# Reasoning About Imperative Programs

COS 441 Slides 10

# Agenda

- The last few weeks
  - reasoning about functional programming
    - It's very simple and very uniform:  substitution of equal expressions for equal expressions
    - It works for any kind of data structure:  Integers, lists, strings, trees; arbitrary user-defined data types; even actions that describe I/O effects
- The next few lectures
  - reasoning about imperative programs
    - It's fundamentally more complicated
    - In a very practical sense, this means it is fundamentally more difficult to write correct imperative programs
    - In addition to having to worry about *what* is true, you have to worry about *when* it is true

# THE PROBLEM

# A Simple Haskell Program

some Haskell definitions:

```
pair x y = (x,y)

sum (x1, y1) (x2, y2) = (x1+y1, x2+y2)

x = pair 2 3

y = x
```

# A Simple Haskell Program

some Haskell definitions:

```
pair x y = (x,y)

sum (x1, y1) (x2, y2) = (x1+y1, x2+y2)

x = pair 2 3

y = x
```

what is `sum x y` equal to?

# A Simple Haskell Program

some Haskell definitions:

```
pair x y = (x,y)

sum (x1, y1) (x2, y2) = (x1+y1, x2+y2)

x = pair 2 3

y = x
```

what is `sum x y` equal to?

```
  sum x y
= sum x x
= sum (pair 2 3) (pair 2 3)
= (2+2, 3+3)
= (4, 6)
```

you should be able to verify this in your sleep

# A Somewhat Similar Java Program

the Java definitions:

```
class Pair {
  int x, y;

  Pair (int a1, int a2) {
     x = a1;
     y = a2;
 }

  static void sum (Pair p1, Pair p2) {
    p2.x = p1.x + p2.x;
    p2.y = p1.y + p2.y;
  }
}

Pair p1 = new Pair (2, 3);

Pair p2 = p1;
```

# A Somewhat Similar Java Program

the Java definitions:

```
class Pair {
  int x, y;

  Pair (int a1, int a2) {
     x = a1;
     y = a2;
 }

  static void sum (Pair p1, Pair p2) {
    p2.x = p1.x + p2.x;
    p2.y = p1.y + p2.y;
  }
}

Pair p1 = new Pair (2, 3);

Pair p2 = p1;
```

a big departure from the Haskell program; we are imperatively updating the contents of the pair data structure with the sum!

(an aside:  notice how much more verbose Java is than Haskell!)

# A Somewhat Similar Java Program

the Java definitions:

```
class Pair {
  int x, y;

  Pair (int a1, int a2) {
     x = a1;
     y = a2;
  }

  static void sum (Pair p1, Pair p2) {
    p2.x = p1.x + p2.x;
    p2.y = p1.y + p2.y;
  }
}

Pair p1 = new Pair (2, 3);

Pair p2 = p1;
```

consider the statement:

```
sum(p1, p2);
Pair p3 = p2;
```

what is p3 equal to?

```
Is p3.x == p1.x + p2.x    ?
Is p3.y == p1.y + p2.y    ?
```

# A Somewhat Similar Java Program

the Java definitions:

```
class Pair {
  int x, y;

  Pair (int a1, int a2) {
     x = a1;
     y = a2;
  }

  static void sum (Pair p1, Pair p2) {
    p2.x = p1.x + p2.x;
    p2.y = p1.y + p2.y;
  }
}

Pair p1 = new Pair (2, 3);

Pair p2 = p1;
```

consider the statement:

```
sum(p1, p2);
Pair p3 = p2;
```

what is p3 equal to?

Is p3.x == p1.x + p2.x ?

Is p3.y == p1.y + p2.y ?

p2.x actually takes on *different values at different times* during the computation

Reasoning by simple equality, ignoring state changes completely breaks down

# A Little Bit More Java

consider these statements:

```
Pair p1 = new Pair(2, 3);
Pair p2 = p1

....
```

suppose p2 does not show up anywhere else in the code

what is p2 equal to?

# A Little Bit More Java

consider these statements:

```
Pair p1 = new Pair(2, 3);
Pair p2 = p1

....
```

suppose p2 does not show up anywhere else in the code

what is p2 equal to? Who knows!?!?

Example:

```
Pair p1 = new Pair(2, 3);
Pair p2 = p1
p1.x = 17;
p1.y = 23;

// p2 != (2,3)
```

# A Little Bit More Java

consider these statements:

```
Pair p1 = new Pair (2, 3);
Pair p2 = p1

....
```

suppose p2 does not show up anywhere else in the code

what is p2 equal to? Isn’t it equal to p1 at least?

# A Little Bit More Java

consider these statements:

```
Pair p1 = new Pair (2, 3);
Pair p2 = p1

....
```

suppose p2 does not show up anywhere else in the code

what is p2 equal to? Isn’t it equal to p1 at least?

Nope. Example:

```
Pair p1 = new Pair (2, 3);
Pair p2 = p1
p1 = new Pair (7,13)

// p1 != p2
```

# Dramatic Differences

**Haskell**

- Variables are constant
- Data structures are immutable
- Properties of data are stable
- Local reasoning is easy
  - if p1 = (2,3) now, no intermittent code "..." changes that fact
- Code is more modular
- Order of definitions is irrelevant (provided names don't clash)
- Except when there are explicit dependencies, program parts can be run in parallel

**Java**

- Variables are updated
- Data structures are mutable
- Properties of data are unstable
- Local reasoning is hard
  - if p1 = (2,3) now, who knows what it will be after some intermittent code "..."
- Code is less modular
- Order of statements is crucial
- Program parts generally cannot be run in parallel

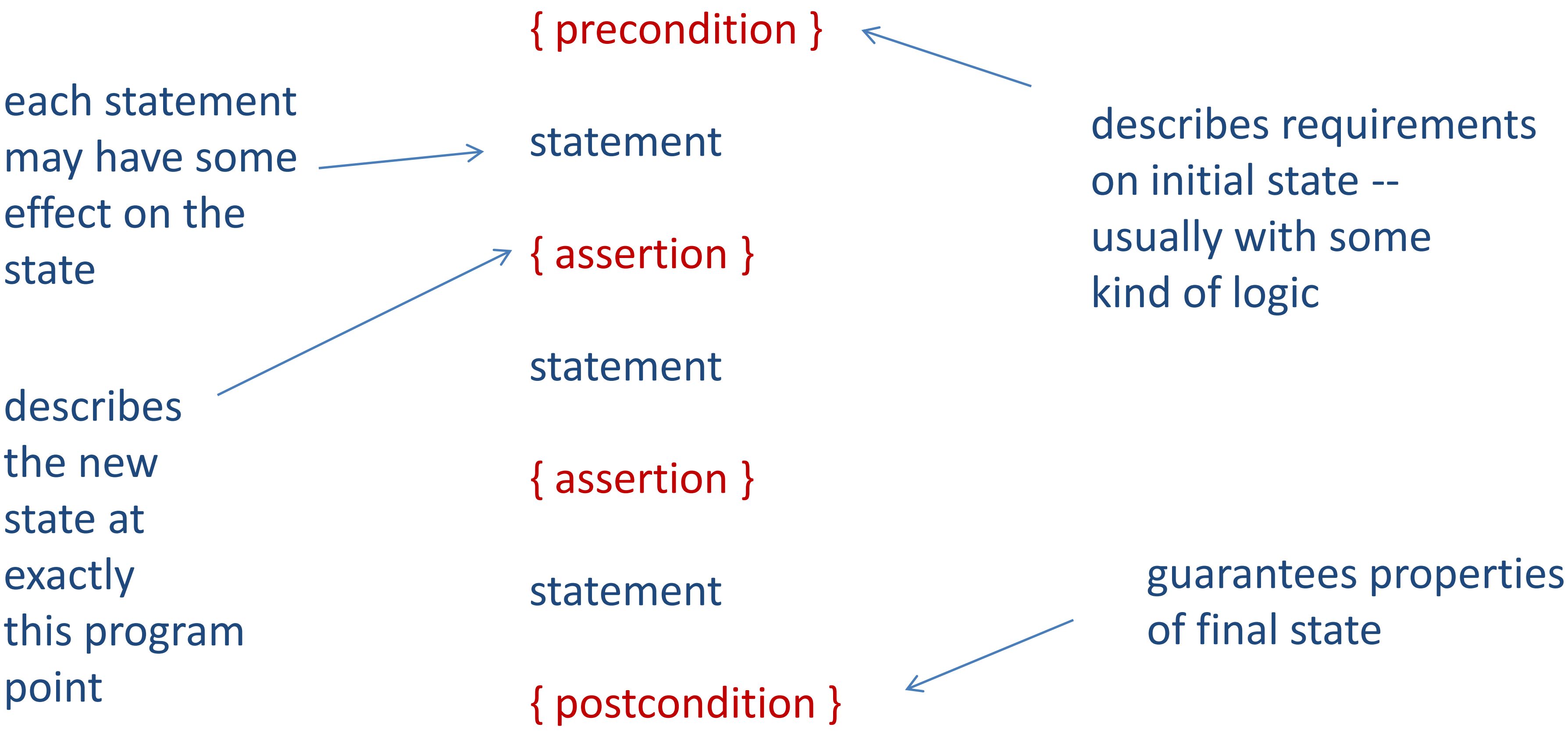
# FLOYD-HOARE LOGIC: AN OVERVIEW

# Hoare Logic: An Overview

- We can’t count on stable properties of data, so what we will do instead is analyze the state of the computation in between every statement:

# Hoare Logic: An Overview

- We can’t count on stable properties of data, so what we will do instead is analyze the state of the computation in between every statement:



{ precondition }

statement

{ assertion }

statement

{ assertion }

statement

{ postcondition }

each statement may have some effect on the state

describes the new state at exactly this program point

describes requirements on initial state -- usually with some kind of logic

guarantees properties of final state

# Hoare Logic:  An Example

- We can't count on stable properties of data, so what we will do instead is analyze the state of the computation in between every statement:

```
z = x + y;


x = x - 1;
```

# Hoare Logic: An Example

- We can't count on stable properties of data, so what we will do instead is analyze the state of the computation in between every statement:

```
{ x > 0 & y > 0 }

z = x + y;



x = x - 1;
```

# Hoare Logic: An Example

- We can't count on stable properties of data, so what we will do instead is analyze the state of the computation in between every statement:

```
{ x > 0 & y > 0 }

z = x + y;

{ z = x + y & x > 0 & y > 0 }

x = x - 1;
```

# Hoare Logic: An Example

- We can’t count on stable properties of data, so what we will do instead is analyze the state of the computation in between every statement:

```
{ x > 0 & y > 0 }

z = x + y;

{ z = x + y & x > 0 & y > 0 }

x = x - 1;

{ z = x + 1 + y & x >= 0 & y > 0}
```

# Hoare Logic:  An Example

- We can’t count on stable properties of data, so what we will do instead is analyze the state of the computation in between every statement:

```
{ x > 0 & y > 0 }

z = x + y;

{ z = x + y & x > 0 & y > 0 }

x = x - 1;

{ z = x + 1 + y & x >= 0 & y > 0}

{ z > x + y  & x >= 0 & y > 0 }
```

# Hoare Logic:  An Example

- We can’t count on stable properties of data, so what we will do instead is analyze the state of the computation in between every statement:

```
{ x > 0 & y > 0 }

z = x + y;

{ z = x + y & x > 0 & y > 0 }

x = x - 1;

{ z = x + 1 + y & x >= 0 & y > 0}

{ z > x +  y  & x >= 0 & y > 0 }
```

A proof
of program
correctness

justifies

A
Hoare
Triple

```
{ x > 0 & y > 0 }
z = x + y;
x = x - 1;
{ z > x + y & x >= 0 & y > 0}
```

# Hoare Triples

- A (partial) Hoare triple has the form { P } C { Q } where
  - P is a precondition that describes allowed initial states
  - C is a (possibly compound C-like or Java-like) statement
  - Q is a postcondition that describes allowed final states

- A (partial) Hoare triple is valid if whenever we start in a state that satisfies the pre-condition P and execution of C terminates, we wind up in a state that satisfies Q

- A fully annotated program {P1} C1 {P2} C2 ... CK {PK+1} serves as a proof of validity for the triple {P1} C1 C2 ... CK {PK+1} provided each individual components {Pi} Ci {Pi+1} obeys the Rules of Hoare Logic

# Partial vs. Total Hoare Triples

- Partial Hoare Triples are valid even when a program does not terminate
- Total Hoare Triples are valid if the partial triple is valid and the program does terminate
- Partial triples are good for establishing safety properties
  - ie: certain “bad things” never happen
  - eg: an array is never indexed out of bounds
  - eg: a null pointer is never dereferenced
- Total triples are good for establishing liveness properties:
  - ie: eventually “something good” happens
  - eg: the program terminates and produces an answer
- Total triples are even more of a pain in the neck than partial ones so we are going to ignore them; fewer people use them

# DESCRIBING PROGRAM STATES

# Program States

- What is a program state?
  - It is a finite partial map from program variables to integer values

# Program States

- What is a program state?
  - It is a finite partial map from program variables to integer values

ie: function

a finite number of elements
in the domain of the map

partial:  not all variables
are necessarily present
(typically there are infinitely
many possible variables)

# Program States

- What is a program state?
  - It is a finite partial map from program variables to integer values

ie: function

a finite number of elements
in the domain of the map

partial: not all variables
are necessarily present
(typically there are infinitely
many possible variables)

  - Example: [x = 2, y = 17, z = 3]

# Program States

- What is a program state?
  - It is a finite partial map from program variables to integer values

a finite number of elements in the domain of the map

partial: not all variables are necessarily present (typically there are infinitely many possible variables)

ie: function

  - Example: [x = 2, y = 17, z = 3]
  - Finite partial maps E typically support several operations:
    - lookup: E(x)
    - update: E[x = N]
    - domain: dom(E)

a new map in which x is mapped to N but is otherwise the same as E

the set of variables in the domain of E

# Program States in Haskell

```
module State where

type Var   = String
type State = [(Var, Int)]

look :: State -> Var -> Maybe Int
look [] v = Nothing
look ((v',i):xs) v =
  if v == v'
  then Just i
  else look xs v

up :: State -> Var -> Int -> State
up [] v i = [(v,i)]
up ((v',i'):xs) v i =
  if v == v'
  then (v,i):xs
  else (v',i'):up xs v i
```

finite maps as lists;
we could implement
them as search trees
for greater efficiency

```
dom :: State -> [Var]
dom = map (\(v,i) -> v)
```

# Describing Program States

- We are going to use logic to describe program states
- For example, this formula:

(x = 3 & y = 0) || (x = 2 & y = 1)

- Describes this state:

[ x = 3, y = 0 ]

- And all of these:

[ x = 2, y = 1 ]

[ x = 2, y = 1, z = 0 ]

[ x = 2, y = 1, z = 1 ]

…

the formula does not necessarily have to constrain all the variables

# Formulae

Math

integer variables

```
x := x1 | x2 | x3 | … | y | z | …
```

integer expressions

```
e ::= N | x | e + e | e * e
```

predicates

```
p ::= e = e | e < e
```

formulae

```
f ::= true
    | false
    | p
    | f & f
    | f || f
    | ~f
```

I will also use P, Q, F for formulae

# Formulae

## Math

integer variables
x := x1 | x2 | x3 | … | y | z | …

integer expressions
e ::= N | x | e + e | e * e

predicates
p ::= e = e | e < e

formulae
```
f ::= true
    | false
    | p
    | f & f
    | f || f
    | ~f
```

## Haskell

```
type Var = String

data Exp =
    Const Int
  | Var Var
  | Add Exp Exp
  | Mult Exp Exp

data Pred =
     Eq Exp Exp
  | Less Exp Exp

data Form =
    Tru
  | Fal
  | Pred Pred
  | And Form Form
  | Or Form Form
  | Not Form
```

# Math vs. Haskell

- Denotational semantics: Math notation or Haskell notation?

- Haskell semantic definitions are clearer
- Haskell gives us an implementation that will evaluate formulae
- Math is more concise, especially in examples:
  - Add (Add (Const 3) (Const 4)) (Const 5) vs (3 + 4) + 5

- If I were writing an academic research paper, I'd do it in math
- For teaching, I'll give semantics first in Haskell but then show you how to redefine them using the standard mathematical notation

# SEMANTICS OF FORMULAE: PRESENTATION I: HASKELL

# Denotational Semantics

- Recall: A denotational semantics gives a meaning to newly defined syntactic objects by translating these objects in to a better understood language or mathematical object

- Denotational semantics of expressions:
  - esem :: State -> Exp -> Maybe Int
  - esem s e == Just n ====> "expression e in state s has value n"
  - esem s e == Nothing ====> "expression e is not defined in state s"

# Semantics of Expressions in Haskell

```
esem :: State -> Exp -> Maybe Int

esem s (Const i)    = Just i

esem s (Var v)      = look s v

esem s (Add e1 e2)  =
  case (esem s e1, esem s e2) of
    (Just i1, Just i2) -> Just (i1 + i2)
    (_, _)             -> Nothing

esem s (Mult e1 e2) =
  case (esem s e1, esem s e2) of
    (Just i1, Just i2) -> Just (i1 * i2)
    (_, _)             -> Nothing
```

# Semantics of Predicates

- Denotational Semantics of Predicates:
  - psem :: State -> Pred -> Maybe Bool
  - psem p e == Just True ====> "predicate p in state s is valid"
  - psem p e == Just False ====> "predicate p in state s is not valid"
  - psem p e == Nothing ====> "predicate p is not defined in state s"

# Semantics of Predicates in Haskell

```
psem :: State -> Pred -> Maybe Bool

psem s (Eq e1 e2) =
  case (esem s e1, esem s e2) of
    (Just i1, Just i2) -> Just (i1 == i2)
    (_, _)             -> Nothing

psem s (Less e1 e2) =
  case (esem s e1, esem s e2) of
    (Just i1, Just i2) -> Just (i1 < i2)
    (_, _)             -> Nothing
```

# Semantics of Formulae

- Denotational semantics of formulae
  - fsem :: State -> Form -> Maybe Int
  - fsem f e == Just True ====> "formula f in state s is valid"
    ====> "formula f describes state s"
  - fsem f e == Just False ====> "formula f in state s is not valid"
    ====> "formula f does not describe state s"
  - fsem f e == Nothing ====> "formula f is not defined in state s"

# Semantics of Formulae in Haskell

```
fsem :: State -> Form -> Maybe Bool
fsem s Tru        = Just True
fsem s Fal        = Just False
fsem s (Pred p) = psem s p

fsem s (And f1 f2) =
  case (fsem s f1, fsem s f2) of
    (Just b1, Just b2) -> Just (b1 && b2)
    (_, _) -> Nothing

fsem s (Or f1 f2) =
  case (fsem s f1, fsem s f2) of
    (Just b1, Just b2) -> Just (b1 || b2)
    (_, _) -> Nothing

fsem s (Not f) =
  case fsem s f of
    Just b -> Just (not b)
    _        -> Nothing
```

Tru describes all states s

Fal describes no states s

# What can we do with the semantics?

- We can determine which formulae are equivalent
  - Equivalent formulae describe the same set of states
  - f1 == f2 iff for all s, fsem s f1 == fsem s f2
- Question: Could you define a type class instance that implemented this notion of equality?
- Exercises. Prove the following using the Haskell definitions:
  - Tru == Not Fal
  - Fal == Not Tru
  - Not (Not f) == f
  - And f1 f2 == And f2 f1
  - Or f1 f2 == Or f2 f1
  - Or (Or f1 f2) f3 == Or f1 (Or f2 f3)
  - Not (And f1 f2) == Or (Not f1) (Not f2)

# What can we do with the semantics?

Lemma: Tru == Not Fal

Proof:

consider any s, we must prove: fsem s Tru = fsem s (Not Fal).

| | |
|---|---|
| fsem s Tru | |
| == Just True | (unfold fsem) |
| == Just (not False) | (fold not) |
| == fsem s (Not Fal) | (fold fsem s) |

# What can we do with the semantics?

- We can define the strength of a formula:
  - f1 is stronger than f2 if f1 describes a subset of the states described by f2. Alternatively, f2 is weaker than f1.
  - we write f1 => f2 iff

    for all s, fsem s f1 == Just True implies fsem s f2 == Just True

- Exercises. Prove the following using the Haskell definitions:
  - Fal => Tru
  - And f1 f2 => f1     (for any f1, f2)

# A bit of a glitch

- f1 => Or f1 f2 is not true in general.  Why?
- Recall: To prove a conjecture isn't true in general, give a counter-example.  Here's one:
  - Let f1 = Tru
  - Let f2 = Eq x x

  - Tru => Or Tru (Eq x x) iff

    for all s, fsem s Tru => fsem s (Or Tru (Eq x x))

  - consider s = [ ];  in this case:
    - fsem s Tru = Just True
    - fsem s (Or Tru (Eq x x)) = Nothing

# Resolving the glitch

- We assume there is some (finite) set of variables G that are allowed to appear in expressions, formulae and programs
  - An expression, formula, or program is well-formed if its variables are a subset of G
    - ie: the expression/formula/program only uses the allowed variables
  - A state s is well-formed if dom(s) is a superset or equal to G
    - ie: s defines all of the allowed variables
- New definitions. Consider any well-formed f1 and f2:
  - f1 == f2 iff for all well-formed s,

    fsem s f1 == fsem s f2
  - f1 => f2 iff for all well-formed s,

    fsem s f1 == Just True implies fsem s f2 == Just True

# Resolving the glitch

- From now on we will only work with well-formed objects
  - ie: we won't mention it, but you can assume every state, formula, etc., from here on out in these slides is well-formed

- From now on, formulae are either valid or invalid
  - ie: `fsem s f == Just True` or `fsem s f == Just False`
  - `fsem s f` is never `Nothing` when `s` and `f` are well-formed
  - hence, we can start ignoring the "Just" in the result
  - I'll often simply say "`fsem s f` is true" or "f is true" (in some state)
  - In this setting f1 => f2 is the classical notion of logical implication you are used to

# SUMMARY!

# Summary

- Hoare Triples characterize program properties
- States map variables to values
- Formulae describe states:
  - semantics in Haskell: fsem :: State -> Form -> Maybe Bool
  - formulae and states we deal with are well-formed
    - well-formedness is a very simple syntactic analysis
  - P => Q means P is stronger than Q; P describes fewer states