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

- 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

{ \text{P} } \ c \ { \text{Q} } \quad \text{precondition} \quad \text{postcondition}

Meaning:

If \textit{right before} the program \texttt{c} is executed the state of mutable variables is described by the proposition \text{P}, then, \textit{if} \texttt{c} \textit{terminates}, the resulting state satisfies the proposition \text{Q}.
Example specification

\{ \text{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

\[
\{ Q[e/x] \} \ x := e \ \{ Q \} \quad \text{(Assign)}
\]

substitute \( x \) with \( e \)

\[
\{ 3 = 3 \} \ x := 3 \ \{ x = 3 \}
\]
Sequential composition

\[
\{ P \} \ c_1 \ { Q \} \ { Q } \ c_2 \ { R } \\
\{ P \} \ c_1 ; \ c_2 \ { R } 
\]

\{????\} \ x := 3; \ y := x \ \{ x = 3 \land y = 3 \}
Sequential composition

\[
\begin{array}{c}
\{ P \} \ c_1 \ \{ Q \} \ \{ Q \} \ c_2 \ \{ R \} \\
\{ P \} \ c_1; \ c_2 \ \{ R \}
\end{array}
\]

(\text{Seq})

Yikes!

\[
\{ 3 = 3 \land 3 = 3 \}
\]

\[
x := 3;
\]

\[
\{ x = 3 \land x = 3 \} \quad \text{(Assign)}
\]

\[
y := x
\]

\[
\{ x = 3 \land y = 3 \} \quad \text{(Assign)}
\]
Rule of consequence

\[
P \Rightarrow P' \quad \{ P' \} \quad c \quad \{ Q' \} \quad Q' \Rightarrow Q
\]

(Conseq)

\[
\{ \text{True} \} \Rightarrow \{ 3 = 3 \land 3 = 3 \}
\]

\[
x := 3; y := x
\]

\[
\{ x = 3 \land y = 3 \}
\]
Rule of consequence

\[ P \Rightarrow P' \quad \{ P' \} \quad \text{c} \quad \{ Q' \} \quad Q' \Rightarrow Q \]

\[ \{ P \} \quad \text{c} \quad \{ Q \} \quad \text{(Conseq)} \]

\{ True \} \quad x := 3; y := x \quad \{ x = 3 \land y = 3 \}
Function subtyping rule

more “precise” type

\[ P <: P' \quad \text{c : P' \to Q'} \quad Q' <: Q \]

less “precise” type

\[ \text{c : P \to Q} \]
Function subtyping rule

\[
P <: P' \quad Q' <: Q
\]

\[
P' \to Q' <: P \to Q
\]
Logical variables

\[ \forall a, b, \]
\[(\{x = a \land y = b\} \ t := x; \ x := y; \ y := t \ \{x = b \land y = a\}) \]
Conditionals and loops

\[
\begin{align*}
\{ P \land e \} & \quad c_1 & \quad \{ Q \} & \quad \{ P \land \neg e \} & \quad c_2 & \quad \{ Q \} \\
\{ P \} & \quad \text{if } e \text{ then } c_1 \text{ else } c_2 & \quad \{ Q \} \\
\end{align*}
\]
(Cond)

\[
\begin{align*}
\{ l \land e \} & \quad c & \quad \{ l \} \\
\{ l \} & \quad \text{while } e \text{ do } c & \quad \{ l \land \neg e \} \\
\end{align*}
\]
(While)

loop invariant (needs to be guessed)
Why Hoare logic doesn’t scale

- The language with mutable variables is too simplistic;
- The lack of procedures means the absence of modularity;
- But the main problem is adding pointers.
A language with pointers

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

This spec is \textit{wrong}!

\[
\{ x \mapsto - \land y \mapsto b \} \quad x ::= 3 \quad \{ x \mapsto 3 \land y \mapsto b \}
\]

assign value 3 to a pointer \( x \)

if \( x \) and \( y \) are \textit{aliases}, the value of \( y \) was affected
\{ x \mapsto - \land y \mapsto b \} \\
\ x \ ::= \ 3 \\
\{ x \mapsto 3 \land \\
\ (x \neq y \land y \mapsto b) \lor (x = y \land y \mapsto 3)\} \\

What about 3 variables? \\

...or an array?
Separation Logic

- 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.

\[
\{ x \mapsto - \land y \mapsto b \} \; x \; \triangleright= \; 3 \; \{ x \mapsto 3 \land y \mapsto b \}
\]
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 - \cdot y \mapsto b\} \quad x ::= 3 \quad \{h \mid h = x \mapsto 3 \cdot y \mapsto b\}

disjoint union of heaps
Revising the language and logic

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

{\text{h | } h = x \mapsto -} \quad x ::= e \quad \{\text{res, h | } h = x \mapsto e \land \text{res} = \text{tt}\}

(Write)
Reading from a pointer

\{h \mid h = x \mapsto v\} \!\! \{\text{res, } h \mid h = x \mapsto v \land \text{res} = v\}\]

(Read)
Frame rule

```
{h | P(h)} c {res, h | Q(res, h)}
```

```
{h | ∃h₁, h = h₁ • h' ∧ P(h₁)} c {res, h | ∃h₁, h = h₁ • h' ∧ Q(res, h₁)}
```

"frame" (universally quantified)
Anti-frame rule

\[
\{h \mid \exists h_1, \ h = h_1 \cdot h' \land P(h_1)\} \overset{c}{\longrightarrow} \{\text{res, } h \mid \exists h_1, \ h = h_1 \cdot h' \land Q(\text{res, } h_1)\}
\]

\[
\{h \mid P(h)\} \overset{c}{\longrightarrow} \{\text{res, } h \mid Q(\text{res, } h)\}
\]

(Anti-Frame)
Allocation/deallocation

\[
\{h \mid h = \text{emp}\} \quad \text{alloc}(e) \quad \{\text{res}, h \mid h = \text{res} \mapsto e\}
\]

\text{(Alloc)}

\[
\{h \mid h = x \mapsto -\} \quad \text{dealloc}(x) \quad \{\text{res}, h \mid h = \text{emp} \land \text{res} = \text{tt}\}
\]

\text{(Dealloc)}
The result of $c_1$ is *bound* within $c_2$ under a name $x$. 
"Oblivious" Binding

\[
\begin{align*}
\{h \mid P(h)\} & \quad c_1 \quad \{\text{res, } h \mid Q(h)\} \\
\{h \mid Q(h)\} & \quad c_2 \quad \{\text{res, } h \mid R(\text{res, } h)\}
\end{align*}
\]

\[\{h \mid P(h)\} \quad c_1 \quad ;; \quad c_2 \quad \{\text{res, } h \mid R(\text{res, } h)\} \quad \text{(BindO)}\]

The result of \( c_1 \) is \textit{irrelevant} for \( c_2 \).
The rule of conjunction

\[ \{ h \mid P_1(h) \} c \{ res, h \mid Q_1(res, h) \} \]
\[ \{ h \mid P_2(h) \} c \{ res, h \mid Q_2(res, h) \} \]
\[ \{ h \mid P_1(h) \land P_2(h) \} c \{ res, h \mid Q_1(res, h) \land Q_2(res, h) \} \]
Working with functions

\[
\{h \mid P(h)\} \text{ ret } e \{\text{res, } h \mid P(h) \land \text{res} = e\}
\]  
(Return)

\[
\forall x, \{h \mid P(x, h)\} f(x) \{\text{res, } h \mid Q(x, \text{res, } h)\} \in \Gamma
\]  
(Hyp)

\[\Gamma \vdash \forall x, \{h \mid P(x, h)\} f(x) \{\text{res, } h \mid Q(x, \text{res, } h)\}\]

\text{a context of assumed functions' specs}

\[
\Gamma \vdash \forall x, \{h \mid P(x, h)\} f(x) \{\text{res, } h \mid Q(x, \text{res, } h)\}
\]
\[
\Gamma \vdash \forall x, \{h \mid P(e, h)\} f(e) \{\text{res, } h \mid Q(e, \text{res, } h)\}
\]

(App)
Representing loops using recursive functions
Fixed point combinator

\[ \text{fix} : (T \rightarrow T) \rightarrow T \]

\[ \text{fix } f = f (\text{fix } f) \]

- A way to implement general recursion in pure calculi;
- **Cannot** be encoded within a primitively-recursive language;
- Its argument \( f \) should be *continuous* (in Scott’s topology).
Factorial implementation using \texttt{fix}

\[ T = \text{nat} \rightarrow \text{nat} \]

\[ \text{fact} = \text{fix} \left( \text{fun} \ (f : \text{nat} \rightarrow \text{nat}) \Rightarrow \begin{array}{l} \text{fun} \ (n : \text{nat}) \Rightarrow \begin{array}{l} \text{if} \ n = 0 \text{ then } 1 \\ \text{else} \ n \ast f \ (n - 1) \end{array} \end{array} \right) \]

\[ T \rightarrow T = (\text{nat} \rightarrow \text{nat}) \rightarrow (\text{nat} \rightarrow \text{nat}) \]

\[ \text{fix} : (T \rightarrow T) \rightarrow T \]

\[ \text{fix} \ f = f \ (\text{fix} \ f) \]
Refactoring loops to functions

while e do c
Refactoring loops to functions

\[(\text{fix } f\ (x: \text{bool}). \text{ if } x \text{ then } c;; f(e') \text{ else } \text{ret } tt)\ (e)\]
A rule for recursive functions

Assuming a specification for $f$, one has to verify its body...

\[
\Gamma; \forall x, \{h \mid P(x, h)\} \; f(x) \; \{\text{res}, h \mid Q(x, \text{res}, h)\} \\
\vdash \{h \mid P(x, h)\} \; c \; \{\text{res}, h \mid Q(x, \text{res}, h)\} \\
\Gamma \vdash \forall y \; \{h \mid P(y, h)\} \; (\text{fix} \; f(x).c)(y) \; \{\text{res}, h \mid Q(y, \text{res}, h)\}
\]

which justifies the inference of the spec for $\text{fix}$.

Remark: $P$ and $Q$ here play the role of the loop invariant.
Verifying imperative programs in Separation Logic
A factorial implementation

```plaintext
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 }

f(N) \equiv \begin{cases} 
N \times f(N') & \text{if } N = N' + 1 \\
1 & \text{else}
\end{cases}

\{ h \mid h = \text{emp} \}

\{ \text{res, } h \mid h = \text{emp} \land \text{res} = f(N) \}
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
}
def fact_loop(tt) =
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

\[ F_{inv}(n, \text{acc}, N, h) \overset{\text{def}}{=} \exists n', a', (h = n \mapsto n' \cdot \text{acc} \mapsto a') \land (f(n') \times a' = f(N)) \]

fun fact_loop (_: unit): nat =
  (fix loop (_: unit).)
    a' <-- !\text{acc};
    n' <-- !n;
    if n' == 0 then ret a'
    else acc ::= a' \times n';
    n ::= n' - 1;;
    loop(tt))
fun fact_loop (_: unit): nat =

{h | F_inv(n, acc, N, h)}

(fix loop (_ : unit).

{h | \exists n\ ', (h = n \mapsto n\' \cdot acc \mapsto a') \land (f(n') \times a' = f(N))}
   a' <-- !acc;

{h | \exists n', (h = n \mapsto n' \cdot acc \mapsto a') \land (f(n') \times a' = f(N))}
   n' <-- !n;

{h | (h = n \mapsto n' \cdot acc \mapsto a') \land (f(n') \times a' = f(N))}
   if n' == 0 then ret a'

{res, h | (h = n \mapsto 0 \cdot acc \mapsto f(N)) \land (res = f(N))}

{res, h | F_inv(n, acc, N, h) \land (res = f(N))}
   else acc ::= a' \times n';;

{h | (h = n \mapsto n' \cdot acc \mapsto a' \times n') \land (f(n') \times a' = f(N))}
   n ::= n' - 1;;

{h | (h = n \mapsto n'-1 \cdot acc \mapsto a' \times n') \land (f(n') \times a' = f(N))}

{h | (h = n \mapsto n'-1 \cdot acc \mapsto a' \times n') \land (f(n'-1) \times a' \times n' = f(N))}
{h | F_inv(n, acc, N, h)}

loop(tt))

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

\text{fact\_loop}(tt)

\{res, h \mid F_{inv}(n, acc, N, h) \wedge (res = f(N))\}
fun fact (N : nat): nat = {
  {h | h = emp}
  n <-- alloc(N);
  {h | h = n \mapsto N}
  acc <-- alloc(1);
  {h | h = n \mapsto N \cdot acc \mapsto 1}
  {h | F_{inv}(n, acc, N, h)}
  res <-- fact_loop(tt);
  {h | F_{inv}(n, acc, N, h) \land (res = f(N))}
  {h | (h = n \mapsto - \cdot acc \mapsto -) \land (res = f(N))}
  dealloc(n);;
  {h | (h = acc \mapsto -) \land (res = f(N))}
  dealloc(acc);;
  {h | (h = emp) \land (res = f(N))}
  ret res }
{h | (h = emp) \land (res = f(N))}
\{ h \mid h = \text{emp} \}

\text{fact}(N)

\{ h \mid (h = \text{emp}) \land (\text{res} = 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:
  \[ tt \]
  \[
  (3 + 2)
  \]
  \[
  \text{fact (42)}
  \]

  **pure**: referentially-transparent, *always* evaluate to a result value

• Commands (aka computations, programs):
  \[
  \text{ret} \ 3
  \]
  \[
  x ::= 5
  \]
  \[
  c <- !x; \ y ::= x + 3;; \ t <- !y; \ \text{ret} \ t
  \]

  **effectful**: might diverge, write to store, throw exceptions
... 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.
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

class Monad m where

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

return :: a -> m a

type of computation

type of pure value
Monadic do-notation

\begin{align*}
  c1 & \gg= (\lambda x \rightarrow c2) \\
  \quad & \gg= (\lambda y \rightarrow c3)
\end{align*}

\begin{align*}
  a \rightarrow (m \ c) \\
  \quad & (m \ a) \\
  \quad & (m \ b) \\
  \quad & b \rightarrow (m \ c)
\end{align*}
Monadic do-notation

do  x <- c1
   y <- c2
   c3
Monadic do-notation

\[
\begin{align*}
\text{main} &= \text{do} \quad \text{putStrLn} \quad \text{"Enter a character"} \\
& \quad \text{c <- getChar} \quad :: \text{IO Char} \\
& \quad \text{putStrLn} \quad \text{"This was: "} \quad ++ \quad \text{[c]} \\
& \quad \text{return} \quad () \quad :: \text{IO ()}
\end{align*}
\]
Key Highlights

• Imperative programs perform \textit{effectful} computations;

• Hoare triples provide a way to specify their effect;

• Computations are composed using \textit{bind} and \textit{return};

• In functional programming, \textit{effects} are specified by \textit{monads};

• Monads are composed using \textit{bind} and \textit{return}. 
Hoare triples
+
Monadic types
Hoare Type Theory

Specifying and verifying effectful programs in Coq with dependent types.
Hoare types

\[ c : \{ x_1, x_2, \ldots \} \] \[ STsep(p, q) \]
8.4 Elements of Hoare Type Theory

All these observations resulted in a series of works on Hoare Type Theory (or just HTT), which defines a notion of an indexed Hoare monad (or, Hoare type) as a mechanism to encode Hoare-style specifications as dependent types and reduce the verification of errorful progress to proving propositions in Coq [41–43].

In the rest of this chapter we will consider a number of important concepts of HTT, so the necessary modules should be imported from the library folder `htt`, which contains the compiled files (see Section 1.3.3 for the instructions on obtaining and building HTT from the sources).

Require Import ssreflect ssrbool ssrnat eqtype seq ssrfun.
Add LoadPath "./../htt".
Require Import pred pcm unionmap heap heaptac stmod stsep stlog stlogR.

Set Implicit Arguments.
Unset Strict Implicit.
Unset Printing Implicit Defensive.

8.4.1 The Hoare monad

The Hoare monad (also dubbed as Hoare type), which is a type of result-returning effectful computations with pre- and postconditions, is represented in HTT by the type `STsep`, which is, in fact, just a notation for a more general but less tractable type `STspec`, whose details we do not present here, as they are quite technical and are not necessary in order to verify programs in HTT.

The Hoare type is usually specified using the HTT-provided notation as `{x1 x2 ...}`, `STsep (p, q)`, where `p` and `q` are the predicates, corresponding to the pre and postcondition with `p` being of type `heap æ Prop` and `q` of type `A æ heap æ Prop`, such that `A` is the type of the result of the command being specified. The identifiers `x1, x2` etc. bind the logical variables that are assumed to be universally quantified and can appear freely in `p` and `q`, similarly to the free variables in the specifications in Hoare logics (Section 8.1).

For example, the `alloc` function has the following (simplified compared to the original one) small footprint specification in the `STsep`-notation:

```
alloc : ∀ (A : Type) (v : A),
STsep (fun h ⇒ h = Unit,
[vfun (res : ptr) h ⇒ h = res :-> v])
```

That is, `alloc` is a procedure, which starts in an empty heap `Unit` and whose argument `v` of type `A` becomes referenced by the pointer (which is also the `alloc`'s result) in the resulting singleton-pointer heap. The notation `x :-> y` corresponds to the the points-to assertion `x `æ `y in the mathematical representation of separation logic, and `{vfun x ...}` notation accounts for the fact that the computation can throw an exception [42], the possibility we do not discuss in this course.

A curious reader can take a look at the definitions in the module `stmod` of the HTT library.

Example: allocator type

```
{h | h = emp}
```

```
{res, h | h = res «-> 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;
- one can implement a language with an arbitrary semantics.
- Examples: 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;

• the DSL’s semantics is essentially it host’s semantics.

• Examples: 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]