Reasoning About Imperative Programs

COS 441 Slides 10b

Agenda

- Last time
  - Hoare Logic:
    - { P } C { Q }
    - If P is true in the initial state s. And C in state s evaluates to s’. Then Q must be true in s’.
  - Program states s:
    - finite partial maps (ie: functions) from variables to values
  - Semantics of formulae:
    - fsem s P :: Maybe Bool
    - fsem s P is always Just b if s and P are “well-formed”
- This time:
  - Mathematical presentation of the semantics of formulae
  - Rules of Hoare Logic

SEMANTICS OF FORMULAE: PRESENTATION II: MATHEMATICS

Two Differences from the Haskell Presentation

- The Haskell definitions used datatype notation for the syntax of formulae:

```
data Exp = ...
    Add Exp Exp
  | Mult Exp Exp
```

```
esem s (Add e1 e2) = ... Just (... + ...)

esem s (Mult e1 e2) =  ... Just (... * ... )
```

Expression syntax (type Exp)
Defined using Haskell datatype

Integer operators (type Int)

- The standard math formulation overloads the same "*" and "+" symbols using them in different ways:

```
expressions
e ::= ... | e + e | e * e
```

```
esem (e1 + e2) = ... + ...

esem (e1 * e2) = ... * ...
```

Expression syntax (type "expression")

Integer (type "math integer")

Two Differences from the Haskell Presentation

- The Haskell semantic function explicitly creates “Maybe” objects all the time:

```
fsem :: State -> Form -> Maybe Bool

...

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
```

- This makes the Haskell function a total function but it is verbose and obscures the main idea:
  – the syntax “And f1 f2” is defined to be f1 && f2

Two Differences from the Haskell Presentation

- Whereas the Haskell function will be total, the math function will be partial -- it will be partial in all the places we would have used Nothing in Haskell
- We will use a standard mathematical convention *that when the result of a function contains undefined parts, the entire result is considered undefined*
- For example, if I were to write in math:

```
esem s (e1 * e2) = esem s (e1) * esem  s (e2)
```

- I mean "the semantics of e1 * e2 is (the semantics of e1) multiplied by (the semantics of e2), provided the semantics of e1 and e2 are both defined. If one of them is not defined, then the semantics of e1 * e2 is not defined either"
- That's exactly what the Haskell code says, which makes it more explicit, but a lot more long-winded

The Semantics Using Conventional Math Notation

Syntax

```
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
```

Semantics

```
[[ . ]] :: int exp -> state ->  int

[[N]]s          = N
[[x]]s          = s(x)
[[e1 + e2]]s = [[e1]]s + [[e2]]s
[[e1 * e2]]s = [[e1]]s + [[e2]]s

[[ . ]] :: predicate -> state -> bool

[[e1 = e2]]s = [[e1]]s == [[e2]]s
[[e1 < e2]]s = [[e1]]s < [[e2]]s

[[ . ]] :: formula -> state -> bool

[[true]]s       = true
[[false]]s     = false
[[p]]s          = [[p]]s
[[f1 & f2]]s  = [[f1]]s & [[f2]]s
[[f1 || f2]]s = [[f1]]s || [[f2]]s
[[~f]]s         = not [[f]]s
```

could be a partial function symbol: $\rightharpoonup$ but making those in powerpoint is irritating.

variable lookup in the environment (a finite partial map)

"[[ ]]" is merely an unusual name for a function

these definitions are incredibly elegant and compact.

experienced researchers can look at them and virtually instantaneously understand the meaning of the language or detect flaws in definition

Math vs. Haskell

- Summary: you should be able to understand and manipulate both kinds of notation.
- In particular, you should be able to take a mathematical definition and convert it in to Haskell program:

```
...

[[f1 & f2]]s  = [[f1]]s & [[f2]]s
[[f1 || f2]]s = [[f1]]s || [[f2]]s
```

$\longrightarrow$

```
...

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
```

ONE MORE BIT OF NOTATION: SUBSTITUTION

One Additional Bit of Notation

- Given an expression containing some variables, we often want to substitute some other expression for one of the variables
  - eg: below, we substitute the expression “2+3” for “x” in the expression “x * x”

```
let x = 2 + 3 in
x * x
```

$\longrightarrow$

```
(2 + 3) * (2 + 3)
```

One Additional Bit of Notation

- Given an expression containing some variables, we often want to substitute some other expression for one of the variables
  - eg: below, we substitute the expression “2+3” for “x” in the expression “x * x”

```
let x = 2 + 3 in
x * x
```

$$\longrightarrow$$

```
(2 + 3) * (2 + 3)
```

  - Another way to write the result is using substitution notation:

```
(x * x) [ 2 + 3 / x ]        ==        (2 + 3) * (2 + 3)
```

One Additional Bit of Notation

- Given an expression containing some variables, we often want to substitute some other expression for one of the variables
  - eg: below, we substitute the expression “2+3” for “x” in the expression “x * x”

```
let x = 2 + 3 in
x * x
```

$\longrightarrow$ (2 + 3) * (2 + 3)

  - Another way to write the result is using substitution notation:

(x * x) [2 + 3 / x]   ==   (2 + 3) * (2 + 3)

  - More generally, for any expressions e and e’, we write:

e [e’/x]

the name of the variable
to be replaced
(could be x or y or z ...)

contains x, possibly
several times or not at all

the expression replacing x

IMP:
A SIMPLE IMPERATIVE LANGUAGE

IMP

- Different languages have different sets of operations
  - the operations available in a language change the reasoning system quite a bit
  - that’s why reasoning about Haskell is so different from reasoning about C or Java!
  - if we added concurrency, it would be a whole new ball of wax ...

- We will look at IMP: the simplest possible language one can imagine that still embodies “imperative programming”

IMP Syntax

- IMP has three parts:
  - integer variables, integer expressions and statements
  - for simplicity, we've intentionally avoided having more than one type -- we are sticking with integers

```
integer variables
x := x1 | x2 | x3 | ... | y | z | ...

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

statements (aka Commands)
C ::= x = e                              (an assignment)
    | skip                               (a no-op)
    | C; C                               (sequencing)
    | if (e > 0) then C else C           (if statement)
    | while (e > 0) do C                 (while loop)
```

An Example Program ...

```
a = 0;
i = N;

while (i > 0) do
    a = a + V;
    i = i - 1
```

... With It's Specification

```
{ true }

a = 0;
i = N;

while (i > 0) do
    a = a + V;
    i = i - 1

{a = N*V}
```

A HOARE LOGIC FOR IMP

The Floyd-Hoare Rules for IMP

- We are looking for very general reasoning rules:
  - We want to figure out the reasoning rules once and for all and then be able to apply them to any program
  - If we figure the rules out once and for all, we can verify that the rules are correct -- any future Hoare proofs that use the rules exactly as stated are guaranteed to be correct

The Floyd-Hoare Rules for IMP

- We are looking for very general reasoning rules:
  - We want the rules to be sound:
    - if the rules allow us to come to a conclusion {P} C {Q}, the definition must hold:
    - Whenever we start in a state s such that [[P]]s, and execution of C leaves us in final state s' then [[Q]]s'
  - We would also like the rules to be complete:
    - if {P} C {Q} is a valid Hoare triple, it should be possible for us to conclude it using the rules supplied.
    - It turns out completeness is unobtainable, but that doesn't prevent us from verifying many programs

The Floyd-Hoare Rules for IMP

- Strategy for devising rules
  - For each simple imperative statement, we define a rule
  - For each compound statements, we define a rule
    - these rules typically use proofs about the underlying statements
  - Finally, we have a few "structural" rules that help us glue proof pieces together

NOT the Rule for Assignment

- Consider this rule. Is it good? How good?

{ true } x = e { x = e }

- Is it sound?
  - The precondition does not restrict the state we are allowed to start in since [[true]]s is always true.

NOT the Rule for Assignment

- Consider this rule. Is it good? How good?

{ true } x = e { x = e }

- Is it sound?
  - The precondition does not restrict the state we are allowed to start in since [[true]]s is always true.
  - What if e is x + 1? Specializing the rule:

{ true } x = x+1 { x = x+1 }

assignment statement that changes the value of x

an equation: for what values of x is x equal to itself plus 1?

NOT the Rule for Assignment

- Consider this rule. Is it good? How good?

{ true } x = e { x = e }

- Is it sound?
  - The precondition does not restrict the state we are allowed to start in since [[true]]s is always true.
  - What if e is x + 1? Specializing the rule:

{ true } x = x+1 { x = x+1 }

assignment statement that changes the value of x

an equation: for what values of x is x equal to itself plus 1?

there are no values of x that satisfy the equation! It doesn't matter what x we start with.

NOT the Rule for Assignment

- Consider this rule. Is it good? How good?

{ false } x = e { x = e }

- Is it sound?
  - The precondition restricts the state we are allowed to start in since [[false]]s is always false for any s.
  - Semantics of Hoare Triples: *If* we start in a state satisfying the precondition then … some other things need to hold
  - So it is *trivially* sound
- Is it complete?
  - No. Here's a simple triple we can't prove with the rule:

{ x = 9 } x = x + 1 { x = 10 }

  - The precondition false is just wayyyyy to strong

The Rule for Assignment

- Consider this rule. Is it good? How good?

$$\{ F [e/x] \}\ x = e\ \{ F \}$$

The Rule for Assignment

- Consider this rule. Is it good? How good?

$$\{ F [e/x] \}\ x = e\ \{ F \}$$

- An example:

$$\{ x + 1 = 10 \}\ x = x + 1\ \{ x = 10 \}$$

- Is it sound?
  - What initial states satisfy $x + 1 = 10$?

The Rule for Assignment

- Consider this rule. Is it good? How good?

  { F [e/x] } x = e { F }

- An example:

  { x + 1 = 10 } x = x + 1 { x = 10 }

- Is it sound?
  - What initial states satisfy x + 1 = 10?
    - [x = 9]
  - When we execute the assignment in that state, what do we get?
    - [x = 10]
  - It seems to work!

The Rule for Assignment

- Consider this rule. Is it good? How good?

$$\{ F [e/x] \} \; x = e \; \{ F \}$$

- Is it complete?
  - Are there valid triples that we cannot prove use this rule?
  - What about this one:

$$\{ x = 9 \} \; x = x + 1 \; \{ x = 10 \}$$

  - $(x = 10) [(x + 1) / x]$ is $x+1 = 10$
  - But $x+1 = 10$ is not syntactically equivalent to $x = 9$
  - However, $x = 9$ is semantically equivalent to $(x + 1) = 10$
- Summary: this assignment rule is pretty good but we need another rule for converting between semantically equivalent formulae and more ...

The Rule of Consequence

- Recall: A 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

- Intuition:
  - P’ => P: any state that satisfies P’ also satisfies P
    - P’ is “stronger” than P
  - Q => Q’: any state that satisfies Q also satisfies Q’
    - Q’ is “weaker” than Q

- The rule of consequence:

  If P’ => P and { P } C { Q } and Q => Q’

  then { P’ } C { Q’ }

The Rule of Consequence

- Rule of consequence:

  If P’ => P and { P } C { Q } and Q => Q’
  then { P’ } C { Q’ }

- Example:

{ x = 9 & y = 7 } x = x + 1 { x < 11 }

The Rule of Consequence

- Rule of consequence:

If P' => P and { P } C { Q } and Q => Q'
then { P' } C { Q' }

- Example:

(2) { x + 1 = 10 } x = x + 1 { x = 10 } (valid assignment rule)

{ x = 9 & y = 7 } x = x + 1 { x < 11 }

The Rule of Consequence

- Rule of consequence:

  If P' => P and { P } C { Q } and Q => Q'
  then { P' } C { Q' }

- Example:

(1) x = 9 & y = 7 => x + 1 = 10 (valid strengthening; more states satisfy x + 1 = 10)

(2) { x + 1 = 10 } x = x + 1 { x = 10 } (valid assignment rule)

{ x = 9 & y = 7 } x = x + 1 { x < 11 }

The Rule of Consequence

- Rule of consequence:

  If P’ => P and { P } C { Q } and Q => Q’
  then { P’ } C { Q’ }

- Example:

(1) x = 9 & y = 7 => x + 1 = 10	(valid strengthening; more states satisfy x + 1 = 10)
(2) { x + 1 = 10 } x = x + 1 { x = 10 }	(valid assignment rule)
(3) x = 10 => x < 11	(valid strengthening)
{ x = 9 & y = 7 } x = x + 1 { x < 11 }	(by (1), (2), (3), rule of consequence)

Compound Statements

- We have a rule for a single assignment, what about a sequence?
- Sequencing rule:

if { F1 } C1 { F2 } and { F2 } C2 { F3}

then { F1 } C1; C2 { F3 }

Compound Statements

- Example:

```
x = x + 1;

y = x – 3

x = y + y
```

Compound Statements

- Example:

```
x = x + 1;

y = x – 3

x = y + y
{ x = 17 & y < 23}
```

Compound Statements

- Example:

```
x = x + 1;

y = x – 3
{ y + y = 17  & y < 23 }
x = y + y
{ x = 17 & y < 23}
```

Compound Statements

- Example:

$x = x + 1;$

$\{ (x - 3) + (x - 3) = 17 \ \& \ x - 3 < 23 \}$

$y = x - 3$

$\{ y + y = 17 \ \& \ y < 23 \}$

$x = y + y$

$\{ x = 17 \ \& \ y < 23\}$

Compound Statements

- Example:

$\{ (x + 1) - 3 + (x + 1) - 3 = 17 \ \& \ (x + 1) - 3 < 23 \}$

$x = x + 1;$

$\{ (x - 3) + (x - 3) = 17 \ \& \ x - 3 < 23 \}$

$y = x - 3$

$\{ y + y = 17 \ \& \ y < 23 \}$

$x = y + y$

$\{ x = 17 \ \& \ y < 23 \}$

Compound Statements

- Example:

{ $2*x = 21$ & $x < 25$ }

{ $(x + 1) – 3 + (x + 1) – 3 = 17$ & $(x + 1) – 3 < 23$ }

$x = x + 1;$

{ $(x – 3) + (x – 3) = 17$ & $x – 3 < 23$ }

$y = x – 3$

{ $y + y = 17$ & $y < 23$ }

$x = y + y$

{ $x = 17$ & $y < 23$}

Skip

- Skip is a no-op "do nothing" statement
- Easy Hoare rule:

$$\{ P \}\ \text{skip}\ \{ P \}$$

- Intuition:
  - If you start with any state s that satisfies the precondition P, and you do nothing, you'll stay in the same state s and satisfy the postcondition P
  - And, of course, you can couple this rule with the rule of consequence. eg:

$$\{ x = 10 \}\ \text{skip}\ \{ x < 11 \}$$

If Statements

- Rule for if statements

  If { e > 0 & P } C1 { Q } and { ~(e > 0) & P } C2 { Q }

  then { P } if e > 0 then C1 else C2 { Q }

- Example:

```
{ true }

if x > 0 then
   skip;
else
   y = 1;

{ x > 0 || y = 1 }
```

If Statements

- Rule for if statements

  If { e > 0 & P } C1 { Q } and { ~(e > 0) & P } C2 { Q }

  then { P } if e > 0 then C1 else C2 { Q }

- Example:

```
{ true }

if x > 0 then
   skip;
else
   y = 1;

{ x > 0 || y = 1 }
```

assignment rule

```
{ x > 0 || y = 1 }
   skip;
{ x > 0 || y = 1 }
```

```
{ x > 0 || 1 = 1 }
   y = 1;
{ x > 0 || y = 1 }
```

If Statements

- Rule for if statements

  If { e > 0 & P } C1 { Q } and { ~(e > 0) & P } C2 { Q }

  then { P } if e > 0 then C1 else C2 { Q }

- Example:

```
{ true }

if x > 0 then
   skip;
else
   y = 1;

{ x > 0 || y = 1 }
```

```
{ x > 0  &  true }
   skip;
{ x > 0 || y = 1 }
```

```
{ x > 0 || y = 1 }
   skip;
{ x > 0 || y = 1 }
```

```
{ ~(x > 0) & true }
   y = 1;
{ x > 0 || y = 1 }
```

```
{ x > 0 || 1 = 1 }
   y = 1;
{ x > 0 || y = 1 }
```

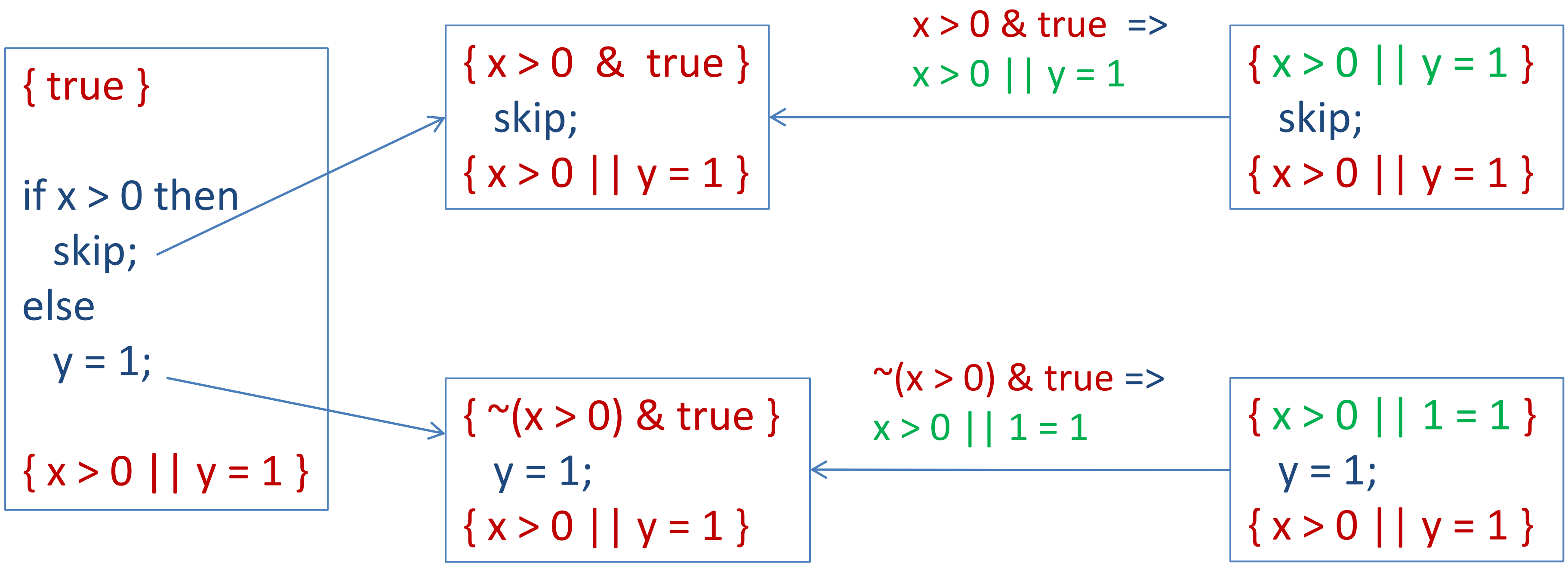
If Statements

- Rule for if statements

  If { e > 0 & P } C1 { Q } and { ~(e > 0) & P } C2 { Q }

  then { P } if e > 0 then C1 else C2 { Q }

- Example:



```
{ true }

if x > 0 then
  skip;
else
  y = 1;

{ x > 0 || y = 1 }
```

```
{ x > 0  &  true }
  skip;
{ x > 0 || y = 1 }
```

x > 0 & true =>
x > 0 || y = 1

```
{ x > 0 || y = 1 }
  skip;
{ x > 0 || y = 1 }
```

```
{ ~(x > 0) & true }
  y = 1;
{ x > 0 || y = 1 }
```

~(x > 0) & true =>
x > 0 || 1 = 1

```
{ x > 0 || 1 = 1 }
  y = 1;
{ x > 0 || y = 1 }
```

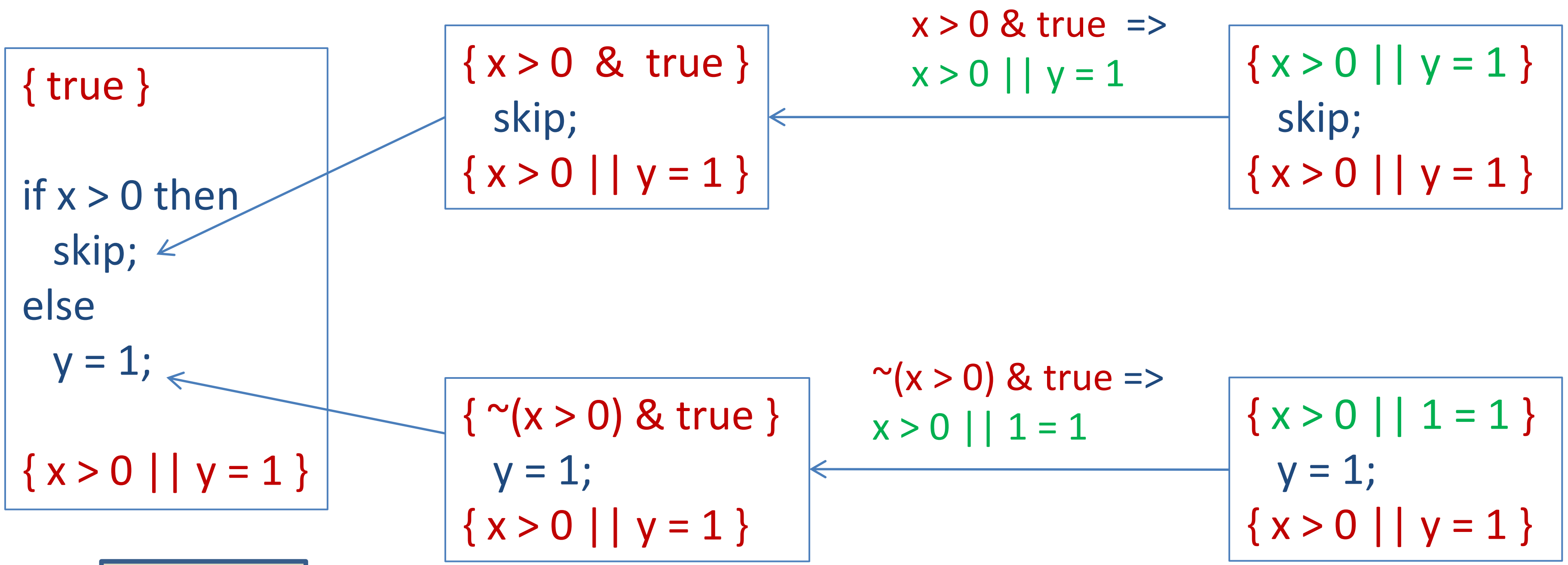
If Statements

- Rule for if statements

  If { e > 0 & P } C1 { Q } and { ~(e > 0) & P } C2 { Q }

  then { P } if e > 0 then C1 else C2 { Q }

- Example:



```
{ true }

if x > 0 then
  skip;
else
  y = 1;

{ x > 0 || y = 1 }
```

```
{ x > 0  &  true }
  skip;
{ x > 0 || y = 1 }
```

x > 0 & true =>
x > 0 || y = 1

```
{ x > 0 || y = 1 }
  skip;
{ x > 0 || y = 1 }
```

```
{ ~(x > 0) & true }
  y = 1;
{ x > 0 || y = 1 }
```

~(x > 0) & true =>
x > 0 || 1 = 1

```
{ x > 0 || 1 = 1 }
  y = 1;
{ x > 0 || y = 1 }
```

DONE!

While Statements

- Rule for while statements

If ???

then { P } while (e > 0) do C { Q }

While Statements

- Bogus rule for while statements

  If { P & e > 0 } C { Q }

  then { P } while (e > 0) do C { Q }

While Statements

- Bogus rule for while statements

  If { P & e > 0 } C { Q }

  then { P } while (e > 0) do C { Q }

basic problem:
this rule only
captures 1 iteration
of the loop,
not all of them

```
{ i = N & a = 0 }
while (i > 0) do
   a = a + K;
   i = i - 1;
{ a = N * K }
```

While Statements

- Bogus rule for while statements

  If { P & e > 0 } C { Q }

  then { P } while (e > 0) do C { Q }

```
{ i = N & a = 0 }
while (i > 0) do
    a = a + K;
    i = i - 1;
{ a = N * K }
```

must prove

```
{ i = N & a = 0 & i > 0 }
    a = a + K;
    i = i - 1;
{ a = N * K }
```

While Statements

- Bogus rule for while statements

  If { P & e > 0 } C { Q }

  then { P } while (e > 0) do C { Q }

```
{ i = N & a = 0 }
while (i > 0) do
    a = a + K;
    i = i - 1;
{ a = N * K }
```

must prove

```
{ i = N & a = 0 & i > 0 }
    a = a + K;
    i = i - 1;
{ a = N * K }
```

this isn't even
close to a valid triple!
With that precondition,
a = K at the end!

While Statements

- Problem: We need to verify all iterations of a loop and we need to do it with a finite amount of work
- Solution: We will come up with an invariant that holds at the beginning and end of all iterations.
  - We prove that the loop body preserves the invariant *every* time around
- Unfortunate reality: Inferring invariants automatically is undecideable.
  - This puts significant limits on the degree to which we can automate verification.

While Statements

- While rule:

loop invariant I

If P => I and { e > 0 & I } C { I } and I & ~(e > 0) => Q

then { P } while (e > 0) do C { Q }

While Statements

- While rule:

loop invariant I

If P => I and { e > 0 & I } C { I } and I & ~(e > 0) => Q

then { P } while (e > 0) do C { Q }

```
{ i = N & a = 0 }
while (i > 0) do
    a = a + K;
    i = i - 1;
{ a = N * K }
```

While Statements

- While rule:

loop invariant I

If P => I and { e > 0 & I } C { I } and I & ~(e > 0) => Q

then { P } while (e > 0) do C { Q }

```
{ i = N & a = 0 }
while (i > 0) do
    a = a + K;
    i = i - 1;
{ a = N * K }
```

```
{ i > 0 & I}
    a = a + K;
    i = i - 1;
{ I }
```

What works as I?
- true initially
- true before/after each iteration
- must imply Q when loop terminates

While Statements

- While rule:

loop invariant I

If P => I and { e > 0 & I } C { I } and I & ~(e > 0) => Q

then { P } while (e > 0) do C { Q }

```
{ i = N & a = 0 }
while (i > 0) do
    a = a + K;
    i = i - 1;
{ a = N * K }
```

```
{ i > 0 & I}
    a = a + K;
    i = i - 1;
{ I }
```

What works as I?

- true initially
- true before/after each iteration
- must imply Q when loop terminates

Invariant is:
a = (N-i) * K & i >= 0

While Statements

- Checking the invariant:
  - True initially:

invariant

$\boxed{i = N \,\&\, a = 0} \Rightarrow \boxed{a = (N-i) * K \,\&\, i >= 0}$

precondition

```
{ i = N & a = 0 }
while (i > 0) do
    a = a + K;
    i = i - 1;
{ a = N * K }
```

Invariant is:
a = (N-i) * K & i >= 0

While Statements

- Checking the invariant:
  - True initially:

    $\boxed{i = N \ \& \ a = 0} \Rightarrow \boxed{a = (N-i) * K \ \& \ i >= 0}$

  - True before/after each loop:

```
{ a = (N-i) * K & i >= 0 & i > 0 }

    a = a + K;

    i = i - 1;
{ a = (N-i) * K & i >= 0 }
```

```
{ i = N & a = 0 }
while (i > 0) do
    a = a + K;
    i = i - 1;
{ a = N * K }
```

Invariant is:
a = (N-i) * K & i >= 0

While Statements

- Checking the invariant:
  - True initially:

    i = N & a = 0 => a = (N-i) * K & i >= 0

  - True before/after each loop:

```
{ a = (N-i) * K & i >= 0 & i > 0 }

  a = a + K;
{ a = (N - (i - 1)) * K & (i - 1) >= 0 }
  i = i - 1;
{ a = (N-i) * K & i >= 0 }
```

```
{ i = N & a = 0 }
while (i > 0) do
   a = a + K;
   i = i - 1;
{ a = N * K }
```

Invariant is:
a = (N-i) * K & i >= 0

While Statements

- Checking the invariant:
  - True initially:

    $\boxed{i = N \ \&\ a = 0}$ => $\boxed{a = (N-i) * K \ \&\ i >= 0}$

  - True before/after each loop:

```
{ a = (N-i) * K & i >= 0 & i > 0 }
{ a + K = (N - (i - 1)) * K & (i - 1) >= 0 }
  a = a + K;
{ a = (N - (i - 1)) * K & (i - 1) >= 0 }
  i = i - 1;
{ a = (N-i) * K & i >= 0 }
```

```
{ i = N & a = 0 }
while (i > 0) do
    a = a + K;
    i = i - 1;
{ a = N * K }
```

Invariant is:
a = (N-i) * K & i >= 0

While Statements

- Checking the invariant:
  - True initially:

    $\boxed{i = N \,\&\, a = 0}$ => $\boxed{a = (N-i) * K \,\&\, i >= 0}$

  - True before/after each loop:

```
{ a = (N-i) * K & i >= 0 & i > 0 }
{ a + K = (N - (i - 1)) * K & (i - 1) >= 0 }
  a = a + K;
{ a = (N - (i - 1)) * K & (i - 1) >= 0 }
  i = i - 1;
{ a = (N-i) * K & i >= 0 }
```

  - Implies post-condition:

    $\boxed{a = (N-i) * K \,\&\, i >= 0}$ & ~(i > 0)
    => $\boxed{a = N * K}$

invariant

post condition

negation of while condition

```
{ i = N & a = 0 }
while (i > 0) do
    a = a + K;
    i = i - 1;
{ a = N * K }
```

Invariant is:
a = (N-i) * K & i >= 0

While Statements: Summary

- Given a Hoare triple for a while loop:
  - { P } while (e > 0) do C { Q }
- We prove it correct by:
  - guessing an invariant I (this is the hard part)
  - proving I holds initially: P => I
  - showing the loop body preserves I:
    - { e > 0 & I } C { I }
  - showing the postcondition holds on loop termination:
    - I & ~(e > 0) => Q
- As a rule:

  If P => I and { e > 0 & I } C { I } and I & ~(e > 0) => Q
  then { P } while (e > 0) do C { Q }

- Note: one often adds I as an annotation on the loop:
  - while [I] (e > 0) do C

FRAMING & MODULARITY

Another Issue: Framing

- Another valid triple:

$$\{ x = 9 \ \& \ y = 7 \ \& \ z = 23 \} \ x = x + 1 \ \{ x = 10 \ \& \ y = 7 \ \& \ z = 23 \}$$

- Proving it using the rules:

Another Issue: Framing

- Another valid triple:

{ x = 9 & y = 7 & z = 23} x = x + 1 { x = 10 & y = 7 & z = 23}

- Proving it using the rules:

(1) { x + 1 = 9 & y = 7 & z = 23 } x = x + 1 { x = 10 & y = 7 & z = 23} (valid assignment rule)

(2) x = 9 & y = 7 & z = 23 => x + 1 = 10 & y =7 & z = 23 (valid strengthening)

(3) { x = 9 & y = 7 & z = 23} x = x + 1 { x = 10 & y = 7 & z = 23} (by (1), (2), consequence)

Another Issue: Framing

- Another valid triple:

{ x = 9 & y = 7 & z = 23} x = x + 1 { x = 10 & y = 7 & z = 23}

- Proving it using the rules:

(1) { x + 1 = 9 & y = 7 & z = 23 } x = x + 1 { x = 10 & y = 7 & z = 23} (valid assignment rule)

(2) x = 9 & y = 7 & z = 23 => x + 1 = 10 & y =7 & z = 23 (valid strengthening)

(3) { x = 9 & y = 7 & z = 23} x = x + 1 { x = 10 & y = 7 & z = 23} (by (1), (2), consequence)

- Note: Formulae not involving x are just propagated
- More generally, formulae not involving variables that are not *modified* are just propagated
- Can we factor those expressions out of most of the proof?

The Simple Frame Rule

- The Simple Frame Rule (also called the rule of constancy)

  if { P } C { Q } and C does not modify the (free) variables of R
  then { P & R } C { Q & R }

- What counts as "modifying"?
  - In our simple language, the only way a variable may be modified is if it appears on the left in an assignment statement
  - In languages with functions or methods, calling one of them may have a modification effect
  - In C, you might be able to intentionally modify variables on the stack
  - In C, you might also have a buffer overflow ... yikes!
- The frame rule is a way of *simplifying* proofs
- Why are Haskell proofs so easy? Nothing is modified!

The Simple Frame Rule

- The Simple Frame Rule (also called the rule of constancy)

  if { P } C { Q } and C does not modify the (free) variables of R
  then { P & R } C { Q & R }

- Example:

{ x = 6 & y = 7 & z = 23} x = x + 1; x = x * 2; x = x - 4; { x = 10 & y = 7 & z = 23}

The Simple Frame Rule

- The Simple Frame Rule (also called the rule of constancy)

  if { P } C { Q } and C does not modify the (free) variables of R
  then { P & R } C { Q & R }

- Example:

{ x = 6 & y = 7 & z = 23} x = x + 1; x = x * 2; x = x - 4; { x = 10 & y = 7 & z = 23}

{ x = 9 } x = x + 1; x = x * x; x = x - 5; { x = 10 }

x = x + 1; x = x * 2; x = x - 4;
does not modify y or z

SUMMARY!

Summary

- States map variables to values
- Formulae describe states:
  - semantics in Haskell: fsem :: State -> Form -> Maybe Bool
  - semantics in Math: [[f]]s
  - 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
- Hoare Triples characterize program properties
  - { P } C { Q } – know when it is valid
  - know the statement rules you can use to conclude { P } C { Q }
  - understand the structural rules:
    - rule of consequence
    - frame rule