Verification of Imperative Programs in Hoare Type Theory

Ilya Sergey

IMDEA Software Institute

ilyasergey.net/pnp-2014

Declarative *vs* Imperative

Declarative Programming

- The program already "specifies" its result;
- Logical/Constraint programming: the program is a number of logical clauses/constraints that specify the requirements for the result.
- (Pure) functional programming: the program is an expression whose value, upon evaluation is the result of the program (if it doesn't diverge).
  - The program can be replaced by its result (*referential transparency*).

Imperative Programming

- The program describes a *sequence of steps* that should be performed in order to obtain the result;
- The result of the program is its *side effect*:
  - An output to the screen;
  - a state of the memory;
  - an exception;
- The lack of referential transparency due to side effects.

How to give a declarative specification to imperative programs?


Use the types, Luke!

A long time ago…

Floyd-Hoare program logic

[Floyd:SAM'67, Hoare:CACM'69]

- Independently discovered by Robert W. Floyd (1967) and Tony Hoare (1969);
- Sometimes referred to as "axiomatic program semantics";
- Specifies a program by means of pre-/postconditions;
- Provides an inference system to infer proofs of specifications of larger programs from specifications of smaller ones.

Hoare triples

{ P } c { Q }

precondition postcondition

Meaning:

If *right before* the program **c** is executed the state of mutable variables is described by the proposition P, then, *if* **c** *terminates*, the resulting state satisfies the proposition Q.

Example specification

$\{\, \mathbf{True} \,\}$ `x := 3` $\{\, x = 3 \,\}$

Hoare logic language

- The state is represented by a (supposedly infinite) set of mutable variables, which can be assigned arbitrary values;
- All variables have distinct names;
- No procedures, no heap/pointers;
- Simple conditional commands (**if-then-else**) and **while**-loops.

Hoare logic rules

Assignment

$$\{\, Q[e/x] \,\} \;\; x := e \;\; \{\, Q \,\} \qquad \text{(Assign)}$$

substitute **x** with **e**

$$\{\, 3 = 3 \,\} \;\; \texttt{x := 3} \;\; \{\, x = 3 \,\}$$

Sequential composition

$$\frac{\{\,P\,\}\ c_1\ \{\,Q\,\} \qquad \{\,Q\,\}\ c_2\ \{\,R\,\}}{\{\,P\,\}\ c_1 ;\ c_2\ \{\,R\,\}}\ \text{(Seq)}$$

$\{???\}$ x := 3; y := x $\{x = 3 \wedge y = 3\}$

Sequential composition

$$\frac{\{\,P\,\}\ c_1\ \{\,Q\,\} \qquad \{\,Q\,\}\ c_2\ \{\,R\,\}}{\{\,P\,\}\ c_1\,;\ c_2\ \{\,R\,\}}\ \text{(Seq)}$$

Yikes!

$\boxed{\{3 = 3 \land 3 = 3\}}$

$x := 3;$

$\{x = 3 \land x = 3\}$ (Assign)

$y := x$

$\{x = 3 \land y = 3\}$ (Assign)

Rule of consequence

$$\frac{P \Rightarrow P' \qquad \{\,P'\,\}\ c\ \{\,Q'\,\} \qquad Q' \Rightarrow Q}{\{\,P\,\}\ c\ \{\,Q\,\}}\ \text{(Conseq)}$$

$$\{\ \mathbf{True}\ \} \Rightarrow \{3 = 3 \land 3 = 3\}$$

$$x := 3;\ y := x$$

$$\{x = 3 \land y = 3\}$$

Rule of consequence

$$\frac{P \Rightarrow P' \qquad \{\, P' \,\}\ c\ \{\, Q' \,\} \qquad Q' \Rightarrow Q}{\{\, P \,\}\ c\ \{\, Q \,\}}\ (\text{Conseq})$$

$\{\, \textbf{True} \,\}\ \ x := 3; y := x\ \ \{x = 3 \land y = 3\}$

Function subtyping rule

more “precise” type

$$\frac{P <: P' \qquad \boxed{c : P' \to Q'} \qquad Q' <: Q}{\boxed{c : P \to Q}}$$

less “precise” type

Function subtyping rule

$$\frac{P <: P' \qquad Q' <: Q}{P' \rightarrow Q' <: P \rightarrow Q}$$

Logical variables

$\forall$ a, b,

$$\Big(\{x = a \wedge y = b\}\ \ t := x;\ x := y;\ y := t\ \ \{x = b \wedge y = a\}\Big)$$

Conditionals and loops

$$\frac{\{\, P \wedge e \,\}\ c_1\ \{\, Q \,\} \qquad \{\, P \wedge \neg e \,\}\ c_2\ \{\, Q \,\}}{\{\, P \,\}\ \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{\, Q \,\}}\ \text{(Cond)}$$

$$\frac{\{\, I \wedge e \,\}\ c\ \{\, I \,\}}{\{\, I \,\}\ \textbf{while}\ e\ \textbf{do}\ c\ \{\, I \wedge \neg e \,\}}\ \text{(While)}$$

loop invariant (needs to be guessed)

Why Hoare logic doesn't scale

- The language with mutable variables is <u>too simplistic</u>;
- The lack of procedures means the <u>absence of modularity</u>;
- But the main problem is adding <u>*pointers*</u>.

A language with pointers

- *Heap* is a finite partial map from **nat** to arbitrary values;
- *Pointers* are natural numbers from the heap domain;
- In the presence of pointers, we assume all variables to be *immutable*.

assign value 3 to a pointer x

$\{ x \mapsto - \wedge y \mapsto b \}$ `x ::= 3` $\{ x \mapsto 3 \wedge y \mapsto b \}$

This spec is *wrong*!

if x and y are *aliases*, the value of y was affected

$\{ x \mapsto \text{-} \wedge y \mapsto b \}$

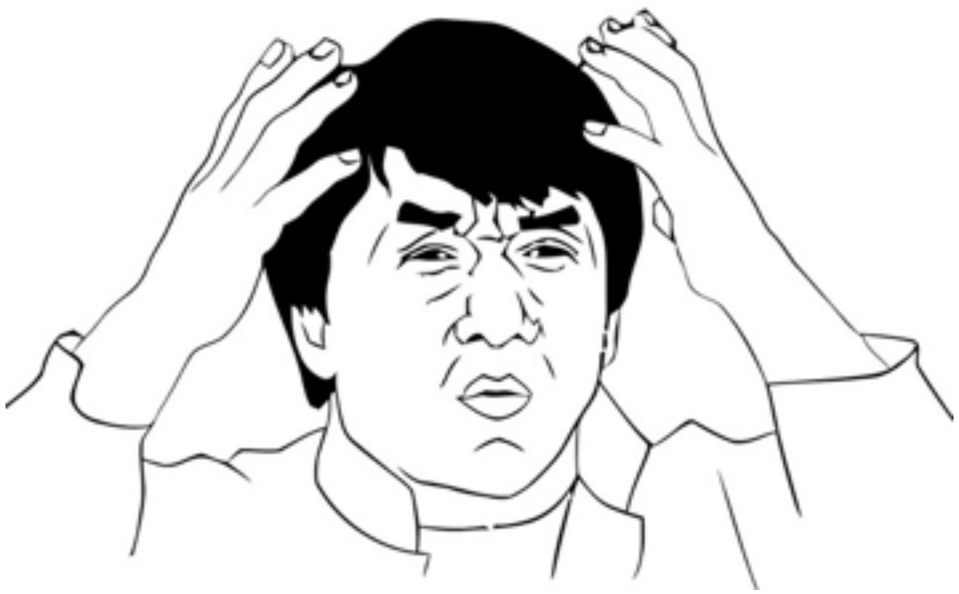
```
x ::= 3
```

$\{ x \mapsto 3 \wedge$

$(x \neq y \wedge y \mapsto b) \vee (x = y \wedge y \mapsto 3)\}$

What about 3 variables?

…or an array?



Separation Logic

[Reynolds:LICS02]

- Co-invented in 2002 by John C. Reynolds, Peter O'Hearn, Samin Ishtiaq and Hongseok Yang;
- The key idea is to make *heap disjointness* *explicit;*
- Aliasing is no longer a problem.

$$\{ x \mapsto - \wedge y \mapsto b \} \; \texttt{x ::= 3} \; \{ x \mapsto 3 \wedge y \mapsto b \}$$

Separation Logic

[Reynolds:LICS02]

- Co-invented in 2002 by John C. Reynolds, Peter O'Hearn, Samin Ishtiaq and Hongseok Yang;
- The key idea is to make *heap disjointness* *__explicit__*;
- Aliasing is no longer a problem.

$\{h \mid h = x \mapsto - \bullet y \mapsto b\}$ `x ::= 3` $\{h \mid h = x \mapsto 3 \bullet y \mapsto b\}$

disjoint union of heaps

Revising the language and logic

- Variables are now <u>immutable</u>, single-assigned—changes in the *state* are changes in the *heap*;
- All commands <u>return results</u>, which are pure expressions;
- Non-result returning operations return an element of <u>*unit*</u>;
- <u>Allocation/deallocation</u> are provided as primitives with appropriate logical rules specifying them;
- **while**-loops are expressed using <u>recursive functions</u>.

Writing to a pointer

$$\{h \mid \boxed{h = x \mapsto -}\}\ \ x ::= e\ \ \{res, h \mid \boxed{h = x \mapsto e} \wedge res = \mathbf{tt}\}$$

(Write)

Reading from a pointer

$$\{h \mid \boxed{h = x \mapsto v}\}\ \ !x\ \ \{res, h \mid h = x \mapsto v \wedge \boxed{res = v}\}$$

(Read)

Frame rule

“small footprint”

$$\frac{\{h \mid P(h)\}\ \ c\ \ \{res, h \mid Q(res, h)\}}{\{h \mid \exists h_l,\ h = h_l \bullet \boxed{h'} \wedge\ P(h_l)\}\ \ c\ \ \{res, h \mid \exists h_l,\ h = h_l \bullet \boxed{h'} \wedge\ Q(res, h_l)\}}\ \text{(Frame)}$$

“large footprint”

“frame” (universally quantified)

Anti-frame rule

h' is taken to be **empty**

$$\frac{\{h \mid \exists h_l,\ h = h_l \bullet \boxed{h'} \wedge\ P(h_l)\}\ \ c\ \ \{res, h \mid \exists h_l,\ h = h_l \bullet \boxed{h'} \wedge\ Q(res, h_l)\}}{\{h \mid P(h)\}\ \ c\ \ \{res, h \mid Q(res, h)\}}\ \text{(Anti-Frame)}$$

Allocation/deallocation

$\{h \mid h = \mathbf{emp}\}$ **alloc**(e) $\{res, h \mid h = res \mapsto e\}$

(Alloc)

$\{h \mid h = x \mapsto \textbf{-}\}$ **dealloc**(x) $\{res, h \mid h = \mathbf{emp} \wedge res = \mathbf{tt}\}$

(Dealloc)

Binding

$$\frac{\{h \mid P(h)\}\ c_1\ \{res, h \mid \boxed{Q(res, h)}\} \qquad \{h \mid \boxed{Q(x, h)}\}\ c_2\ \{res, h \mid R(res, h)\}}{\{h \mid P(h)\}\ \ x \texttt{<-}\ c_1;\ \ c_2\ \ \{res, h \mid R(res, h)\}}\ \text{(Bind)}$$

The result of $\mathbf{c_1}$ is *bound* within $\mathbf{c_2}$ under a name $\mathbf{x}$.

“Oblivious” Binding

$$\frac{\{h \mid P(h)\}\ c_1\ \{res, h \mid Q(h)\} \qquad \{h \mid Q(h)\}\ c_2\ \{res, h \mid R(res, h)\}}{\{h \mid P(h)\}\ \ c_1\ ;;\ \ c_2\ \ \{res, h \mid R(res, h)\}}\ \text{(BindO)}$$

The result of c_1 is *irrelevant* for c_2.

The rule of conjunction

$$\frac{\{h \mid P_1(h)\}\ c\ \{res, h \mid Q_1(res, h)\} \qquad \{h \mid P_2(h)\}\ c\ \{res, h \mid Q_2(res, h)\}}{\{h \mid P_1(h) \wedge P_2(h)\}\ c\ \{res, h \mid Q_1(res, h) \wedge Q_2(res, h)\}}\ (\text{Conj})$$

Working with functions

$$\{h \mid P(h)\}\ \ \mathbf{ret}\ e\ \ \{res, h \mid P(h) \wedge res = e\} \qquad \text{(Return)}$$

$$\frac{\forall x,\ \{h \mid P(x, h)\}\ f\ (x)\ \{res, h \mid Q(x, res, h)\} \in \Gamma}{\Gamma \vdash \forall x,\ \{h \mid P(x, h)\}\ f\ (x)\ \{res, h \mid Q(x, res, h)\}} \ \text{(Hyp)}$$

a *context* of *assumed* functions' specs

$$\frac{\Gamma \vdash \forall x,\ \{h \mid P(x, h)\}\ f\ (x)\ \{res, h \mid Q(x, res, h)\}}{\Gamma \vdash \{h \mid P(e, h)\}\ f(e)\ \{res, h \mid Q(e, res, h)\}} \ \text{(App)}$$

Representing loops using recursive functions

Fixed point combinator

$$\mathbf{fix} \;:\; (T \to T) \to T$$

$$\mathbf{fix}\ f \;=\; f\ (\mathbf{fix}\ f)$$

- A way to implement <u>general recursion</u> in pure calculi;
- <u>Cannot</u> be encoded within a primitively-recursive language;
- Its argument f should be <u>*continuous*</u> (in Scott's topology).

Factorial implementation using fix

$$T = \mathtt{nat} \to \mathtt{nat}$$

```
fact = fix (fun (f : nat → nat) =>
              (fun (n : nat) =>
                   if n = 0 then 1
                   else n * f (n - 1)))
```

$$T \to T = (\mathtt{nat} \to \mathtt{nat}) \to (\mathtt{nat} \to \mathtt{nat})$$

$$\mathbf{fix} : (T \to T) \to T$$

$$\mathbf{fix}\ f = f\ (\mathbf{fix}\ f)$$

Refactoring loops to functions

while e **do** c

Refactoring loops to functions

In-loop condition

(**fix** f (x: bool). **if** x
then c ;; f(e')
else **ret** tt) (e)

Initial condition

A rule for recursive functions

Assuming a specification for f… one has to verify its body…

$$\frac{\Gamma;\ \forall x,\ \{h \mid P(x,h)\}\ f(x)\ \{res, h \mid Q(x, res, h)\} \quad \vdash\ \{h \mid P(x,h)\}\ c\ \{res, h \mid Q(x, res, h)\}}{\Gamma \vdash \forall y\ \{h \mid P(y,h)\}\ (\textbf{fix}\ f(x).c)(y)\ \{res, h \mid Q(y, res, h)\}}\ (\text{Fix})$$

which justifies the inference of the spec for **fix**.

Remark: P and Q here play the role of the *loop invariant.*

Verifying imperative programs in Separation Logic

A factorial implementation

```
fun fact (N : nat): nat = {
  n   <-- alloc(N);
  acc <-- alloc(1);
  res <--
    (fix loop (_ : unit).
      a' <-- !acc;
      n' <-- !n;
      if n' == 0 then ret a'
      else acc ::= a' * n';;
           n   ::= n' - 1;;
           loop(tt)
    )(tt);
  dealloc(n);;
  dealloc(acc);;
  ret res
}
```

A factorial implementation

```
fun fact (N : nat): nat = {
  n   <-- alloc(N);
  acc <-- alloc(1);
  res <--
    (fix loop (_ : unit).
      a' <-- !acc;
      n' <-- !n;
      if n' == 0 then ret a'
      else acc ::= a' * n';;
           n   ::= n' - 1;;
           loop(tt)
    )(tt);
  dealloc(n);;
  dealloc(acc);;
  ret res
}
```

$$\mathsf{f}(N) \overset{\text{def}}{=} \textbf{if } N = N' + 1 \textbf{ then } N \times \mathsf{f}(N') \textbf{ else } 1$$

$\{h \mid h = \textbf{emp}\}$

`fact`(N)

$\{res, h \mid h = \textbf{emp} \wedge res = \mathsf{f}(N)\}$

Compositional verification

```
fun fact (N : nat): nat = {
  n   <-- alloc(N);
  acc <-- alloc(1);
  res <--
    (fix loop (_ : unit).
      a' <-- !acc;
      n' <-- !n;
      if n' == 0 then ret a'
      else acc ::= a' * n';;
           n   ::= n' - 1;;
           loop(tt)
    )(tt);
  dealloc(n);;
  dealloc(acc);;
  ret res
}
```

$\stackrel{\text{def}}{=}$ fact_loop(tt)

Compositional verification

```
fun fact (N : nat): nat = {
  n   <-- alloc(N);
  acc <-- alloc(1);
  res <-- fact_loop(tt);
  dealloc(n);;
  dealloc(acc);;
  ret res
}
```

```
fun fact_loop (_: unit): nat =
 (fix loop (_ : unit).
     a' <-- !acc;
     n' <-- !n;
     if n' == 0 then ret a'
     else acc ::= a' * n';;
          n   ::= n' - 1;;
          loop(tt))
```

Compositional verification

$$F_{inv}(\mathrm{n}, \mathrm{acc}, N, \mathrm{h}) \stackrel{\mathrm{def}}{=} \exists n', a', (\ \mathrm{h} = \mathrm{n} \mapsto n' \bullet \mathrm{acc} \mapsto a'\ ) \land (\ \mathrm{f}(n') \times a' = \mathrm{f}(N)\ )$$

```
fun fact_loop (_: unit): nat =
 (fix loop (_ : unit).
      a' <-- !acc;
      n' <-- !n;
      if n' == 0 then ret a'
      else acc ::= a' * n';;
           n   ::= n' - 1;;
           loop(tt))
```

$\{\mathrm{h} \mid F_{inv}(\mathrm{n}, \mathrm{acc}, N, \mathrm{h})\}$

`fact_loop(tt)`

$\{\mathrm{res}, \mathrm{h} \mid F_{inv}(\mathrm{n}, \mathrm{acc}, N, \mathrm{h}) \land \mathrm{res} = \mathrm{f}(N)\}$

$\boxed{\{h \mid F_{inv}(n, acc, N, h)\}}$

```
fun fact_loop (_: unit): nat =
```

$\{h \mid F_{inv}(n, acc, N, h)\}$ (precondition)

```
 (fix loop (_ : unit).
```

$\{h \mid \exists\, n'\, a',\ (h = n \mapsto n' \bullet acc \mapsto a') \wedge (f(n') \times a' = f(N))\}$ (F_{inv} definition)

```
        a' <-- !acc;
```

$\{h \mid \exists\, n',\ (h = n \mapsto n' \bullet acc \mapsto a') \wedge (f(n') \times a' = f(N))\}$ (Read), (Conj)

```
        n' <-- !n;
```

$\{h \mid (h = n \mapsto n' \bullet acc \mapsto a') \wedge (f(n') \times a' = f(N))\}$ (Read), (Conj)

```
        if n' == 0 then ret a'
```

$\{res, h \mid (h = n \mapsto 0 \bullet acc \mapsto f(N)) \wedge (res = f(N))\}$ (Cond), (Return)

$\boxed{\{res, h \mid F_{inv}(n, acc, N, h) \wedge (res = f(N))\}}$ (F_{inv} definition)

```
        else acc ::= a' * n';;
```

$\{h \mid (h = n \mapsto n' \bullet acc \mapsto a' \times n') \wedge (f(n') \times a' = f(N))\}$ (Cond), (Write)

```
            n   ::= n' - 1;;
```

$\{h \mid (h = n \mapsto n'\text{-}1 \bullet acc \mapsto a' \times n') \wedge (f(n') \times a' = f(N))\}$ (Write)

$\{h \mid (h = n \mapsto n'\text{-}1 \bullet acc \mapsto a' \times n') \wedge (f(n' - 1) \times a' \times n' = f(N))\}$ (mathematics)

$\{h \mid F_{inv}(n, acc, N, h)\}$ (F_{inv} definition)

```
             loop(tt))
```

$\boxed{\{res, h \mid F_{inv}(n, acc, N, h) \wedge (res = f(N))\}}$ (Fix), (Hyp), (App)

$$\{h \mid F_{inv}(n, acc, N, h)\}$$

```
fact_loop(tt)
```

$$\{res, h \mid F_{inv}(n, acc, N, h) \wedge (res = f(N))\}$$

$\{h \mid F_{inv}(n, acc, N, h)\}$

```
fact_loop(tt)
```

$\{res, h \mid F_{inv}(n, acc, N, h) \wedge (res = f(N))\}$

$\boxed{\{h \mid h = emp\}}$

```
fun fact (N : nat): nat = {
```

$\{h \mid h = emp\}$ (precondition)

```
  n   <-- alloc(N);
```

$\{h \mid h = n \mapsto N\}$ (Alloc)

```
  acc <-- alloc(1);
```

$\{h \mid h = n \mapsto N \bullet acc \mapsto 1\}$ (Alloc)

$\{h \mid F_{inv}(n, acc, N, h)\}$ (F_{inv} definition)

```
  res <-- fact_loop(tt);
```

$\{h \mid F_{inv}(n, acc, N, h) \wedge (res = f(N))\}$ (`fact_loop` spec), (Hyp), (App)

$\{h \mid (h = n \mapsto - \bullet acc \mapsto -) \wedge (res = f(N))\}$ (F_{inv} definition)

```
  dealloc(n);;
```

$\{h \mid (h = acc \mapsto -) \wedge (res = f(N))\}$ (Dealloc)

```
  dealloc(acc);;
```

$\{h \mid (h = emp) \wedge (res = f(N))\}$ (Dealloc)

```
  ret res }
```

$\boxed{\{h \mid (h = emp) \wedge (res = f(N))\}}$ (Ret)

$$\{\mathrm{h} \mid h = \mathbf{emp}\}$$

$$\texttt{fact}(N)$$

$$\{\mathrm{h} \mid (h = \mathbf{emp}) \wedge (\mathrm{res} = \mathrm{f}(N))\}$$

Lessons learned

- Hoare/separation logic inference rules are reminiscent to datatype constructors;
- Large proofs can be decomposed into small ones;
- Hoare triples make reasoning modular;
  - also, they are similar to types in many ways;
- Paper-and-pencil reasoning is error-prone;
- We should be able to use Coq's dependent types to mechanize reasoning in Hoare/separation logic.

Missing ingredient:

monads

Expressions and Commands

- *Expressions:*
  ```
  tt
  (3 + 2)
  fact (42)
  ```
  pure: referentially-transparent, *always* evaluate to a result value

- *Commands* (aka *computations*, *programs*):
  ```
  ret 3
  x ::= 5
  c <- !x; y ::= x + 3;; t <- !y; ret t
  ```
  effectful: might diverge, write to store, throw exceptions

Eugenio Moggi

Notions of computation and monads, *Inf. Comput.*, 1991

... we identify the type A with the object of *values* (of type A) and obtain the object of *computations* (of type A) by applying an unary type-constructor T to A.
We call T a *notion of computation*, since it abstracts away from the type of pure values computations may produce.

Philip Wadler

Comprehending Monads, LFP, 1991

It is relatively straightforward to adopt Moggi's technique of structuring denotational specifications into a technique for structuring functional programs. This paper presents a simplified version of Moggi's ideas, framed in a way better suited to functional programmers than semanticists; in particular, no knowledge of category theory is assumed.

In functional programming, *monads* are datatypes that represent *effectful* computations:

state, I/O, exceptions, continuations, divergence, …

A *monad* datatype defines a strategy to *chain* (or *bind*) several operations together as well as to inject (or *return*) pure expressions into computations.

Monads in Haskell

type of computation

type of pure value

```
class Monad m where

    (>>=)       :: m a -> (a -> m b) -> m b

    return      :: a -> m a
```

Monadic do-notation

$$\underbrace{\overbrace{\texttt{c1}}^{\texttt{(m a)}}\ \ \texttt{>>=}\ \ \overbrace{\texttt{(\textbackslash x -> }\overbrace{\texttt{c2}}^{\texttt{(m b)}}\texttt{ >>= }\overbrace{\texttt{(\textbackslash y -> c3)}}^{\texttt{b -> (m c)}}\texttt{)}}^{\texttt{a -> (m c)}}}_{\texttt{(m c)}}$$

Monadic do-notation

```
do  x <- c1
    y <- c2
    c3
```

Monadic do-notation

```
                              :: IO ()
main = do putStrLn “Enter a character”
          c <- getChar     :: IO Char
          putStrLn “This was: ” ++ [c]        :: IO ()
          return ()     :: IO ()
```

Key Highlights

- Imperative programs perform *effectful* computations;
- Hoare triples provide a way to specify their effect;
- Computations are composed using *bind* and *return*;
- In functional programming, *effects* are specified by *monads*;
- Monads are composed using *bind* and *return*.

Hoare triples

+

Monadic types

Hoare Type Theory

Specifying and verifying effectful programs
in Coq with dependent types.

Hoare types

logical variables
appearing in p and q

$A \rightarrow$ heap $\rightarrow$ `Prop`

$$\texttt{c} \ : \ \boxed{\{\, x_1 \, x_2 \, \ldots \,\}} \, STsep \, (\, p \, , \, q \,)$$

heap $\rightarrow$ `Prop`

Example: allocator type

$\{h \mid h = \textbf{emp}\}$

$$
\begin{aligned}
&\text{alloc} : \forall\ (A : \texttt{Type})\ (v :\ A),\\
&\qquad\qquad STsep\ (\texttt{fun}\ h \Rightarrow h = \text{Unit},\\
&\qquad\qquad\qquad [\texttt{vfun}\ (res :\ \textbf{ptr})\ h \Rightarrow h = res\ \texttt{:->}\ v])
\end{aligned}
$$

$\{res, h \mid h = res \mapsto v\}$

Structuring the verification of imperative programs in Hoare Type Theory

[demo]

Deep *vs* shallow embedding

Deep embedding

- Sometimes is referred to as *"external DSLs"*;
- a new language is implemented *from scratch*;
  - parser, interpreter, name binding, type checking— all should be implemented;
- usually, easier to debug and profile;
- <u>one can implement a language with an arbitrary semantics.</u>
- <u>Examples</u>: MPS approach, any modern mainstream PL.

Shallow embedding

- Also known as *"internal/embedded DSLs"*;
- a new language reuses its *host language*'s infrastructure;
  - implementation amounts to defining the "de-sugaring" into the host language;
  - Any host program is also a DSL program;
- <u>the DSL's semantics is essentially it host's semantics.</u>
- <u>Examples</u>: Lisp DSLs, Scala parser combinators/actors, PLT Redex *etc*.

HTT is shallow embedding into Coq

- Hoare triples are instances of a particular datatypes;
- Higher-order specifications (e.g., *iterator*) for free;
- Coq takes care of the proof verification (i.e., *type checking*);
- Enables verification of real-life examples;
- Limitations of the framework are caused by Coq's model:
  - for instance, effectful functions cannot be stored into a heap (can be fixed by adding extra axioms).

Soundness of Hoare Type Theory

- Imperative programs are constructed as Coq expressions, but they *cannot* be run within Coq (because of general recursion);
- Extraction into a general-purpose language can be implemented;
- Soundness is established via denotational semantics:
  - Each imperative program is a *state transformer;*
  - Each pre/postcondition is a set of *state transformers*;
  - Soundness is established as an element/set inclusion.

More programming in HTT

[demo]