Hoare Triples

\[ (P \land \neg e \rightarrow Q) \land \\
\{ P \land e \} \; s \; \{ P' \} \land \\
\{ P' \} \; \text{while} \; e \; \{ s \} \; \{ Q \} \]

\[ (P' \land \neg e \rightarrow Q) \land \\
\{ P' \land e \} \; s \; \{ P'' \} \land \\
(P'' \land \neg e \rightarrow Q) \land \\
\cdots \]

Invent infinitely-many conditions \( P^{(n)} \).
Loop Invariants

**New syntax** for writing loop invariants:

\[
\{P\} \text{ while } \{I\} \text{ e } \{s\} \{Q\}
\]

\[
P \rightarrow I \land \\
(I \land \neg e \rightarrow Q) \land \\
\{I \land e\} \text{ s } \{I\}
\]

**Weakest precondition** computed from invariant:

\[
WP(Q) = I \land (I \land \neg e \rightarrow Q) \land (I \land e \rightarrow WP[s](I))
\]
Bounding Iterations

Bound must **decrease** every iteration

Function that computes bound called the “measure” $M$

```latex
while e:
\{  I \land e \}\hspace{7em} \textbf{while } e:\hspace{7em}
\{  M() = n \}
\{  I \}\hspace{7em} \{  M() < n \}
```

**Can assume** $I$ and $e$ to prove measure decreases
Class Progress

Logical reasoning

Program logics

Static analysis

Expressions

Statements

Loops

Procedures
Procedures

Re-conceptualizing programs as collections of functions
   Naming, linking, and the type environment

Reusing pre-/post-conditions for function calls
   And ensuring that recursive functions terminate

Separation to allow modular function reasoning
   You can’t change what you can’t touch
Procedures

Breakthroughs from the 1970s
Function Calls

What does this script return in \( m1 \) and \( m2 \)?

\[
\begin{align*}
l1 &= \text{range}(-n, n) \\
l2 &= \text{map}(\text{abs}, l1) \\
m1 &= \text{max}(l1) \\
m2 &= \text{max}(l2)
\end{align*}
\]

But how are \textit{range}, \textit{map}, \textit{abs}, and \textit{max} implemented?
Functions

What are functions? Why do we use them?

- **Reuse** common functionality
- **Abstract** over common code
- **Reason** modularly about code
- **Isolate** code from its surroundings

Functions are present in **every modern language**
A function definition has a couple of **parts**:

```python
def f(x, y):
    s
    return e
```

A function is **defined** by them: `<f, [x, y], s, e>`
Other statements **call** a function:

\[ r = f(ex, ey) \]

The interpreter **links** the function; then **assigns** the arguments; then **runs** the body; then **saves** the output.

A “different” \( x \)

\[ x = ex; \quad y = ey; \quad s; \quad r = e \]

Store map: names to other data
Rewrite to have no function calls:

```python
def abs(x):
    if x > 0:
        return x
    else:
        return -x
```

```python
x = abs(y)
```
Verifying Procedures

Inverting preconditions and postconditions
Function Calls

Which of these pre-/post-conditions hold?

```python
def abs(x):
    \{ T \}
    if x > 0:
        return x
    else:
        return -x
    \{ return ≥ 0 \}

def max(l):
    \{ len(l) > 0 \}
    cur = l[0]
    for x in l[1:]:
        cur = max(cur, x)
    return cur
    \{ ∀i, return ≥ l[i] \}
```
Function Calls

Which of these **pre-/post-conditions** hold?

\[
\{ \top \} \; \text{abs}(x) \; \{ \; \text{return} \geq 0 \; \}
\]

\[
\{ \text{len}(l) > 0 \} \; \text{max}(l) \; \{ \; \forall i, \text{return} \geq l[i] \; \}
\]

\[
\{ \top \}
\begin{align*}
l_1 &= \text{range}(-n, n) \\
l_2 &= \text{map}(\text{abs}, l_1) \\
m_1 &= \text{max}(l_1) \\
m_2 &= \text{max}(l_2)
\end{align*}
\]

\[
\{ m_2 \geq 0 \land m_1 \geq n \}
\]
Function Verification

Verifying functions requires additional syntax:

```python
def f(x, y):
    { P }
    s
    return e
    { Q }
```

These pre/post-conditions are true when:

```plaintext
{ P } s; return = e { Q }
```
Function Verification

Stored $P$ and $Q$ are used to verify calls:

$$\begin{align*}
\{ & \ P[x := e_x, y := e_y] \} \\
& r = f(e_x, e_y) \\
& \{ & Q[\text{return} := r] \} \\
\end{align*}$$

Other calls can be rewritten into this form:

$$\begin{align*}
& r = f(e_1) + g(e_2) \\
& x = f(e_1) \\
& y = g(e_2) \\
& r = x + y
\end{align*}$$
Weakest Precondition

Weakest preconditions work \textbf{the same way}:

\textbf{Given} \quad < f, [x, y], s, e, P_f, Q_f >

\begin{align*}
WP[r = f(ex, ey)](Q) &= \quad \\
&= P_f[x := e_x, y := e_y] \land \\
&= Q_f[\text{return} := r] \to Q
\end{align*}

Note that \textbf{linking} must precede verification

This make \textbf{higher-order functions} quite hard to verify
Example

\[
\begin{align*}
\{ \top \} \ \text{range}(l, r) \ \{ \ \text{len}(\text{return}) = r - l \ \} \\
\{ \top \} \ \text{range}(l, r) \ \{ \ \forall i, \text{return}[i] = l + i \ \} \\
\{ \ \text{len}(l) > 0 \ \} \ \text{max}(l) \ \{ \ \forall i, \text{return} \geq l[i] \ \}
\end{align*}
\]

\[
l = \text{range}(-n, n); \ m = \text{max}(l) \ \{ \ m \leq n \ \}
\]

\[
WP[t](m \geq n) = \text{len}(l) > 0 \land (\forall i, m \geq l[i]) \rightarrow (m \leq n)
\]
\[
= \text{len}(l) > 0 \land \exists i, l[i] \leq n
\]

\[
WP[s](\text{len}(l) > 0) = \top \land (\text{len}(l) = n - (-n) \rightarrow \text{len}(l) > 0)
\]
\[
= 2n > 0
\]

\[
WP[s](\exists i, l[i] \leq n) = \top \land (\forall i, l[i] = -n + i) \rightarrow (\exists i, l[i] \leq n)
\]
\[
= \exists i, i - n \leq n
\]
Exercise

Compute the **weakest precondition**:

\[
\{ \top \} \quad \text{abs}(x) \quad \{ \text{return} \geq 0 \land (\text{return} = x \lor \text{return} = -x) \} \\
\{ \text{len}(l) > 0 \} \quad \text{max}(l) \quad \{ \forall i, \text{return} \geq l[i] \} \\
\]

\[
m = \text{max}(l); \quad a = \text{abs}(m) \quad \{ a \geq l[0] \} \\
\]
Recursion

Recursive function calls work just like any other
But, a recursive function may not terminate

```python
def f():
    { ⊤ }
    return f()
    { ⊥ }
```

```python
def g(x):
    { ⊤; x decreases }
    if x > 0:
        return g(x - 1)
    { ⊤ }
```

For loops, we prove a decreasing measure
Same idea for functions; prove decreasing on recursive calls

Mutual recursion extra tricky!
Programs + Logic

The friends we made along the way
Class Progress

- Logical reasoning
- Program logics
- Static analysis
- Expressions
- Statements
- Loops
- Procedures
What We Learned

How to evaluate expressions into logical formulas

Plus: path conditions, symbolic environments, and more

Extended symbolic environments to predicates

\{P\} s \{Q\}

If P true of a state, and s executed, then Q true after
Hoare Logic

Each \{P\} s \{Q\} is a logical statement

Weakest preconditions systematically generate that statement

Program verification via verification conditions

Convert each function to a logical statement
Solve verification conditions via a solver

Everything you need to build the Dafny language
Statement Types

Loops as a form of infinite statement

- Invariants a short-hand for verifying that statement
- Measures for proving a loop terminates

Functions for modular bits of code

- Reusing function pre-/post-conditions at call sites
- Measures for proving recursive functions terminate
The Frame Problem

Limiting access makes reasoning easier
Mutation

What is the value of \( a \) after execution?

```python
def f(a):
    { len(a) > 0 }
    a[0] = 1
    { a[0] = 1 }

a = [0]
f(a)
```

How can we \textbf{specify} this behavior?
Mutation

Specifications must describe **before and after** values:

```python
def f(a):
    {  len(a) > 0  }
    a[0] = 1
    {  a[0] = 1 ∧\forall i, i > 0 → a[i] = old(a)[i]  }

    a = [0, 1]
    f(a)
```

The **old** syntax refers to the value **before execution**
Framing

What is **strongest post-condition** after execution?

\[
\begin{align*}
&\{\top\}\ f(x) \ \{\top\} \\
&\{\top\}\ g(x, y) \ \{\top\}
\end{align*}
\]

\[
\begin{align*}
a &= [0] & a &= [0] & a &= [0] \\
b &= [0] & b &= [0] & b &= a \\
f(a) & & g(a, b) & & g(a, a)
\end{align*}
\]

Applying a function can **change its arguments**

\[
\begin{align*}
&\{\top\}\ f(x) \ \{\ b = \text{old}(b)\ \}
\end{align*}
\]
Framing

Attempted fix to preserve facts about other variables:

\[
\{ P(x) \} \quad f(x) \quad \{ Q(x) \}
\]

\[
\downarrow
\]

\[
\{ P(x) \land F(y) \} \quad f(x) \quad \{ Q(x) \land F(y) \}
\]

“Frame” or context

Problem: what about relationships between variables?

\[
\{ \ a[0] = 0 \ \} \quad f(a) \quad \{ \ a[0] = 1 \ \}
\]

\[
\rightarrow
\]

\[
\{ \ a[0] = 0 \land a[0] < b[0] \ \} \quad f(a) \quad \{ \ a[0] = 1 \land a[0] < b[0] \ \}
\]
Separation

Want to separate variables into two groups:

- **Written** by the function $W = \{a\}$
- **Only read** or ignored $R = \{b\}$

Split **clauses in precondition** in the same way:

$$\{ P(W \cup R) \land F(R) \} \quad f(x) \quad \{ Q(W \cup R) \land F(R) \}$$

“Separating And”
Examples

Add syntax to describe \textbf{read/written} variables:

\[
\begin{array}{l}
\{ \top \} \; f(x) \; \{ \top \} \; \text{writes } x \\
\{ \top \} \; g(x, y) \; \{ \top \} \; \text{writes } x \\
a = [0] \\
b = [0] \\
f(a) \\
g(a, b) \\
g(a, a) \\
\end{array}
\]

If mutable values are \textbf{returned}, need to track identity
Next class:
Milestone I

To do:
- Course feedback
- Milestone I presentation
- Assignment 4
Procedures

Re-conceptualizing programs as collections of functions
  Naming, linking, and the type environment

Reusing pre-/post-conditions for function calls
  And ensuring that recursive functions terminate

Separation to allow modular function reasoning
  You can’t change what you can’t touch
Scaling solvers are slow.
Abstract

Interpretation

Propositions

As Data
DEPENDENT TYPES

DATA AS

PROPOSITIONS
Next class:

Milestone I

To do:

☐ Course feedback
☐ Milestone I presentation
☐ Assignment 4