Preconditions and Postconditions

Test Properties, Preconditions and Postconditions

Let **a** be some type. Then a -> Bool is the type of properties of **a**.

An **a** property is a function for **classifying a** objects.

Properties can be used for testing:

Let f be a function of type a -> a.

A **precondition** for f is a property of the input.

A **postcondition** for f is a property of the output.

Hoare Statements, or Hoare Triples

`{p} f {q}`

Read: if the input **x** of **f** satisfies **p**, then the output **f x** satisfies **q**
Intuitively, special case of specifications for design by contract software development.

$$\{\text{even}\}(\lambda x \mapsto x+1)\ \{\text{odd}\}$$

$$\{\text{odd}\}(\lambda x \mapsto x+1)\ \{\text{even}\}$$

$$\{\top\}(\lambda x \mapsto 2x)\ \{\text{even}\}$$

$$\{\bot\}(\lambda x \mapsto 2x)\ \{\text{odd}\}$$

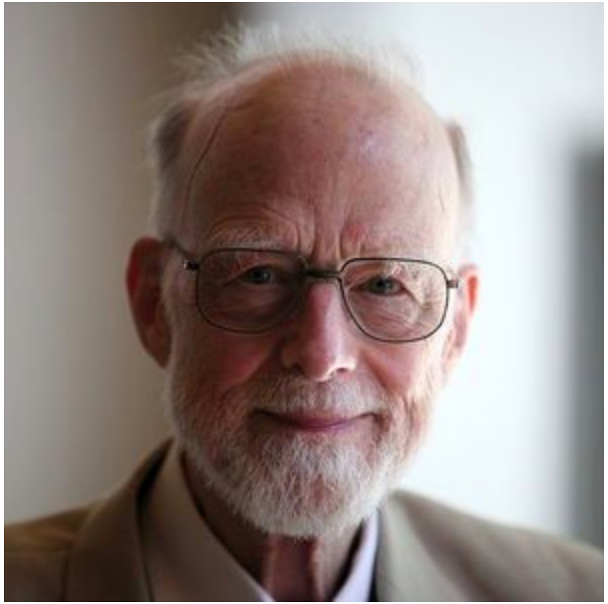
Hoare

Formalizing inter-process communication

Quicksort

1980 Turing Award

Null pointer :)



Testing with Quicksort

```
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
  quicksort [ a | a <- xs, a <= x ]
  ++ [x]
  ++ quicksort [ a | a <- xs, a > x ]
```

Property: Quicksort turns *any* finite list of items into an *ordered* list of items.

Precondition?

Testing with Quicksort

```
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
  quicksort [ a | a <- xs, a <= x ]
  ++ [x]
  ++ quicksort [ a | a <- xs, a > x ]
```

Property: Quicksort turns *any* finite list of items into an *ordered* list of items.

Precondition: the property `isTrue` that holds for any input

Postcondition ?

Testing with Quicksort

```
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
   quicksort [ a | a <- xs, a <= x ]
   ++ [x]
   ++ quicksort [ a | a <- xs, a > x ]
```

Property: Quicksort turns *any* finite list of items into an *ordered* list of items.

Precondition: the property `isTrue` that holds for any input

```
isTrue :: a -> Bool
isTrue _ = True
```

Postcondition: the property `prop_ordered` that should hold for any output list

```
prop_ordered :: Ord a => [a] -> Bool
prop_ordered [] = True
prop_ordered (x:xs) = all (>= x) xs && prop_ordered xs
```

A Hoare triple for Quicksort

`{ isTrue xs } ys = quicksort xs { prop_ordered ys }`

And we are ready for automated testing.

Build your own Quickcheck

Automated test generation

```
testR :: Int -> Int -> ([Int] -> [Int])
                 -> ([Int] -> [Int] -> Bool) -> IO ()
testR k n f r =     if k == n then print (show n ++ " tests passed")
                    else do
                            xs <- genIntList
                            if r xs (f xs) then
                                    do print ("pass on: " ++ show xs)
                                    testR (k+1) n f r
                            else error ("failed test on: " ++ show xs)
```

Example: running 100 tests with a given postcondition

```
testPost :: ([Int] -> [Int]) -> ([Int] -> Bool) -> IO ()
testPost f p = testR 1 100 f (\_ -> p)
```

Tools for Test Generation: Random Numbers

Random number generation. Getting a random integer:

```
*Lecture2> :t randomR
randomR :: (RandomGen g, Random a) => (a, a) -> g -> (a, g)

getRandomInt n = getStdRandom (randomR (0,n))

*Lecture2> getRandomInt 20
15

*Lecture2> :t getRandomInt
getRandomInt :: Int -> IO Int
```

Randomly flipping the value of an Int

```
randomFlip x = do
   b <- getRandomInt 1
   if b==0 then return x else return (-x)
```

```
*Lecture2> :t randomFlip
randomFlip :: Int -> IO Int
```

Random integer list

```
genIntList :: IO [Int]
genIntList = do
  k <- getRandomInt 20
  n <- getRandomInt 10
  getIntL k n

getIntL :: Int -> Int -> IO [Int]
getIntL _ 0 = return []
getIntL k n = do
   x <-  getRandomInt k
   y <- randomFlip x
   xs <- getIntL k (n-1)
   return (y:xs)
```

What is the role of k? What is the role of n?

[More on monads]()

Some output lines

```
*Lecture2> genIntList
[-2,-10,-1,9,0,4,-5,-9,-11,9]
*Lecture2> genIntList
[]
*Lecture2> genIntList
[1,2,11,13,8,12,6]
*Lecture2> genIntList
[-14,0,5,-7,-10,-6,-1,-3]
*Lecture2> genIntList
[-5,-1]
*Lecture2> genIntList
[-10,17,6,-13,-18,8,-12,-18,10,-15]
*Lecture2> genIntList
[-14,7,6,-14,7,13]
```

Quickcheck on Quicksort

Our own version

```
*Lecture2> testPost quicksort prop_ordered
"pass on: [-8,-9,-11,5,-11,1,-3,7]"
"pass on: [0]"
...
"pass on: [0,-8,-1,4,4,-3,-8]"
"100 tests passed"
```

The original version

```
*Lecture2> quickCheck (prop_ordered . quicksort)
+++ OK, passed 100 tests.
```

Another Quicksort

Let’s write a slightly different implementation:

```
quicksrt :: Ord a => [a] -> [a]
quicksrt [] = []
quicksrt (x:xs) =
  quicksrt [ a | a <- xs, a < x ]
  ++ [x]
  ++ quicksrt [ a | a <- xs, a > x ]
```

Passes the tests using the ordered property:

```
*Lecture2> quickCheck (prop_ordered . quicksrt)
+++ OK, passed 100 tests.
```

Is there a postcondition property that quicksort has, but quicksrt lacks?

Quicksort versus Quicksrt with properties

Yes, there is a postcondition property that quicksort has, but quicksrt lacks:

```
samelength :: [Int] -> [Int] -> Bool
samelength xs ys = length xs == length ys
```

As testPost expects properties with a different type, we write a new test function:

```
testRl :: ([Int] -> [Int]) -> ([Int] -> [Int] -> Bool) -> IO ()
testRl f r = testR 1 100 f r
```

And we test with:

```
*Lecture2> testRl quicksrt samelength
"pass on: [1,4,2,-1,-6,-14,-10]"
*** Exception: failed test on: [5,0,3,-3,-1,0,-4,-4,2,-3]
```

Or

```
*Lecture2>  quickCheck (\ xs -> samelength xs (quicksrt xs))
*** Failed! Falsifiable (after 7 tests and 1 shrink):
[-3,-3]
```

Precondition strengthening

If `{p} f {q}` holds

And p’ is a *stronger* property than p

Then `{p’} f {q}` holds.

What is makes a property p’ stronger than property p?

Revisit isTrue property: A stronger property is “being different from the empty list”.

```
{ not.null xs } ys = quicksort xs { sorted ys }
```

Postcondition weakening

If `{p} f {q}` holds

And q’ is a *weaker* property than q

Then `{p} f {q’}` holds.

Implementing Hoare Logic tests

To actually run Hoare tests, we need a domain of test instances and a way to generate them, from **{p} f {q}** to **p x → q (f x)**

```
infix 1 -->

(-->) :: Bool -> Bool -> Bool
p --> q = (not p) || q

forall :: [a] -> (a -> Bool) -> Bool
forall = flip all


hoareTest :: (a -> Bool) -> (a -> a) -> (a -> Bool) -> [a] -> Bool
hoareTest precondition f postcondition =
    all (\x -> precondition x --> postcondition (f x))

hoareTest odd succ even [0..100]
```

Recognizing the Relevant Test Cases

The *relevant* test cases are the cases that satisfy the precondition.

The following function keeps track of the proportion of relevant tests:

```
hoareTestR ::  Fractional t =>
          (a -> Bool) -> (a -> a) -> (a -> Bool) -> [a] -> (Bool,t)
hoareTestR precond f postcond testcases = let
    a = fromIntegral (length $ filter precond testcases)
    b = fromIntegral (length testcases)
  in
    (all (\x -> precond x --> postcond (f x)) testcases,a/b)
```

Some output:

```
*Lecture2> hoareTest odd succ even [0..100]
True
*Lecture2> hoareTestR odd succ even [0..100]
(True,0.49504950495049505)
*Lecture2> hoareTestR (\_ -> True) succ even [0..100]
(False,1.0)
```

The Hoare rule for while statements

From `{p} f {q}` we derive `{p} while c f  {p . && . not.c}`

Property p is a **loop invariant**. Further reading on Hoare logic [Mike Gordon's notes]

```
invarTest :: (a -> Bool) -> (a -> a) -> [a] -> Bool
invarTest invar f = hoareTest invar f invar

*Lecture2>  invarTest odd (succ.succ) [0..100]
True
```

And the counting version:

```
invarTestR ::  Fractional t =>
                (a -> Bool) -> (a -> a) -> [a] -> (Bool,t)
invarTestR invar f = hoareTestR invar f invar

*Lecture2> invarTestR odd (succ.succ) [0..100]
(True,0.49504950495049505)
```

Again, QuickCheck

The predefined operator ==> allows us to check for precondition failures.

- What was the difference to --> ?

Let's take a simple example:

```
f1,f2 :: Int -> Int
f1 = \n -> sum [0..n]
f2 = \n -> (n*(n+1)) `div` 2

test = verboseCheck (\n -> n >= 0 ==> f1 n == f2 n)
```

Hoare triples - Expressing relations

What if we need to express *relations* between inputs and outputs of functions?

- For instance, preserving parity for `succ.succ`

```
parity n = mod n 2
```

A relational version of Hoare tests:

```
testRel :: (a -> a -> Bool) -> (a -> a) -> [a] -> Bool
testRel spec f = all (\x -> spec x (f x))
```

An invariant relation version (what is the main difference?):

```
testInvar :: Eq b => (a -> b) -> (a -> a) -> [a] -> Bool
testInvar specf = testRel (\ x y -> specf x == specf y)

*Lecture2> testInvar parity (succ.succ) [0..100]
True
```

Stronger and Weaker as Predicates on Test Properties

Stronger than and *weaker than* are relations on the class of test properties.

- We assume a finite domain given by [a]

```
stronger, weaker :: [a] ->
      (a -> Bool) -> (a -> Bool) -> Bool
stronger xs p q = forall xs (\ x -> p x --> q x)
weaker   xs p q = stronger xs q p
```

Use ⊤ for the property that *always* holds. This is the *weakest possible property*.

- Implementation: \ _ -> True. Remember the isTrue property.

Use ⊥ for the property that *never* holds. This is the strongest property.

- Implementation: \ _ -> False.

Everything satisfies \ _ -> True.

Nothing satisfies \ _ -> False.

Manipulating Properties

- Negating a property

```
neg :: (a -> Bool) -> a -> Bool
neg p = \ x -> not (p x)
```

Also (not.) ; \ p -> not . p ; \ p x -> not (p x)

- Conjunctions and Disjunctions of Properties

```
infixl 2 .&&.
infixl 2 .||.

(.&&.) :: (a -> Bool) -> (a -> Bool) -> a -> Bool
p .&&. q = \ x -> p x && q x

(.||.) :: (a -> Bool) -> (a -> Bool) -> a -> Bool
p .||. q = \ x -> p x || q x
```

What is the difference between && and .&&.?

Useful bits of code

```
compar :: [a] -> (a -> Bool) -> (a -> Bool) -> String
compar xs p q = let pq = stronger xs p q
                    qp = stronger xs q p
                in
                  if pq && qp then "equivalent"
                  else if pq  then "stronger"
                  else if qp  then "weaker"
                  else             "incomparable"

compar [0..10] even (even Lecture2..&&. (>3))
compar [0..10] even (even Lecture2..||. (>3))
```

Importance of Precondition Strengthening for Testing

If you *strengthen* the requirements on your *input*, your testing procedure gets *weaker*.

Reason: the set of *relevant tests* gets smaller.

Remember: the precondition specifies the *relevant tests*.

Should preconditions be *as weak as possible*?

Importance of Postcondition Weakening for Testing

If you *weaken* the requirements on your *output*, your testing procedure gets *weaker*.

Reason: the requirements that you use for verifying the output get less strict.

Should postconditions be *as strong as possible*?

Next time: Function composition, pre-/post-conditions

If `{p} f {q}` and `{q} g {r}` hold

Then `{p} g . f {r}` holds.

→ derive useful specifications for compositions from specifications for their parts.

Flipped order:

```
infixl 2 #

(#) :: (a -> b) -> (b -> c) -> (a -> c)
(#) = flip (.)
```

We can write `{p}  f # g {r}` holds. (read `f` followed by `g`)

Finding (loop) invariants

- Consider

```
r := 1
i := 0
while i<m do
    r := r*n
    i := i+1
```

- Prove that it computes n^m and leaves the result in r → postcondition $r=n^m$

What should be our precondition? What should be the loop invariant?

Full example at https://www.cs.cmu.edu/~aldrich/courses/15-819O-13sp/resources/hoare-logic.pdf

Finding (loop) invariants

- Precondition should be as weak as possible
  - We know nothing about division by n, so m >= 0
  - We want to avoid 0^0, so n>0
- Formulating a loop invariant: change postcondition to depend on loop index rather than some other variable
  - $r=n^m \rightarrow r=n^i$
  - Add loop condition: i<=m
  - Add conditions for loop body correctness: 0<=i^n>0
  - Loop invariant: $r=n^i$^0<=i<=m^n>0
- {m>=0^n>0} r:=1; i:=0; {$r=n^i$^0<=i<=m^n>0}
- Apply twice from base case
- Consider induction case i and show for i+1