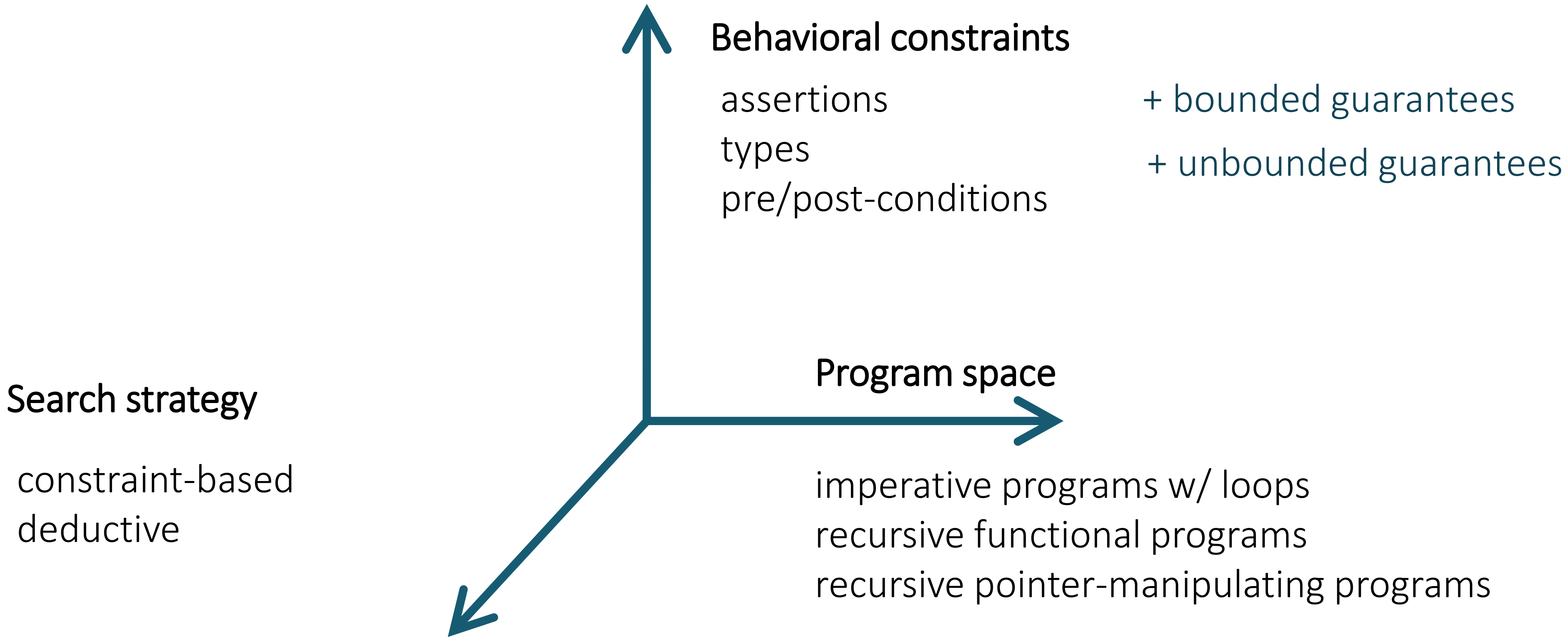
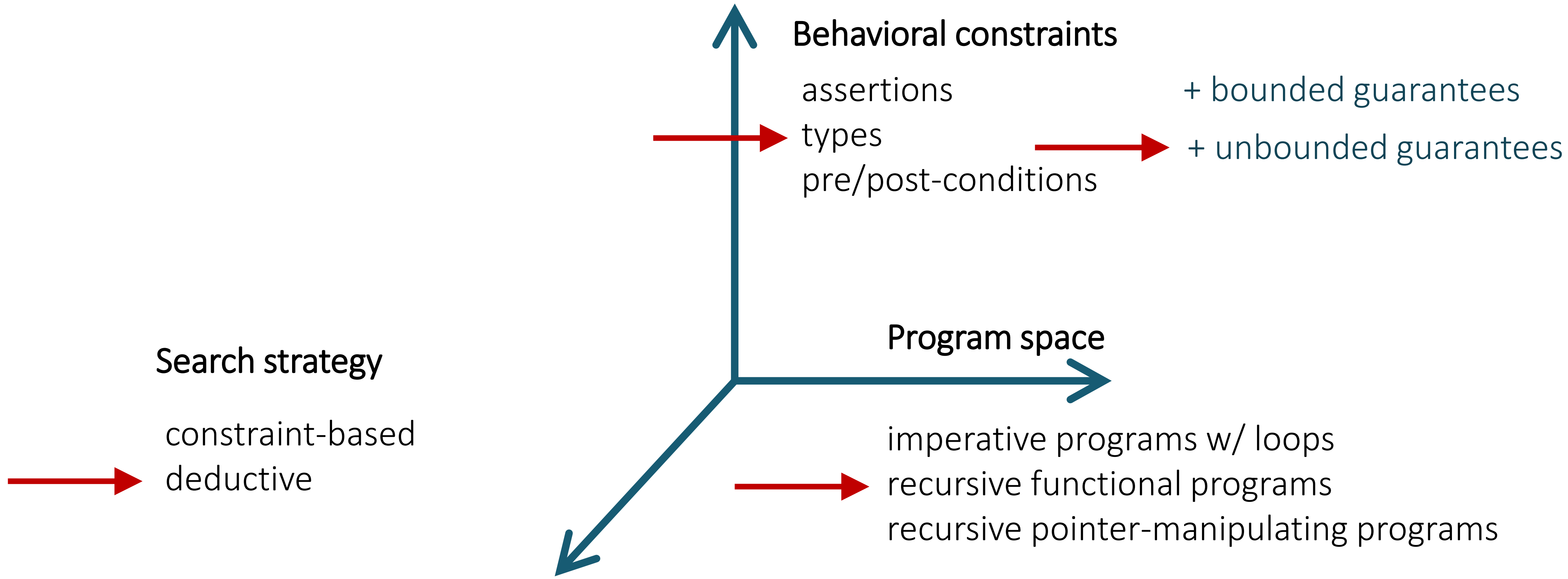
Lecture 12
Hoare Logic

Nadia Polikarpova
(some material from Peter Müller, ETH Zurich)

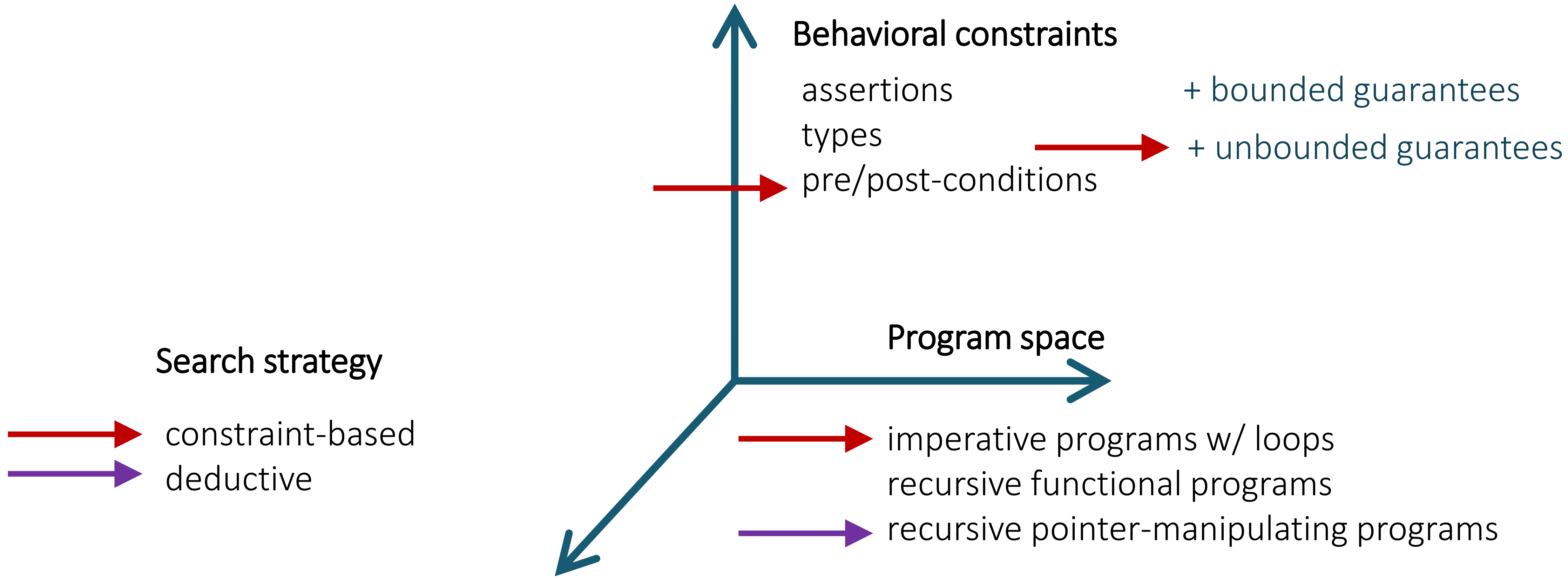
Module II


Behavioral constraints
assertions
types
pre/post-conditions
+ bounded guarantees
+ unbounded guarantees
Program space
imperative programs w/ loops
recursive functional programs
recursive pointer-manipulating programs
Search strategy
constraint-based
deductive

Last week


Behavioral constraints
assertions
types
pre/post-conditions
+ bounded guarantees
+ unbounded guarantees
Program space
imperative programs w/ loops
recursive functional programs
recursive pointer-manipulating programs
Search strategy
constraint-based
deductive

This week


Behavioral constraints
assertions
types
pre/post-conditions
+ bounded guarantees
+ unbounded guarantees
Program space
imperative programs w/ loops
recursive functional programs
recursive pointer-manipulating programs
Search strategy
constraint-based
deductive

Constraint-based synthesis

Behavioral constraints
= assertions, reference implementation, pre/post

Structural constraints

encoding $\longrightarrow$

$$\exists c\,.\,\forall x\,.\,Q(c,x)$$

Why is this hard?

```
Euclid (int a, int b) returns (int x)
  requires a > 0 ∧ b > 0
  ensures x = gcd(a, b)
{
  int x , y := a, b;
  while (x != y) {
    if (x > y) x := ??*x + ??*y + ??;
    else y := ??*x + ??*y + ??;
}}
```

infinitely many inputs

infinitely many paths!

Loop unrolling

```
Euclid (int a, int b) returns (int x)
  requires a > 0 ∧ b > 0
  ensures  x = gcd(a, b)
{
  int x , y := a, b;
  while (x != y) {
    if (x > y) x := ??*x + ??*y + ??;
    else y := ??*x + ??*y + ??;
}}
```

Unroll with depth = 1

```
if (x != y) {
  if (x > y)
    x := ??*x + ??*y + ??;
  else
    y := ??*x + ??*y + ??;
  assert !(x != y);
}
```

What's wrong with unrolling?

```
Euclid (int a, int b) returns (int x)
  requires a > 0 ∧ b > 0
  ensures  x = gcd(a, b)
{
  int x , y := a, b;
  while (x != y) {
    if (x > y) x := ??*x + ??*y + ??;
    else y := ??*x + ??*y + ??;
}}
```

Unroll with depth = 1

```
if (x != y) {
  if (x > y)
    x := ??*x + ??*y + ??;
  else
    y := ??*x + ??*y + ??;
  assert !(x != y);
}
```

Unsatisfiable sketch

What's wrong with unrolling?

What if we restrict inputs to [1, 2]?

```
Euclid (int a, int b) returns (int x)
  requires a > 0 ∧ b > 0
  ensures  x = gcd(a, b)
{
  int x , y := a, b;
  while (x != y) {
    if (x > y) x := ??*x + ??*y + ??;
    else y := ??*x + ??*y + ??;
}}
```

Unroll with depth = 1

```
if (x != y) {
  if (x > y)
    x := 0 *x + 0 *y + 1 ;
  else
    y := 0 *x + 0 *y + 1 ;
  assert !(x != y);
}
```

Unsound solution!

Constraint-based synthesis

Behavioral constraints
= assertions, reference implementation, pre/post

Structural constraints

encoding

$$\exists c\,.\,\forall x\,.\,Q(c,x)$$

If we want to synthesize programs that are correct on all inputs,
we need a better way to deal with loops!

Solution

Hoare logic = a program logic for simple imperative programs

- in particular: loop invariants

The Imp language

```
e ::= n | x |
      e + e | e - e | e * e |
      e = e | e < e | !e | e && e
c ::= skip                   |
      x := e                 |
      c ; c                  |
      if e then c else c     |
      while e do c
```

Hoare triples

Properties of programs are specified as judgments

$$\{P\}\ c\ \{Q\}$$

where c is a command and $P, Q: \sigma \to Bool$ are predicates

- e.g. if $\sigma = [x \mapsto 2]$ and $P \equiv x > 0$ then $P\ \sigma = \mathrm{T}$

Terminology

- Judgments of this kind are called *(Hoare) triples*
- P is called precondition
- Q is called postcondition

Meaning of triples

The meaning of $\{P\}\ c\ \{Q\}$ is:

- **if** P holds in the initial state σ, and
- **if** the execution of c from σ terminates in a state σ'
- **then** Q holds in σ'

This interpretation is called *partial correctness*

- termination is not essential

Another possible interpretation: *total correctness*

- **if** P holds in the initial state σ
- **then** the execution of c from σ terminates in a state (call it σ')
- **and** Q holds in σ'

Example: swap

$\{\top\}$

```
x := x + y; y := x - y; x := x - y
```

$\cancel{\{x = y \wedge y = x\}}$

We have to express that y in the final state is equal to x in the initial state!

Logical variables

$\{x = N \land y = M\}$

```
x := x + y; y := x - y; x := x - y
```

$\{x = M \land y = N\}$

Assertions can contain *logical variables*

- may occur only in pre- and postconditions, not in programs
- the state maps logical variables to their values, just like normal variables

Inference system

We formalize the semantics of a language by describing which judgments are valid about a program

An *inference system*

- a set of *axioms* and *inference rules* that describe how to derive a valid judgment

We combine axioms and inference rules to build *inference trees* (derivations)

Semantics of skip

`skip` does not modify the state

$$\{\, P \,\}\ \texttt{skip}\ \{\, P \,\}$$

Semantics of assignment

$\{x > 0\}$ `x := x + 1` $\{???\}$

$\{???\}$ `x := x + 1` $\{x > 1\}$

Semantics of assignment

`x := e` assigns the value of `e` to variable `x`

$$\{\, P[x \mapsto e] \,\}\ \ \texttt{x} \coloneqq \texttt{e}\ \ \{\, P \,\}$$

- Let σ be the initial state
- Precondition: $(P[x \mapsto e])\sigma$, i.e., $P(\sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma])$
- Final state: $\sigma' = \sigma[x \mapsto \mathcal{A}[\![e]\!]\sigma]$
- Consequently, P holds in the final state

Semantics of composition

Sequential composition `c1 ; c2` executes **`c1`** to produce an intermediate state and from there executes **`c2`**

$$\frac{\{P\}\ c_1\ \{R\} \qquad \{R\}\ c_2\ \{Q\}}{\{P\}\ c_1; c_2\ \{Q\}}$$

Example: swap

$$\dfrac{}{\{\, P[x \mapsto e] \,\}\ \texttt{x := e}\ \{\, P \,\}}$$

inference tree

leaves = axioms

$$\dfrac{}{\{x = N \land y = M\}\quad \texttt{x := x + y}\quad \{y = M \land x - y = N\}}\ \text{assign}$$

$$\dfrac{}{\{y = M \land x - y = N\}\ \texttt{y := x - y}\ \{x - y = M \land y = N\}}\ \text{assign}$$

edges = rules

$$\dfrac{}{\{x = N \land y = M\}\ \texttt{x := x + y; y := x - y}\ \{x - y = M \land y = N\}}\ \text{comp}$$

$$\dfrac{}{\{x - y = M \land y = N\}\ \texttt{x := x - y}\ \{x = M \land y = N\}}\ \text{assign}$$

$$\dfrac{}{\{x = N \land y = M\}\ \texttt{x := x + y; y := x - y; x := x - y}\ \{x = M \land y = N\}}\ \text{comp}$$

root = triple to prove

Proof outline

$\{ P[x \mapsto e] \}$ `x := e` $\{ P \}$

An alternative (more compact) representation of inference trees

$\{x = N \land y = M\}$

$\Rightarrow$

$\{(x + y) - ((x + y) - y) = M \land (x + y) - y = N\}$

```
x = x + y;
```

$\{x - (x - y) = M \land x - y = N\}$

```
y = x - y;
```

$\{x - y = M \land y = N\}$

```
x = x - y
```

$\{x = M \land y = N\}$

Rule of consequence

$$\frac{\{P'\}\ c\ \{Q'\}}{\{P\}\ c\ \{Q\}}\ \text{ if }\ P \Rightarrow P' \land Q' \Rightarrow Q$$

Corresponds to adding $\Rightarrow$ steps in a proof outline

Here $P \Rightarrow P'$ should be read as

- "We can prove for all states σ, that $P\ \sigma$ implies $P'\ \sigma$"

Semantics of conditionals

$$\frac{\{P \land e\}\ c_1\ \{Q\} \qquad \{P \land \neg e\}\ c_2\ \{Q\}}{\{P\}\ \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{Q\}}$$

Example: absolute value

$\{\top\}$

```
if x < 0 then
```

$$\begin{array}{l} \{\top \land x < 0\} \\ \Rightarrow \\ \{-x \geq 0\} \end{array}$$

```
    x := -x
```

$\{x \geq 0\}$

```
else
```

$$\begin{array}{l} \{\neg(x < 0)\} \\ \Rightarrow \\ \{x \geq 0\} \end{array}$$

```
    skip
```

$\{x \geq 0\}$

$\{x \geq 0\}$

$$\frac{\{P \land e\}\ c_1\ \{Q\} \qquad \{P \land \neg e\}\ c_2\ \{Q\}}{\{P\}\ \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{Q\}}$$

Semantics of loops

$$\frac{\{?\}\ c\ \{?\}}{\{P\}\ \texttt{while}\ e\ \texttt{do}\ c\ \{Q\}}$$

Challenge: *c* needs to execute multiple times with the same pre/post

Semantics of loops

loop invariant

$$\frac{\{I\}\ c\ \{I\}}{\{I\}\ \texttt{while}\ e\ \texttt{do}\ c\ \{I\}}$$

Challenge: *c* needs to execute multiple times with the same pre/post

Solution: make its pre and post *the same*!

- called a *loop invariant*

Semantics of loops

$$\frac{\{I \land e\}\ c\ \{I\}}{\{I\}\ \texttt{while}\ e\ \texttt{do}\ c\ \{\neg e \land I\}}$$

Challenge: *c* needs to execute multiple times with the same pre/post

Solution: make its pre and post *the same*!

- called a *loop invariant*
- + strengthen the semantics with the info about the loop condition

Example: GCD

$\{x = N \land y = M \land N > 0 \land M > 0\}$

$\Rightarrow$

$\{I\}$

```
while x != y do
  {I ∧ x ≠ y}
  if x > y then
    x := x - y
  else
    y := y - x
  {I}
```

$\{I \land x = y\}$

$\Rightarrow$

$\{x = \gcd(N, M)\}$

Guessing the loop invariant:

x	y	N	M
10	4	10	4
6	4	10	4
2	4	10	4
2	2	10	4

$$I \equiv \gcd(x, y) = \gcd(N, M)$$

Example: GCD

$\{x = N \wedge y = M \wedge N > 0 \ \wedge M > 0\}$

$\Rightarrow$

$\{\mathrm{gcd}(x, y) = \mathrm{gcd}(N, M) \wedge x, y > 0\}$

```
while x != y do
```

$\{\mathrm{gcd}(x, y) = \mathrm{gcd}(N, M) \wedge x, y > 0 \wedge \ x \neq y\}$

```
if x > y then
```

$\{\mathrm{gcd}(x, y) = \mathrm{gcd}(N, M) \wedge \ x \neq y \wedge x > y\}$

$\Rightarrow$

$\{\mathrm{gcd}(x - y, y) = \mathrm{gcd}(N, M) \wedge x - y, y > 0\}$

```
x := x - y
```

$\{\mathrm{gcd}(x, y) = \mathrm{gcd}(N, M) \wedge x, y > 0\}$

```
else
    y := y - x
```

$\{\mathrm{gcd}(x, y) = \mathrm{gcd}(N, M) \wedge x, y > 0\}$

$\{\mathrm{gcd}(x, y) = \mathrm{gcd}(N, M) \wedge x, y > 0 \wedge \ x = y\}$

$\Rightarrow$

$\{x = \mathrm{gcd}(N, M)\}$

Termination

loop variant / ranking function /
termination metric

$$\frac{\{I \land e \land r = R\}\ c\ \{I \land r < R \land r \geq 0\}}{\{I\}\ \texttt{while}\ e\ \texttt{do}\ c\ \{\neg e \land I\}}$$

Example: GCD

```
while x != y do

    if x > y then
        x := x - y
    else
        y := y - x
```

Example: GCD

$\{x = N \land y = M \land N > 0 \ \land M > 0\}$

$\Rightarrow$

$\{\mathrm{gcd}(x, y) = \mathrm{gcd}(N, M) \land x, y > 0\}$

```
while x != y do
```

$\{\mathrm{gcd}(x, y) = \mathrm{gcd}(N, M) \land x, y > 0 \land x + y = R \land \ x \neq y\}$

```
if x > y then
    x := x - y
else
    y := y - x
```

$\{\mathrm{gcd}(x, y) = \mathrm{gcd}(N, M) \land x, y > 0 \land x + y < R \land x + y \geq 0\}$

$\{\mathrm{gcd}(x, y) = \mathrm{gcd}(N, M) \land x, y > 0 \land \ x = y\}$

$\Rightarrow$

$\{x = \mathrm{gcd}(N, M)\}$

Program Verification

```
method Euclid (a: int, b: int) returns (gcd: int)
  requires a > 0 && b > 0
  ensures x == gcd(a,b)
{
  var x, y := a, b;
  while (x != y)
    invariant y > 0 && x > 0 && gcd(x,y) == gcd(a,b)
    decreases x + y
  {
    if (x > y) {
      x := x - y;
    } else {
      y := y - x;
    }
  }
}
```

Dafny

correct!

can't proof correctness

Program synthesis

```
method Euclid (a: int, b: int) returns (gcd: int)
  requires a > 0 && b > 0
  ensures x == gcd(a,b)
{
  var x, y := ?;
  ?;
  while (?)
    invariant ?
    decreases ?
  {
   ?;
  }
  ?;
}
```

found a correct program!

```
var x, y := a, b;
while (x != y)
  invariant y > 0 && x > 0 && gcd(x,y) == gcd(a,b)
  decreases x + y
{
  if (x > y) {
    x := x - y;
  } else {
    y := y - x;
  }
}
```

can’t find a (program, invariant) pair that I can prove correct

Verification → synthesis

Srivastava, Gulwani, Foster: From program verification to program synthesis. POPL'10

- idea: make constraint-based synthesis unbounded by synthesizing loop invariants alongside programs
- synthesized some looping programs with integers, including Bresenheim algorithm
- won "Most Influential Paper" at POPL'20!

Qiu, Solar-Lezama: Natural Synthesis of Provably-Correct Data-Structure Manipulations. OOPSLA'17

- same approach for pointer-manipulating programs

Verification → synthesis

Verification

Program logic

verification condition $\forall x \,.\, Q(x)$

SMT

Synthesis

Program logic

$\exists I\ P.\forall x \,.\, Q(I, P, x)$

synthesis constraint

unknown formulas for invariants and commands

synthesizer for loop-free programs

program + invariant

no solution

How verification works

Verification

$$\forall x \,.\, Q(x)$$

Step 1: eliminate loops

$\{pre\}$

```
init;
while (c)
  invariant inv
{ body; }
final;
```

$\{post\}$

$\{pre\}$

```
init;
```

$\{inv\}$

$\{inv \land cond\}$

```
body;
```

$\{inv\}$

$\{inv \land \neg(cond)\}$

```
final;
```

$\{post\}$

Step 2: generate VCs

$\{pre\}$
```
init
while (c)
  invariant inv;
{ body }
final
```
$\{post\}$

$\{pre\}$ `init` $\{inv\}$ → $\{pre\}\ init(x,x')\ \{inv[x \mapsto x']\}$

$\{inv \land c\}$ `body` $\{inv\}$ → $\{inv \land c\}\ body(x,x')\ \{inv[x \mapsto x']\}$

$\{inv \land \neg c\}$ `final` $\{post\}$ → $\{inv \land \neg c\}\ final(x,x')\ \{post\}$

$$\underbrace{\forall x\,.\ pre \land init \Rightarrow inv' \ \land\ inv \land c \land body \Rightarrow inv' \ \land\ inv \land \neg c \land final \Rightarrow post}_{\text{verification condition}}$$

$\forall x\,.\,Q(x)$

From verification to synthesis

Verification

$$\forall x\,.\,Q(x)$$
$$=$$
$$\exists x\,.\,\neg Q(x)$$

$\xrightarrow{\text{SMT}}$

UNSAT / SAT

Synthesis

$$\exists I\ P.\forall x\,.\,Q(I, P, x)$$

Program synthesis

$\{pre\}$

```
??
while (??)
  invariant ??;
{ ?? }
??
```

$\{post\}$

$\{pre\}$
$S_i(x, x')$
$\{I[x \mapsto x']\}$

$\{I \wedge G\}$
$S_b(x, x')$
$\{I[x \mapsto x']\}$

$\{I \wedge \neg G_0\}$
$S_f(x, x')$
$\{post\}$

$$\exists S\ G\ I . \forall x .$$

$$pre \wedge S_i \Rightarrow I'$$

$$\wedge$$

$$I \wedge G \wedge S_b \Rightarrow I'$$

$$\wedge$$

$$\underbrace{I \wedge \neg G \wedge S_f \Rightarrow post}_{\text{synthesis constraint}}$$

$$\exists I\ P . \forall x . Q(I, P, x)$$

Synthesis constraints

$$pre \wedge S_i \Rightarrow I'$$

$$I \wedge G \wedge S_b \Rightarrow I'$$

$$I \wedge \neg G \wedge S_f \Rightarrow post$$

Domain for I, G: formulas over program variables

Domain for $S = \{x' = e_x \wedge y' = e_y \wedge \cdots \mid e_x, e_y, \ldots \in Expr\}$

- conjunction of equalities, one per variables

Solving synthesis constraints

$$pre \wedge S_i \Rightarrow I'$$

$$I \wedge G \wedge S_b \Rightarrow I'$$

$$I \wedge \neg G \wedge S_f \Rightarrow post$$

Can be solved this with...

- SyGuS solvers
- Sketch
  - Look we made an unbounded synthesizer out of Sketch!
- VS3 uses Lattice search
  - More efficient for predicates