Logics for functional, higher-order languages

Xavier Leroy

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?

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).

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:

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, ...

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

$$\text{state "before"} \rightarrow \text{value} \times \text{state "after"}$$

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).
```

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

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

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.

Functions in PTR

Commands: $c ::= \ldots$

$\mid \texttt{let}\ f\ (\vec{x}) = c\ \texttt{in}\ c'$	function definition
$\mid f\ (\vec{a})$	function call

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

Functions in PTR

Commands: $c ::= \ldots$

$$\mid \texttt{let}\ f\ (\vec{x}) = c\ \texttt{in}\ c' \quad \text{function definition}$$

$$\mid f\ (\vec{a}) \quad \text{function call}$$

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

Example: the *minmaxplus* function.

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

Specifying a function

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

Specifying a function

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

Example: the *minmaxplus* function.

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

Specifying a function

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

Example: the *minmaxplus* function.

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

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 Γ = a set of function specifications.

Function calls:

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

Rules for functions

A context Γ = a set of function specifications.

Function calls:

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

Rules for functions

A context Γ = a set of function specifications.

Function calls:

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

Rules for functions

A context Γ = a set of function specifications.

Function calls:

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

Rules for functions

A context Γ = a set of function specifications.

Function calls:

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

Rules for functions

A context Γ = 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:

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

Rules for functions

A context Γ = 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:

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

Rules for functions

A context Γ = 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:

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

Rules for functions

A context Γ = 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:

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

Rules for functions

A context Γ = 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:

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

Rules for functions

A context Γ = 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:

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

An example of verification

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

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

An example of verification

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

$\texttt{let}\ \mathit{slowset}\ (x, n) =$ $\{\, x \mapsto \_ \,\}$

$\quad \texttt{if}\ n = 0\ \texttt{then}$

$\quad\quad \texttt{set}(x, 0)$

$\quad \texttt{else}$

$\quad\quad \mathit{slowset}\ (x, n - 1);$

$\quad\quad \texttt{let}\ v = \texttt{get}(x)\ \texttt{in}\ \texttt{set}(v, x + 1)$

$\texttt{in}$

$\quad \mathit{slowset}(a, 2);$

$\quad \mathit{slowset}(b, 3)$

An example of verification

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

$\texttt{let}\ \mathit{slowset}\ (x, n) = \qquad \{ x \mapsto \_ \}$

$\quad \texttt{if}\ n = 0\ \texttt{then}$

$\qquad \texttt{set}(x, 0) \qquad \{ x \mapsto 0 \}$

$\quad \texttt{else}$

$\qquad \mathit{slowset}\ (x, n - 1);$

$\qquad \texttt{let}\ v = \texttt{get}(x)\ \texttt{in}\ \texttt{set}(v, x + 1)$

$\texttt{in}$

$\quad \mathit{slowset}(a, 2);$

$\quad \mathit{slowset}(b, 3)$

An example of verification

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

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

$$
\begin{array}{l}
slowset(a, 2); \\
slowset(b, 3)
\end{array}
$$

An example of verification

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

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

$$
\begin{array}{l}
\quad \mathit{slowset}(a, 2); \\
\quad \mathit{slowset}(b, 3)
\end{array}
$$

An example of verification

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

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

An example of verification

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

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

An example of verification

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

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

Functions as first-class values in separation logic

PTR with first-class functions

Expressions: $a ::= \ldots$

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

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

$\mid a_1\, a_2$ 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.

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 β-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:

$$P ::= \langle A \rangle \mid \texttt{emp} \mid \ell \mapsto v \mid P_1 \ast P_2 \mid \dots$$

$$\mid \{ P \} \; c \; \{ Q \}$$ Hoare triple

Postconditions:

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

Hoare triples as assertions

Assertions, preconditions:

$$P ::= \langle A \rangle \mid \texttt{emp} \mid \ell \mapsto v \mid P_1 \ast P_2 \mid \ldots$$
$$\mid \{\, P \,\}\; c \;\{\, Q \,\} \qquad \text{Hoare triple}$$

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:

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

Rules for functions

Recursive abstraction:

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

Rules for functions

Recursive abstraction:

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

Rules for functions

Recursive abstraction:

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

Rules for functions

Recursive abstraction:

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

Rules for functions

Recursive abstraction:

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

Rules for functions

Recursive abstraction:

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

Nonrecursive abstraction (derived rule):

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

Rules for functions

Recursive abstraction:

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

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:

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

Rules for functions

Recursive abstraction:

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

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:

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

(The rule is drawn with a double line.)

Rules for functions

Recursive abstraction:

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

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:

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

(double inference line)

Rules for functions

Recursive abstraction:

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

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:

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

Rules for functions

Recursive abstraction:

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

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:

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

(double inference line: the rule holds in both directions)

Specifying a higher-order function

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

Specifying a higher-order function

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Representing an object with an internal state

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

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:

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

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:

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

Representation predicate

We define the predicate $Counter(p, n)$, “at location p there is a counter whose current value is n”, as follows:

Representation predicate

We define the predicate *Counter*(p, n), "at location p there is a counter whose current value is n", as follows:

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

Representation predicate

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

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

Representation predicate

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

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

Representation predicate

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

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

Representation predicate

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

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

Representation predicate

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

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

Representation predicate

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

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

Representation predicate

We define the predicate $Counter(p, n)$, “at location p there is a counter whose current value is n”, as follows:

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

We can then prove

$$\{ \mathtt{emp} \} \quad counter\ () \quad \{ \lambda p.\ Counter(p, 0) \}$$

Representation predicate

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

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

We can then prove

$$\begin{array}{rcl} \{\, \texttt{emp} \,\} & \mathit{counter}\ () & \{\, \lambda p.\ \mathit{Counter}(p, 0) \,\} \\ \{\, \mathit{Counter}(p, n) \,\} & \texttt{get}(p)\ () & \{\, \lambda v. \qquad\qquad\qquad\qquad \,\} \end{array}$$

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

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

Representation predicate

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

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

We can then prove

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

Representation predicate

We define the predicate *Counter*(p, n), "at location p there is a counter whose current value is n", as follows:

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

We can then prove

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

Semantic soundness of the rule for recursion

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

Semantic soundness of the rule for recursion

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

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

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

Semantic soundness of the rule for recursion

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

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

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

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

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

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

Step-indexing to the rescue

Idea: in the definition of the semantic Hoare triple

$$\{\{\, P \,\}\}\ c\ \{\{\, Q \,\}\} = \forall n,\ h,\ P\ h \Rightarrow \texttt{Term}\ c\ h\ Q$$

Step-indexing to the rescue

Idea: in the definition of the semantic Hoare triple

$$\{\{\, P \,\}\}\; c\; \{\{\, Q \,\}\} = \forall n,\ h,\ P\, h \Rightarrow \ \mathtt{Term}\; c\; h\; Q$$

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

Step-indexing to the rescue

Idea: in the definition of the semantic Hoare triple

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

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

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:

$$\langle A \rangle\ h\ n =$$

$$(\ell \mapsto v)\ h\ n =$$

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:

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

$$(\ell \mapsto v)\ h\ n =$$

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:

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

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

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

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:

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

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

but it is taken into account for "triple" assertions

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

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:

$$\langle A \rangle\ h\ n = \mathit{Dom}(h) = \emptyset \wedge A$$

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

but it is taken into account for "triple" assertions

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

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

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:

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

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

but it is taken into account for "triple" assertions

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

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

The semantic triple, then, becomes

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

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: $[\![t]\!]\ Q = wp(t, Q)$.

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

Characteristic formulas for pure programs

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

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

Some representative cases:

$$\begin{aligned}
[\![v]\!] &= \\
[\![\texttt{fail}]\!] &= \\
[\![\texttt{let } x = t \texttt{ in } t']\!] &= \\
[\![\texttt{if } v \texttt{ then } t_1 \texttt{ else } t_2]\!] &=
\end{aligned}$$

Characteristic formulas for pure programs

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

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

Some representative cases:

$$\begin{aligned} [\![v]\!] &= \lambda Q.\ Q\ \lceil v\rceil \\ [\![\texttt{fail}]\!] &= \\ [\![\texttt{let}\ x = t\ \texttt{in}\ t']\!] &= \\ [\![\texttt{if}\ v\ \texttt{then}\ t_1\ \texttt{else}\ t_2]\!] &= \end{aligned}$$

Characteristic formulas for pure programs

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

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

Some representative cases:

$$
\begin{aligned}
[\![v]\!] &= \lambda Q.\ Q\ \lceil v \rceil \\
[\![\texttt{fail}]\!] &= \lambda Q.\ \bot \\
[\![\texttt{let}\ x = t\ \texttt{in}\ t']\!] &= \\
[\![\texttt{if}\ v\ \texttt{then}\ t_1\ \texttt{else}\ t_2]\!] &=
\end{aligned}
$$

Characteristic formulas for pure programs

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

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

Some representative cases:

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

Characteristic formulas for pure programs

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

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

Some representative cases:

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

Characteristic formulas for pure programs

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

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

Some representative cases:

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

Characteristic formulas for pure programs

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

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

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

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

Some representative cases:

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

Characteristic formulas for pure programs

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

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

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

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

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:

Characteristic formulas for pure programs

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

$$[\![v]\!] = \qquad [\![f\ v]\!] = \qquad [\![\texttt{fail}]\!] =$$

$$[\![\texttt{let}\ x = t\ \texttt{in}\ t']\!] =$$

$$[\![\texttt{if}\ v\ \texttt{then}\ t_1\ \texttt{else}\ t_2]\!] =$$

Characteristic formulas for pure programs

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

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

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

$$[\![\mathtt{if}\ v\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2]\!] =$$

Characteristic formulas for pure programs

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

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

$$[\![\texttt{let}\ x = t\ \texttt{in}\ t']\!] =$$

$$[\![\texttt{if}\ v\ \texttt{then}\ t_1\ \texttt{else}\ t_2]\!] =$$

Characteristic formulas for pure programs

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

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

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

$$[\![\mathtt{if}\ v\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2]\!] =$$

Characteristic formulas for pure programs

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

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

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

$$[\![\mathtt{if}\ v\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2]\!] =$$

Characteristic formulas for pure programs

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

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

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

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

Characteristic formulas for pure programs

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

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

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

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

where the combinators are defined as

$$\texttt{Ret}\ V = \qquad \texttt{App}\ F\ V = \qquad \texttt{Fail} =$$

$$\texttt{Let}\ x = F\ \texttt{In}\ F' =$$

$$\texttt{If}\ V\ \texttt{Then}\ F\ \texttt{Else}\ F' =$$

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

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

$$\mathtt{Let}\ x = F\ \mathtt{In}\ F' =$$

$$\mathtt{If}\ V\ \mathtt{Then}\ F\ \mathtt{Else}\ F' =$$

Characteristic formulas for pure programs

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

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

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

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

where the combinators are defined as

$$\texttt{Ret}\ V = \lambda Q.\ Q\ V \qquad \texttt{App}\ F\ V = \qquad \texttt{Fail} = \lambda Q.\ \bot$$

$$\texttt{Let}\ x = F\ \texttt{In}\ F' =$$

$$\texttt{If}\ V\ \texttt{Then}\ F\ \texttt{Else}\ F' =$$

Characteristic formulas for pure programs

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

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

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

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

where the combinators are defined as

$$\texttt{Ret}\ V = \lambda Q.\ Q\ V \qquad \texttt{App}\ F\ V = \qquad \texttt{Fail} = \lambda Q.\ \bot$$

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

$$\texttt{If}\ V\ \texttt{Then}\ F\ \texttt{Else}\ F' =$$

Characteristic formulas for pure programs

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

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

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

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

where the combinators are defined as

$$\texttt{Ret}\ V = \lambda Q.\ Q\ V \qquad \texttt{App}\ F\ V = \qquad \texttt{Fail} = \lambda Q.\ \bot$$

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

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

Characteristic formulas for pure programs

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

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

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

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

where the combinators are defined as

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

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

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

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
```

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

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

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

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

that is,

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

Representing functions

A function is represented by a value of the abstract type *Func*. The *AppReturns* operator associates a characteristic formula to each function:

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

Representing functions

A function is represented by a value of the abstract type *Func*. The *AppReturns* operator associates a characteristic formula to each function:

$$AppReturns : \forall A, B,\ Func \rightarrow A \rightarrow (B \rightarrow \mathtt{Prop}) \rightarrow \mathtt{Prop}$$

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

Representing functions

A function is represented by a value of the abstract type *Func*. The *AppReturns* operator associates a characteristic formula to each function:

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

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 *AppReturns f*:

$$\forall x, P\ x \Rightarrow AppReturns\ f\ x\ Q$$

Specifying functions

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

Specifying functions

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

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

Specifying functions

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

$$\forall x, P\ x \Rightarrow AppReturns\ f\ x\ Q$$

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:

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

Specifying higher-order functions

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

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

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".

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”.

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

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 *heap* $\rightarrow$ `Prop` instead of propositions `Prop`.

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

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

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:

$\forall x : A.\ P(x) \to B$	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)

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

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 $\{\, P \,\}\ c\ \{\, Q \,\}$.

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

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

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

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

Hoare monads

The interesting aspect of these monadic operations is their types:

`ret` :

`bind` :

Hoare monads

The interesting aspect of these monadic operations is their types:

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

$\texttt{bind}$:

Hoare monads

The interesting aspect of these monadic operations is their types:

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

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

Hoare monads

The interesting aspect of these monadic operations is their types:

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

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

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

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

“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" 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:

$$\begin{aligned}
\texttt{get}\ \ell &: \\
\texttt{set}\ \ell\ \mathbf{v} &: \\
\texttt{alloc} &: \\
\texttt{free}\ \ell &:
\end{aligned}$$

Imperative operations

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

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

Imperative operations

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

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

Imperative operations

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

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

Imperative operations

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

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

A separation monad

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

$$STsep\ P\ A\ Q = \forall R,\ ST\ (P \ast R)\ A\ (\lambda v.\ Q\ v \ast R)$$

A separation monad

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

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

The frame rule corresponds to a retyping function:

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

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:

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

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

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:

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

with $Q : A \to State \to State \to \mathtt{Prop}$.

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

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

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.

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.

$$
\begin{aligned}
PRE &= \\
POST \; A &= \\
TRANSF \; A &= \\
ST \; A \; (W : TRANSF \; A) &=
\end{aligned}
$$

Dijkstra monads

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

Example: the state monad.

$$\begin{aligned} \mathit{PRE} &= \mathit{State} \to \mathtt{Prop} \\ \mathit{POST}\ A &= \\ \mathit{TRANSF}\ A &= \\ \mathit{ST}\ A\ (W : \mathit{TRANSF}\ A) &= \end{aligned}$$

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.

$$\begin{aligned} PRE &= State \to \texttt{Prop} \\ POST\ A &= A \to State \to \texttt{Prop} \\ TRANSF\ A &= \\ ST\ A\ (W : TRANSF\ A) &= \end{aligned}$$

Dijkstra monads

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

Example: the state monad.

$$\begin{aligned} \mathit{PRE} &= \mathit{State} \to \texttt{Prop} \\ \mathit{POST}\;A &= A \to \mathit{State} \to \texttt{Prop} \\ \mathit{TRANSF}\;A &= \mathit{POST}\;A \to \mathit{PRE} \\ \mathit{ST}\;A\;(W : \mathit{TRANSF}\;A) &= \end{aligned}$$

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.

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

Dijkstra monads

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

Example: the state monad.

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

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

The operations of the Dijkstra state monad

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

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

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 $W_2 : A \rightarrow TRANSF\ B$ and $m : ST\ A\ W_1$ and $f : \forall a : A, ST\ B\ (W_2\ a)$,

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 $f : \forall a : A, ST\ B\ (W_2\ a)$,

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

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

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:

$$\mathit{GET}\ \ell : \mathit{TRANSF}\ Z = \lambda Q, s,\ \ell \in \mathit{Dom}(s) \wedge Q\ (s\ \ell)\ s$$

$$\texttt{get}\ \ell : \mathit{ST}\ Z\ (\mathit{GET}\ \ell)$$

The operations of the Dijkstra state monad

Operations on memory follow the same pattern:

$$GET\ \ell : TRANSF\ Z = \lambda Q, s,\ \ell \in Dom(s) \wedge Q\ (s\ \ell)\ s$$

$$\texttt{get}\ \ell : ST\ Z\ (GET\ \ell)$$

$$SET\ \ell\ v : TRANSF\ \texttt{unit} = \lambda Q, s,\ \ell \in Dom(s) \wedge Q\ ()\ s[\ell \leftarrow v]$$

$$\texttt{set}\ \ell\ v : ST\ \texttt{unit}\ (SET\ \ell)$$

The operations of the Dijkstra state monad

Operations on memory follow the same pattern:

$$GET\ \ell : TRANSF\ Z = \lambda Q, s,\ \ell \in Dom(s) \wedge Q\ (s\ \ell)\ s$$

$$\texttt{get}\ \ell : ST\ Z\ (GET\ \ell)$$

$$SET\ \ell\ v : TRANSF\ \texttt{unit} = \lambda Q, s,\ \ell \in Dom(s) \wedge Q\ ()\ s[\ell \leftarrow v]$$

$$\texttt{set}\ \ell\ v : ST\ \texttt{unit}\ (SET\ \ell)$$

$$ALLOC : TRANSF\ \texttt{addr} = \lambda Q, s,\ \forall \ell \notin Dom(s), Q\ \ell\ s[\ell \leftarrow 0]$$

$$\texttt{alloc} : ST\ \texttt{addr}\ ALLOC$$

The operations of the Dijkstra state monad

Operations on memory follow the same pattern:

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

The operations of the Dijkstra state monad

Operations on memory follow the same pattern:

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

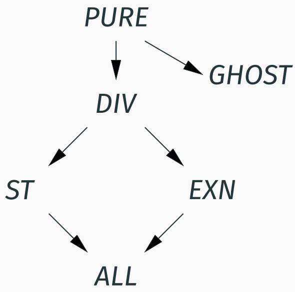
Remark: we can define `get`, `set`, ..., in accordance with our definition of *ST*; but we can also leave these operations abstract, which leads to an axiomatization of a built-in “mutable state” effect.

The Dijkstra monad for exceptions

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

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

A hierarchy of monads



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.

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

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.