\texttt{get}\ \ell &: \forall v, R, ST\ (\ell \mapsto v \ast R)\ \texttt{Z}\ (\lambda r.\ \langle r = v \rangle \ast \ell \mapsto v \ast R) \\
\texttt{set}\ \ell\ v &: \forall R, ST\ (\ell \mapsto \_ \ast R)\ \texttt{unit}\ (\lambda \_.\ \ell \mapsto v \ast R) \\
\texttt{alloc} &: \forall R, ST\ R\ \texttt{addr}\ (\lambda \ell.\ \ell \mapsto \_ \ast R) \\
\texttt{free}\ \ell &: \forall R, ST\ (\ell \mapsto \_ \ast R)\ \texttt{unit}\ (\lambda \_.\ R)
\end{aligned}
$$

# A separation monad

We can recover the "small rules" and gain the frame rule by quantifying over all frames:

$$\mathit{STsep}\ P\ A\ Q = \forall R,\ \mathit{ST}\ (P \star R)\ A\ (\lambda v.\ Q\ v \star R)$$

The frame rule corresponds to a retyping function:

$$\texttt{frame}\ R : \mathit{STsep}\ P\ A\ Q \to \mathit{STsep}\ (P \star R)\ A\ (\lambda v.\ Q\ v \star R)$$

The "small rules" are here:

$$\begin{aligned}
\texttt{ret}\ v &: \mathit{STsep}\ \texttt{emp}\ A\ (\lambda r.\ \langle r = v\rangle) \\
\texttt{get}\ \ell &: \forall v, \mathit{STsep}\ (\ell \mapsto v)\ \texttt{Z}\ (\lambda r.\ \langle r = v\rangle \star \ell \mapsto v) \\
\texttt{set}\ \ell\ v &: \mathit{STsep}\ (\ell \mapsto _)\ \texttt{unit}\ (\lambda _.\ \ell \mapsto v) \\
\texttt{alloc} &: \mathit{STsep}\ \texttt{emp}\ \texttt{addr}\ (\lambda \ell.\ \ell \mapsto _) \\
\texttt{free}\ \ell &: \mathit{STsep}\ (\ell \mapsto _)\ \texttt{unit}\ \texttt{emp}
\end{aligned}$$

# Relational Hoare monad

For reference: the Ynot system of Nanevsky *et al* encodes an relational Hoare logic, where the postcondition relates the initial state and the final state:

$$\mathit{STrel}\ P\ A\ Q = \forall s,\ \ P\ s \rightarrow \{\,(a, s')\mid Q\ a\ s\ s'\,\}$$

with $Q : A \rightarrow \mathit{State} \rightarrow \mathit{State} \rightarrow \texttt{Prop}$.

This avoids using auxiliary variables in some rules, but complicates the type of `bind`:

$$\begin{aligned}\texttt{bind} :\ & \forall A, B, P_1, Q_1, P_2, Q_2,\\ & \quad \mathit{STrel}\ P_1\ A\ Q_1 \rightarrow (\forall (a : A), \mathit{STrel}\ (P_2\ a)\ B\ (Q_2\ a)) \rightarrow \mathit{STrel}\ P\ B\ Q\end{aligned}$$

with $P = \lambda s_1.\ P_1\ s_1 \wedge \forall a, s_2.\ Q_1\ a\ s_1\ s_2 \Rightarrow P_2\ s_2$
and $Q = \lambda b, s_1, s_3.\ \exists a, s_2.\ Q_1\ a\ s_1\ s_2 \wedge Q_2\ a\ b\ s_2\ s_3$.

# Summary on Hoare monads

It's the "program and verify at the same time" approach promoted by dependent types, implemented so that

- we can use effects;
- programming is done in a monadic style;
- verification is done in a Hoare logic style.

The embedding in Coq (the Ynot system) is hard to use:

- little inference of intermediate assertions;
- need retyping functions to materialize purely logical rules (consequence, frame):

$$\texttt{cons\_pre} : (P' \to P) \to ST\ P\ A\ Q \to ST\ P'\ A\ Q$$

# The F* approach

The F* language also uses dependent types to program and to verify in the presence of effects, but with a slightly different approach:

- Dijkstra monads instead of Hoare monads ($\approx$ weakest precondition calculus instead of triples).
- A custom type-checker that infers verification conditions and solves them automatically if possible.
- A hierarchy of effects and monads, making it possible to handle each part of the program with the minimum amount of effects.

## Dijkstra monads

Idea: for a computation $c$, instead of triples $\{\, P \,\}\ c\ \{\, Q \,\}$, consider the predicate transformers $W : POST \to PRE$ and the triples $\{\, W\ Q \,\}\ c\ \{\, Q \,\}$ for all postconditions $Q$.

Example: the state monad.

$$
\begin{aligned}
PRE &= State \to \texttt{Prop} \\
POST\ A &= A \to State \to \texttt{Prop} \\
TRANSF\ A &= POST\ A \to PRE \\
ST\ A\ (W : TRANSF\ A) &= \forall Q, s,\ W\ Q\ s \to \{\, (a, s') \mid Q\ a\ s' \,\}
\end{aligned}
$$

The type $ST\ A\ W$ is the type of monadic computations producing a value of type $A$ and validating the “contract” $W$.

# The operations of the Dijkstra state monad

$$RET\ (v : A) : TRANSF\ A = \lambda Q.\ Q\ v$$

$$\texttt{ret}\ (v : A) : ST\ A\ (RET\ v) = \lambda Q, s, p, \langle (v, s), p \rangle$$

For `bind`, with $W_1 : TRANSF\ A$ and $W_2 : A \to TRANSF\ B$ and $m : ST\ A\ W_1$ and $f : \forall a : A, ST\ B\ (W_2\ a)$,

$$BIND\ W_1\ W_2 : TRANSF\ B = \lambda Q.\ W_1\ (\lambda a.\ W_2\ a\ Q)$$

$$\texttt{bind}\ m\ f : ST\ A\ (BIND\ W_1\ W_2) = \ldots$$

Remark: the types of `ret` and `bind` always have the form above for all Dijkstra monads; only the operators *RET*, *BIND* change.

Remark: *RET* and *BIND* also form a (continuation) monad!

# The operations of the Dijkstra state monad

Operations on memory follow the same pattern:

$$
\begin{aligned}
GET\ \ell &: TRANSF\ Z = \lambda Q, s,\ \ell \in Dom(s) \wedge Q\ (s\ \ell)\ s \\
\mathtt{get}\ \ell &: ST\ Z\ (GET\ \ell) \\
SET\ \ell\ v &: TRANSF\ \mathtt{unit} = \lambda Q, s,\ \ell \in Dom(s) \wedge Q\ ()\ s[\ell \leftarrow v] \\
\mathtt{set}\ \ell\ v &: ST\ \mathtt{unit}\ (SET\ \ell) \\
ALLOC &: TRANSF\ \mathtt{addr} = \lambda Q, s,\ \forall \ell \notin Dom(s), Q\ \ell\ s[\ell \leftarrow 0] \\
\mathtt{alloc} &: ST\ \mathtt{addr}\ ALLOC \\
FREE\ \ell &: TRANSF\ \mathtt{unit} = \lambda Q, s,\ \ell \in Dom(s) \wedge Q\ ()\ (s \setminus \ell) \\
\mathtt{free}\ \ell &: ST\ \mathtt{unit}\ (FREE\ \ell)
\end{aligned}
$$

Remark: we can define `get`, `set`, …, in accordance with our definition of *ST*; but we can also leave these operations abstract, which leads to an axiomatization of a built-in “mutable state” effect.

# The Dijkstra monad for exceptions

Postconditions describe both kinds of results: normal results and exceptional results.

$$
\begin{aligned}
PRE &= \texttt{Prop} \\
POST\ A &= (A + \texttt{exn}) \to \texttt{Prop} \\
TRANSF\ A &= POST\ A \to PRE \\
EXN\ A\ W &= \forall Q : POST\ A,,\ W\ Q \to \{\, r \mid Q\ r \,\} \\
RET\ v &= \lambda Q.\ Q\ (\texttt{left}\ v) \\
BIND\ W_1\ W_2 &= \lambda Q.\ W_1\ (\lambda r.\ \texttt{match}\ r\ \texttt{with} \\
&\qquad\qquad\qquad\qquad\qquad \mid \texttt{left}\ v \Rightarrow W_2\ v\ Q \\
&\qquad\qquad\qquad\qquad\qquad \mid \texttt{right}\ e \Rightarrow Q\ (\texttt{right}\ e))
\end{aligned}
$$

# A hierarchy of monads

$$
\begin{tikzcd}
 & PURE \arrow[d] \arrow[r] & GHOST \\
 & DIV \arrow[dl] \arrow[dr] & \\
ST \arrow[dr] & & EXN \arrow[dl] \\
 & ALL &
\end{tikzcd}
$$

Each arrow corresponds to a monad transformer, for example

$$EXN\ A\ W \rightarrow ALL\ A\ (EXN\_to\_ALL\ W)$$

# Inferring the smallest monad = effect inference

Computations are automatically placed in the smallest monad they need.

Example: the `let` rule for sequencing and binding.

$$\frac{\Gamma \vdash e_1 : M_1\ \tau_1\ W_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : M_2\ \tau_2\ W_2 \qquad\qquad M = M_1 \sqcup M_2 \qquad W_1' = M_1\_to\_M\ W_1 \qquad W_2' = M_2\_to\_M\ W_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : M\ \tau_2\ (M.BIND\ W_1'\ (\lambda x.\ W_2'))}$$

# Summary

# Summary

A nice example of program logic for a functional language: F* and its applications to the verification of cryptographic libraries.

Other approaches are possible, such as CFML and Iris.
No consensus.

Higher-order functions (`map`, `iter`, `fold`, ...) are difficult to specify, especially in conjunction with mutable state.

# A puzzle

The “awkward example” of Pitts and Stark:

```
let awkward =
  let r = ref 0 in
  fun f -> assert (!r mod 2 = 0); incr r; f(); incr r
```

The assertion fails if `awkward` is applied to itself…

What specifications can we give to `awkward`?

# References

# References

The F* language: `https://www.fstar-lang.org/`

The CFML system: `https://www.chargueraud.org/softs/cfml/`

Functions as first-class values in separation logic:

- L. Birkedal, A. Bizjak, *Lecture Notes on Iris: Higher-Order Concurrent Separation Logic*, chapters 4 to 6.

THE END