Procedures

Programs section, **Lecture 16**

Pavel Panchekha

CS 6110, U of Utah

27 February 2020

Hoare Triples

$(P \wedge \neg e \rightarrow Q) \wedge$
`{P ∧ e} s {P’}` $\wedge$
`{ P’} while e { s } {Q}`

↓

$(P \wedge \neg e \rightarrow Q) \wedge$
`{P ∧ e} s {P’}` $\wedge$
$(P' \wedge \neg e \rightarrow Q) \wedge$
`{P’ ∧ e} s {P’’}` $\wedge$
$(P'' \wedge \neg e \rightarrow Q) \wedge$
...

Invent infinitely-many conditions $P^{(n)}$

Loop Invariants

New syntax for writing loop invariants:

```
{P} while {I} e { s } {Q}
```

$P \to I \land$

$(I \land \neg e \to Q) \land$

```
{I ∧ e} s {I}
```

Weakest precondition computed from invariant:

$$WP(Q) = I \land (I \land \neg e \to Q) \land (I \land e \to WP[s](I))$$

Bounding Iterations

Bound must **decrease** every iteration

- Function that computes bound called the "measure" M

```
while e:
  { I ∧ e }
  s
  { I }
```

```
while e:
  { M() = n }
  s
  { 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`**?

```
l1 = range(-n, n)
l2 = map(abs, l1)
m1 = max(l1)
m2 = max(l2)
```

[-n, -n+1, …, n-1]

But how are **range**, **map**, **abs**, and **max** implemented?

Functions

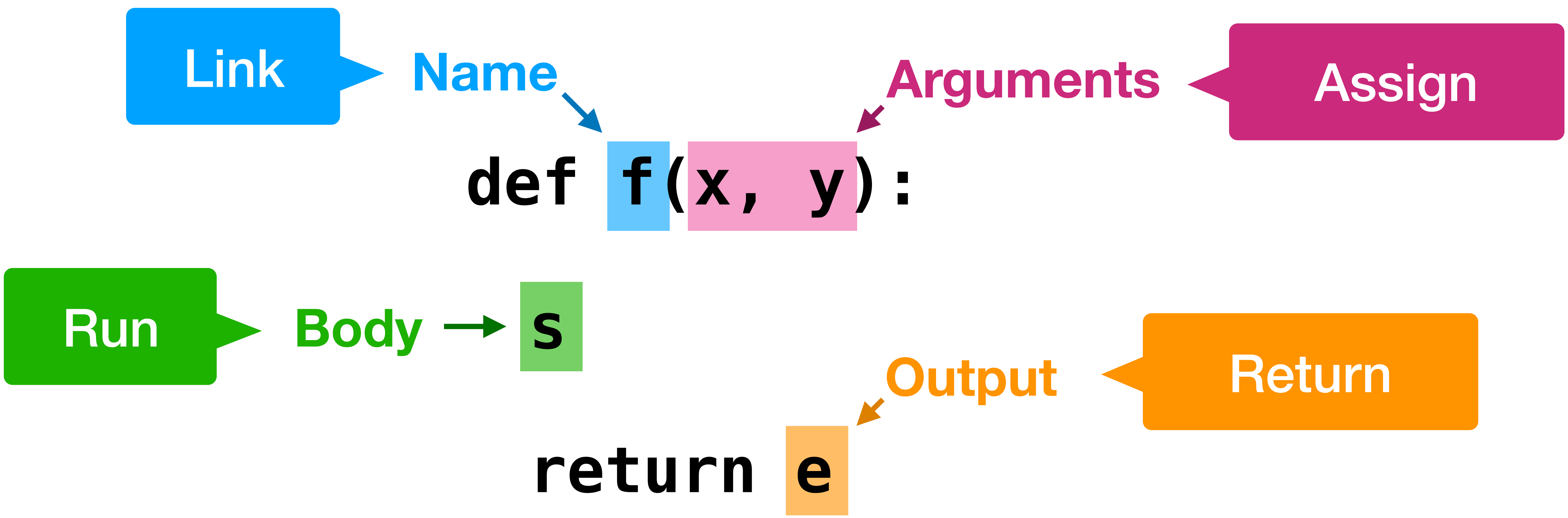
What are functions? Why do we use them?

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

Functions are present in **every modern language**

Function Anatomy

A function definition has a couple of **parts**:



A function is **defined** by them: $< f, [x, y], s, e >$

How Functions Work

Other statements **call** a function:

Store map: names to other data

```
r = f(ex, ey)
```

$< f, [x, y], s, e >$

A “different” x

x `= ex;` y `= ey;` s`;` `r =` e

The interpreter **links** the function;

then **assigns** the arguments;

then **runs** the body;

then **saves** the output.

Exercise

Rewrite to have **no function calls**:

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

x = abs(y)
```

Verifying Procedures

Inverting preconditions and postconditions

Function Calls

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

```
def abs(x):
  { ⊤ }
  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?

$\{\ \top\ \}$ `abs(x)` $\{\ \texttt{return} \geq 0\ \}$

$\{\ \texttt{len}(l) > 0\ \}$ `max(l)` $\{\ \forall i, \texttt{return} \geq l[i]\ \}$

$\{\ \top\ \}$

```
l1 = range(-n, n)
l2 = map(abs, l1)
m1 = max(l1)
m2 = max(l2)
```

$\{\ m_2 \geq 0 \land m_1 \geq n\ \}$

Function Verification

Verifying functions requires **additional syntax**:

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

→ $< f, [x, y], s, e, P, Q >$

Store P, Q in function data

These pre/post-conditions are true when:

$$\{\ P\ \}\ \texttt{s; return = e}\ \{\ Q\ \}$$

Function Verification

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

{ $P[x := e_x, y := e_y]$ }
`r = f(ex, ey)`
{ $Q[$`return` $:= r]$ }

$< f, [x, y], s, e, P, Q >$

Other calls can be **rewritten** into this form:

```
r = f(e1) + g(e2)
```

```
x = f(e1)
y = g(e2)
r = x + y
```

Weakest Precondition

Weakest preconditions work **the same way**:

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

$$WP[\texttt{\textbf{r = f(ex, ey)}}](Q) =$$

$$P_f[x := e_x, y := e_y] \land$$

$$Q_f[\texttt{\textbf{return}} := r] \rightarrow Q$$

Note that **linking** must precede verification

This make **higher-order functions** quite hard to verify

Example

$\{\ \top\ \}$ `range(l, r)` $\{\ \mathtt{len}(\mathtt{return}) = r - l\ \}$

$\{\ \top\ \}$ `range(l, r)` $\{\ \forall i, \mathtt{return}[i] = l + i\ \}$

$\{\ \mathtt{len}(l) > 0\ \}$ `max(l)` $\{\ \forall i, \mathtt{return} \geq l[i]\ \}$

$$_s\underline{\texttt{l = range(-n, n);}}\ _t\underline{\texttt{m = max(l)}}\ \{\ m \leq n\ \}$$

$$WP[t](m \geq n) = \textbf{len}(l) > 0 \land (\forall i, m \geq l[i]) \to (m \leq n)$$

$$= \textbf{len}(l) > 0 \land \exists i, l[i] \leq n$$

$$WP[s](\textbf{len}(l) > 0) = \top \land (\textbf{len}(l) = n - (-n) \to \textbf{len}(l) > 0)$$

$$= 2n > 0$$

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

$$= \exists i, i - n \leq n$$

Exercise

Compute the **weakest precondition**:

$\{\ \top\ \}$ `abs(x)` $\{\ \texttt{return} \geq 0 \land (\texttt{return} = x \lor \texttt{return} = -x)\ \}$

$\{\ \texttt{len}(l) > 0\ \}$ `max(l)` $\{\ \forall i, \texttt{return} \geq l[i]\ \}$

$$ {}_{s}\underline{\texttt{m = max(l);}}\ {}_{t}\underline{\texttt{a = abs(m)}}\ \ \{\ a \geq l[0]\ \} $$

Recursion

Recursive function calls work **just like** any other

But, a recursive function **may not terminate**

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

```
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**

$$\{P\}\ \mathbf{s}\ \{Q\}$$

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?

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

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

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

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

How can we **specify** this behavior?

Mutation

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

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

$\{ \top \}$ `f(x)` $\{ \top \}$
$\{ \top \}$ `g(x, y)` $\{ \top \}$

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

```
a = [0]
b = [0]
g(a, b)
```

```
a = [0]
b = a
g(a, a)
```

Applying a function can **change its arguments**

$\{ \top \}$ `f(x)` $\{ b = \mathtt{old}(b) \}$?

Framing

Attempted fix to preserve facts about other variables:

$$\{\ P(x)\ \}\ \mathtt{f(x)}\ \{\ Q(x)\ \}$$

$$\downarrow$$

$$\{\ P(x) \wedge \underline{F(y)}\ \}\ \mathtt{f(x)}\ \{\ Q(x) \wedge \underline{F(y)}\ \}$$

"Frame" or context

Problem: what about **relationships** between variables?

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

$$\{\ a[0] = 0 \wedge a[0] < b[0]\ \}$$
```
f(a)
```
$$\{\ a[0] = 1 \wedge a[0] < b[0]\ \}$$

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:

$$\{ \ P(W \cup R) \underline{*} F(R) \ \} \ \mathtt{f(x)} \ \{ \ Q(W \cup R) \underline{*} F(R) \ \}$$

"Separating And"

Examples

Add syntax to describe **read/written** variables:

```
{ ⊤ } f(x) { ⊤ } writes x
{ ⊤ } g(x, y) { ⊤ } writes x
```

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

```
a = [0]
b = [0]
g(a, b)
```

```
a = [0]
b = [0]
g(a, a)
```

If mutable values are **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