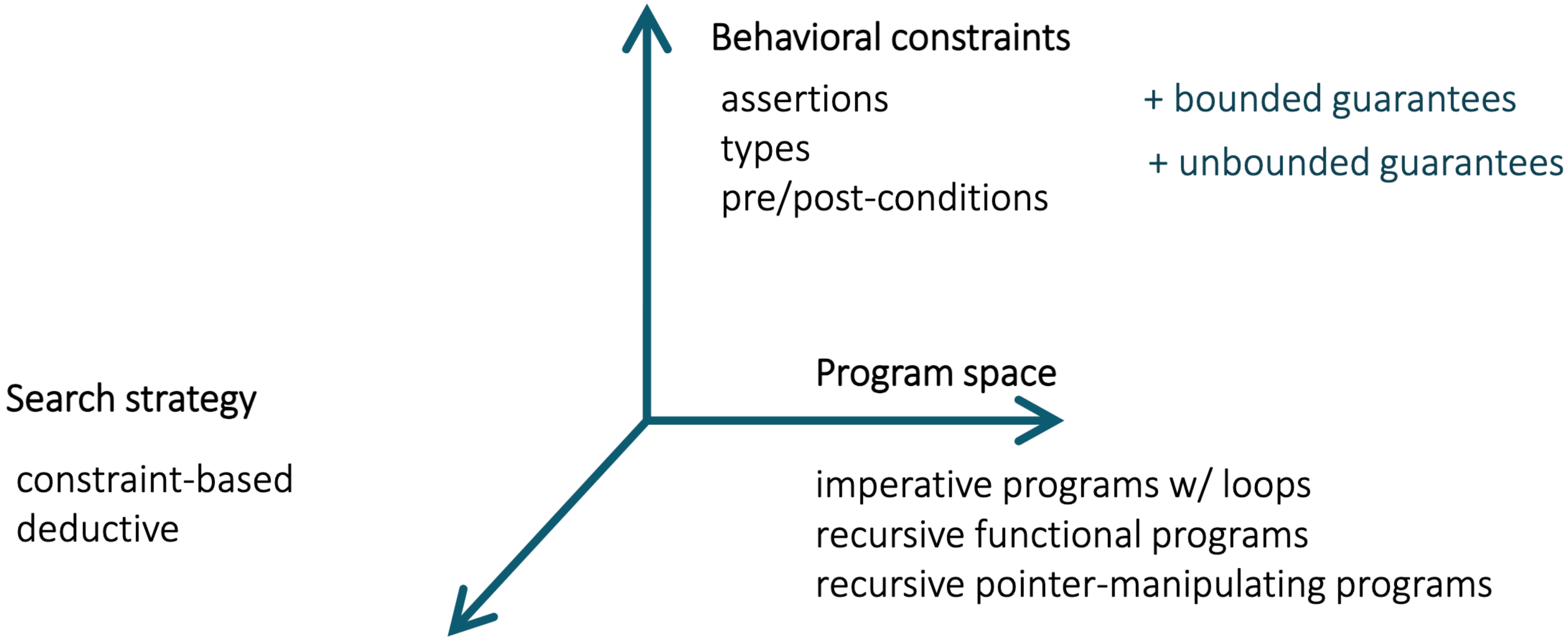
CS5733 Program Synthesis

#17. Hoare Logic and Synthesis

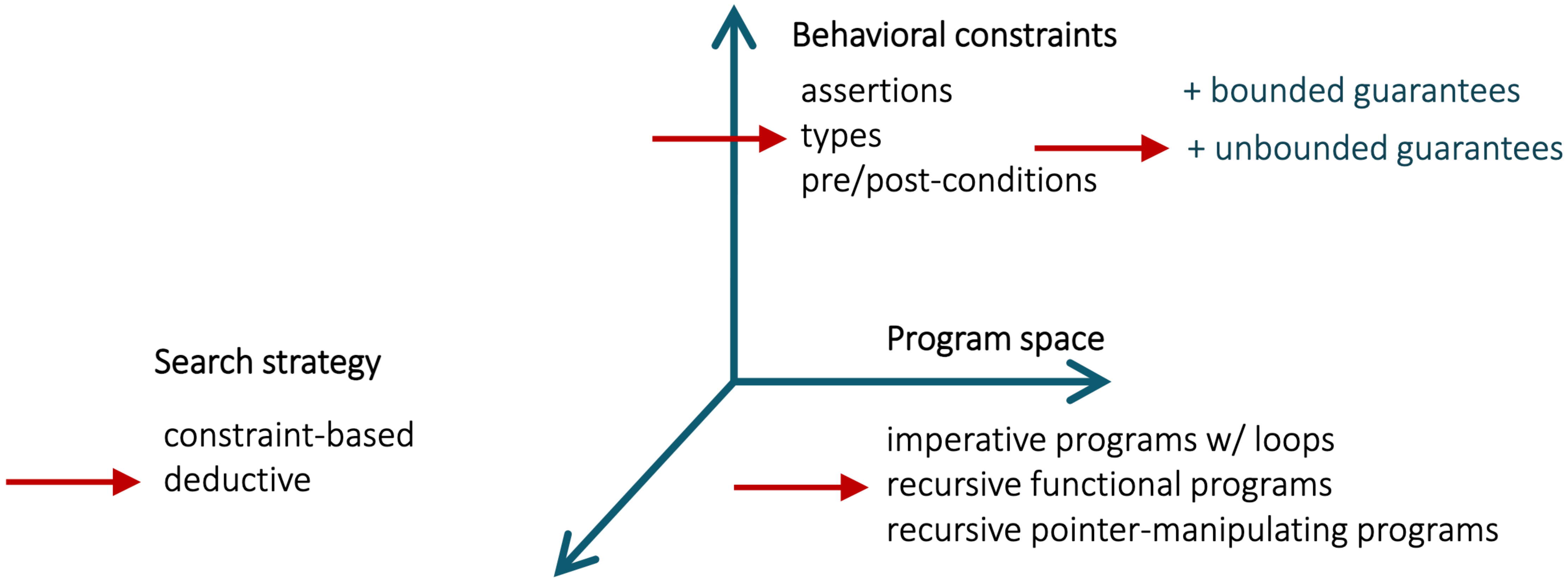
Ashish Mishra, September 24, 2024

With slides from Nadia Polikarpova and Yu-Fang Chen

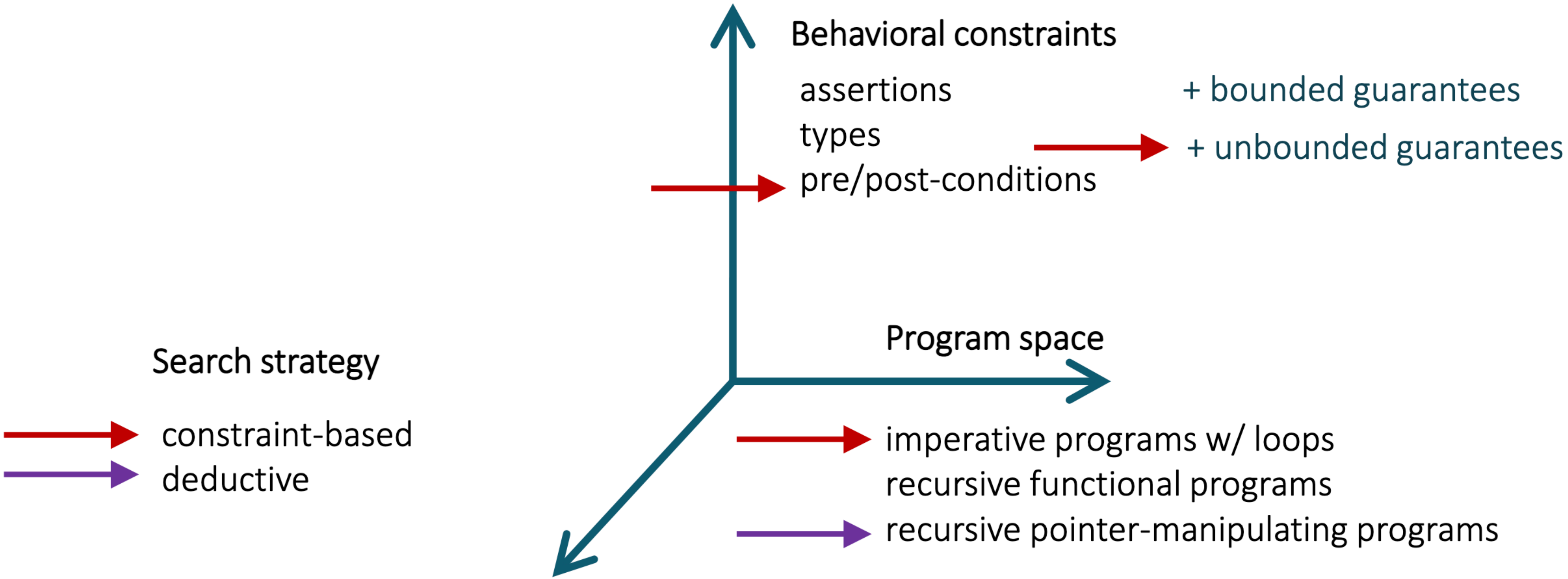
Module II


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

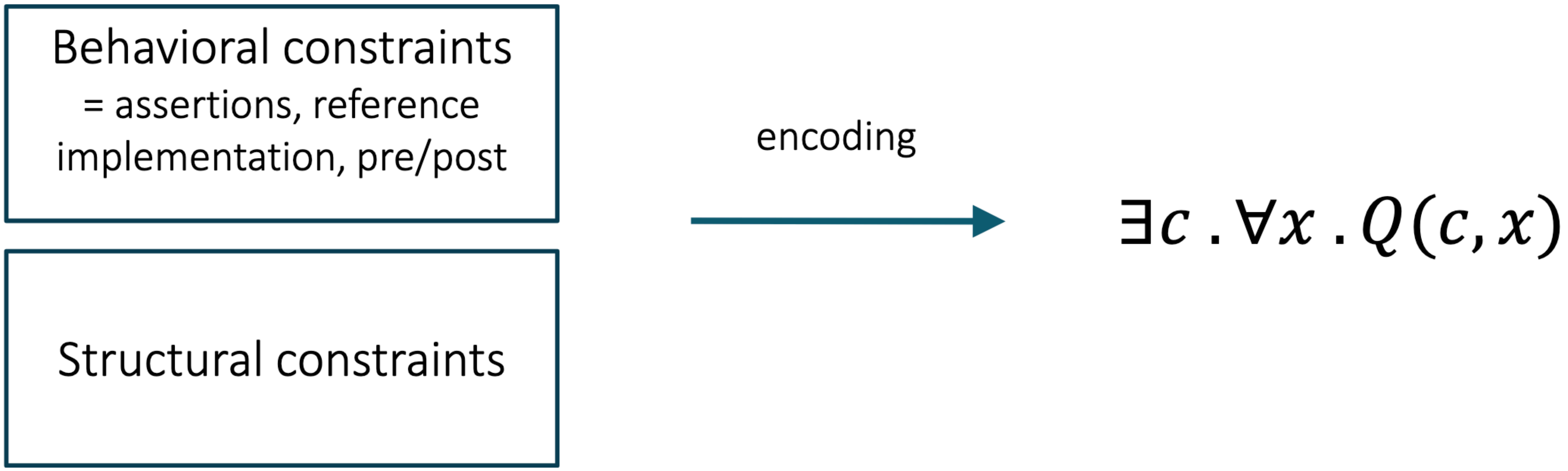
Last week


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



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 is unsound and incomplete

```
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);
}
```

Loop unrolling is unsound and incomplete

```
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

Loop unrolling is unsound and incomplete

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

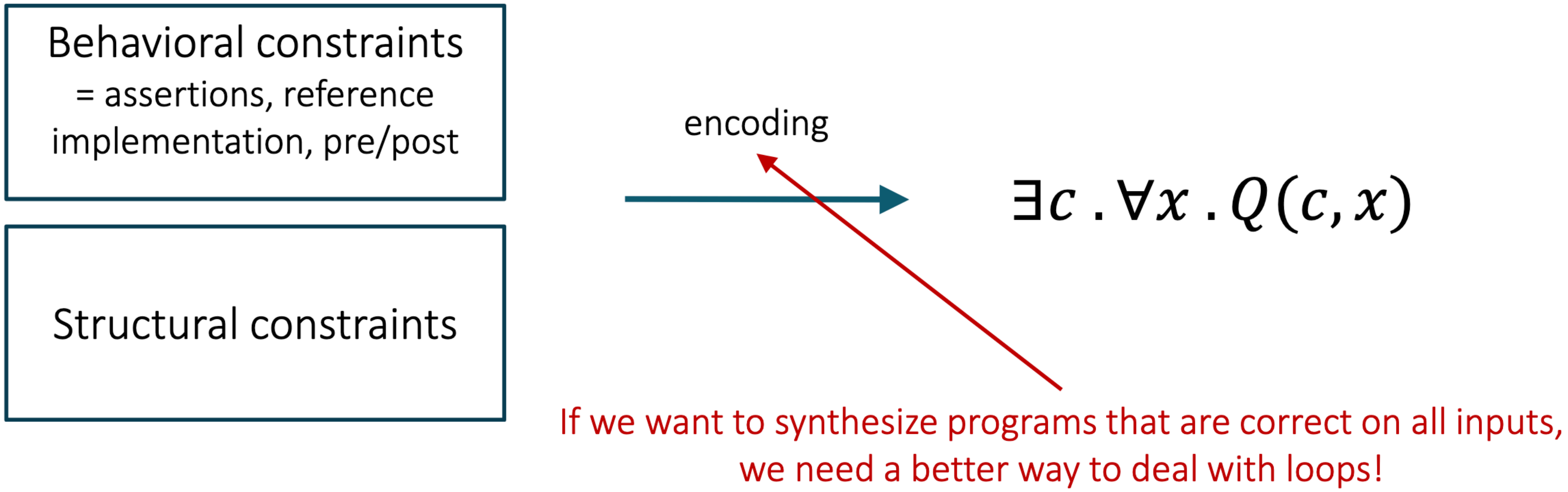
```
Euclid (int a, int b) returns (int x)
  requires a > 0 ∧ b > 0
  ensures  x = gcd(a, b)
{
  int x , y := a, b;
  while (x != y) {
    if (x > y) x := ??*x + ??*y + ??;
    else y := ??*x + ??*y + ??;
}}
```

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



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 $P, Q: \sigma \rightarrow Bool$ 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

$\{T\}$

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

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

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

Logical Variables

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

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

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

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

- Similar to the Logical System in PL and FOL.
- Called as the Hoare Logic

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

$x > 1$

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

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

Semantics of Assignment

We begin with Foyld's version of the assignment axiom

{P} **X := E** {?}

The term **E** might contain **X**, e.g. **E** $\equiv$ **X+1**

An example: **X := X + 1**

The value of X **after** executing the statement

The value of X **before** executing the statement

We need to differentiate these two values!

Floyd's version

We begin with Foyld's version of the assignment axiom

{P} X := E {?}

$$\exists V.(\ X=E[V/X]\ \wedge\ P[V/X]\ )$$

Intuition: we use new variable V to denote the **old value of X**

Notations	
E[V/X] **P[V/X]**	replacing all **free occurrences** of **X** in **E** / **P** with **V**

Flyod's version

Foyld's Assignment Axiom

$$\frac{}{\{P\}\ X{:=}E\ \{\exists\ V.\ X{=}E[V/X] \wedge P[V/X]\}}$$

Example

$\{Y + X = 42\}\ X := X + 5\ \{\exists\ V.\ X = V + 5 \wedge Y + V = 42\}$

Example

$\{Y = 5\}\ X := X/Y + X\ \{?\}$

We do not want to have quantifiers in the reasoning path!

Hoare's backward semantics of assignment

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

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

- 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

Hoare's backward semantics

Backward reasoning

Hoare's Assignment Axiom

$$\frac{}{\{Q[E/X]\}\ X{:=}E\ \{Q\}}$$

Read as If Q holds in the post-condition then …

Let s be the state before X := E and s′ the state after.
So, s′ = s[X → E] (assuming E has no side-effect).

Q[E/X] holds in s if and only if Q holds in s′, because
(1) Every variable, except X, has the same value in s and s′, and
(2) Q[E/X] has every X in Q replaced by E,
(3) Q has every X evaluated to E in s (s' = s[X → E]).

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

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

leaves = axioms

inference tree

$$\frac{}{\{x = N \wedge y = M\}\ \texttt{x := x + y}\ \{y = M \wedge x - y = N\}}\ \text{assign}$$

$$\frac{}{\{y = M \wedge x - y = N\}\ \texttt{y := x - y}\ \{x - y = M \wedge y = N\}}\ \text{assign}$$

edges = rules

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

$$\frac{}{\{x - y = M \wedge y = N\}\ \texttt{x := x - y}\ \{x = M \wedge y = N\}}\ \text{assign}$$

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

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
```

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

Try out example

Example
P: $\{true\}\ X:=2\ ;\ Y:=X\ \{X > 0 \wedge Y=2\}$

(1) $2>0 \wedge 2 = 2 \Leftrightarrow true$ (Integer arithmetic)
(2) $\{2 > 0 \wedge 2 = 2\}\ X:=2\ \{X > 0 \wedge X = 2\}$ (assignment axiom)
(3) $\{X > 0 \wedge X = 2\}\ Y:=X\ \{X > 0 \wedge Y = 2\}$ (assignment axiom)
(4) $\{true\}\ X:=2\ \{X > 0 \wedge X = 2\}$ (by (1), we can replace $2>0 \wedge 2 = 2$ in (3) with true)
(5) $\{true\}\ X:=2\ ;\ Y:=X\ \{X > 0 \wedge Y=2\}$ (by (3), (4), and composition rule)

Rule of consequence

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

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 $P'\ \sigma$"

Consequence rule

Consequence Rule

$$\frac{P \Rightarrow P' \quad \{P'\}\ S\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ S\ \{Q\}}$$

- We can strengthen the precondition, i.e. assume more than we need
- We can weaken the postcondition, i.e. conclude less than we are allowed to

Consequence rule

Consequence Rule

$$\frac{P \Rightarrow P' \;\; \{P'\}\; S\; \{Q'\} \;\; Q' \Rightarrow Q}{\{P\}\; S\; \{Q\}}$$

Example

P_1: {true ∧ X < 10} X:=10 {X=10 ∨ X=0}

(1) {true} X:=10 {X=10 ∨ X=0} (by Assignment Rule)
(2) true∧X<10 ⇒ true (by underlying logic)
(3) X = 10 ∨ X = 0 ⇒ X = 10 ∨ X = 0 (by underlying logic)
(4) {true ∧ X < 10} X:=10 {X=10 ∨ X=0} (by consequence rule, (2), and (3))

Consequence rule

Consequence Rule

$$\frac{P \Rightarrow P' \quad \{P'\}\ S\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ S\ \{Q\}}$$

Example

P_2: $\{true \wedge X \geq 10\}\ X{:=}0\ \{X{=}10 \vee X{=}0\}$

Try it yourself!

Semantics of conditionals

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

Example: absolute value

$\{\top\}$

```
if x < 0 then
```

$\{\top \land x < 0\}$

$\Rightarrow$

$\{-x \geq 0\}$

```
    x := -x
```

$\{x \geq 0\}$

```
else
```

$\{\neg(x < 0)\}$

$\Rightarrow$

$\{x \geq 0\}$

```
    skip
```

$\{x \geq 0\}$

$\{x \geq 0\}$

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

Hoare Logic Continued…

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

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

$\Rightarrow$

$\{I\}$

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

$\{I \wedge x \neq y\}$

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

$\{I\}$

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

$\Rightarrow$

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

Guessing the loop invariant:

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

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

Example: GCD

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

$\Rightarrow$

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

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

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

```
if x > y then
```

$\{\text{gcd}(x, y) = \text{gcd}(N, M) \land \ x \neq y \land x > y\}$

$\Rightarrow$

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

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

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

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

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

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

$\Rightarrow$

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

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

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

$\Rightarrow$

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

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

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

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

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

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

$\Rightarrow$

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

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 ↓

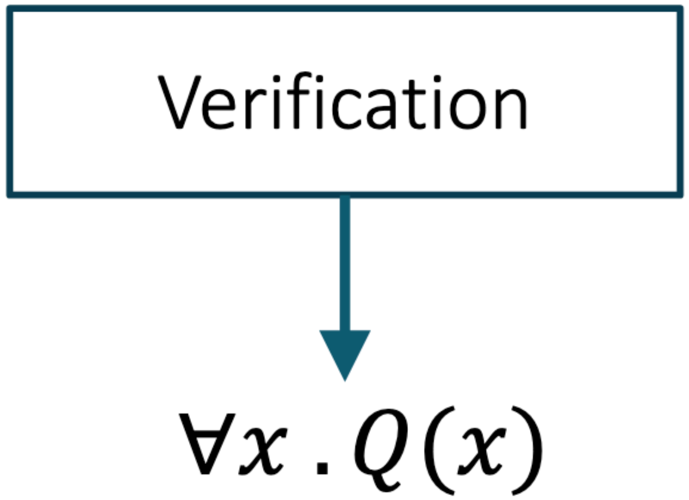
synthesis constraint: $\exists I\ P.\ \forall x\,.\,Q(I, P, x)$

unknown formulas for invariants and commands

synthesizer for loop-free programs ↓

program + invariant ✓ / ✗ no solution

How verification works



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 $\{inv\}$	$\{pre\}\ init(x,x')\ \{inv[x \mapsto x']\}$	$pre \wedge init \Rightarrow inv'$
$\{inv \wedge c\}$ body $\{inv\}$	$\{inv \wedge c\}\ body(x,x')\ \{inv[x \mapsto x']\}$	$inv \wedge c \wedge body \Rightarrow inv'$
$\{inv \wedge \neg c\}$ final $\{post\}$	$\{inv \wedge \neg c\}\ final(x,x')\ \{post\}$	$inv \wedge \neg c \wedge final \Rightarrow post$

$$\forall x .\ (pre \wedge init \Rightarrow inv') \wedge (inv \wedge c \wedge body \Rightarrow inv') \wedge (inv \wedge \neg c \wedge final \Rightarrow post)$$

verification condition

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

From verification to synthesis

Verification

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

SMT

UNSAT / SAT

Synthesis

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

Program synthesis

$\{pre\}$

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

$\{post\}$

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

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

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

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

$pre \wedge S_i \Rightarrow I'$

$\wedge$

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

$\wedge$

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

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

Component-based synthesis using Hoare Logic

Component-based synthesis (CBS)

library

sort: list

reverse: list -> list

take: list -> int -> list

sum: list

query

best_ksum: (l : list) -> (k : int) -> int

i/o examples

49	62	82	54	76	→ k=2	158

best_ksum l k = sum (take (reverse (sort l)) k)

CBS: with effectful components

heap

library

Φ

f: list ref -> int -> list ref

Φ'

Ψ

g: list ref ->

Ψ'

non empty list

query to a CBS

goal: (l : list ref) -> (s : int) -> int

the required heap state for the g function is violated

A sound synthesizer must take changing heap state and library protocol into account

a blowup in the space of programs

A query over a mutable Table

Library

type **pair =** Pair of float ∗ int
type **table =** [string] ref

add_tbl : adds a string in the table if not already present.

mem_tbl : checks if a string is in the table

fresh_str : returns a fresh string not in the table.

size_tbl : gives the size of the tbl

average_len_tbl : gives a float value equall to the average length of the strings in the table

Maintains a Uniqueness Invariant

Query

add_and_incr : (tbl : table ∗ s : string)→
pair

returns a pair type

(*requires*)
{true}

no constraints on the initial table

(*ensures*)
{ mem (Tbl', s) ∧
size (Tbl') = size (Tbl) + 1};

updates the table to include s and increment size by 1

Tbl, Tbl' : [string]

add_and_incr (tbl : table ∗ s :string) **=**
??

Effect agnostic CBS on query

What if s is already in tbl?

add_and_incr : (tbl : table ∗ s : string)→
(*requires*)
{true}
v : pair
(*ensures*)
{ mem (Tbl', s) ∧
size (Tbl') = size (Tbl) + 1};

creates a fresh string if s already in tbl

Cobalt solution

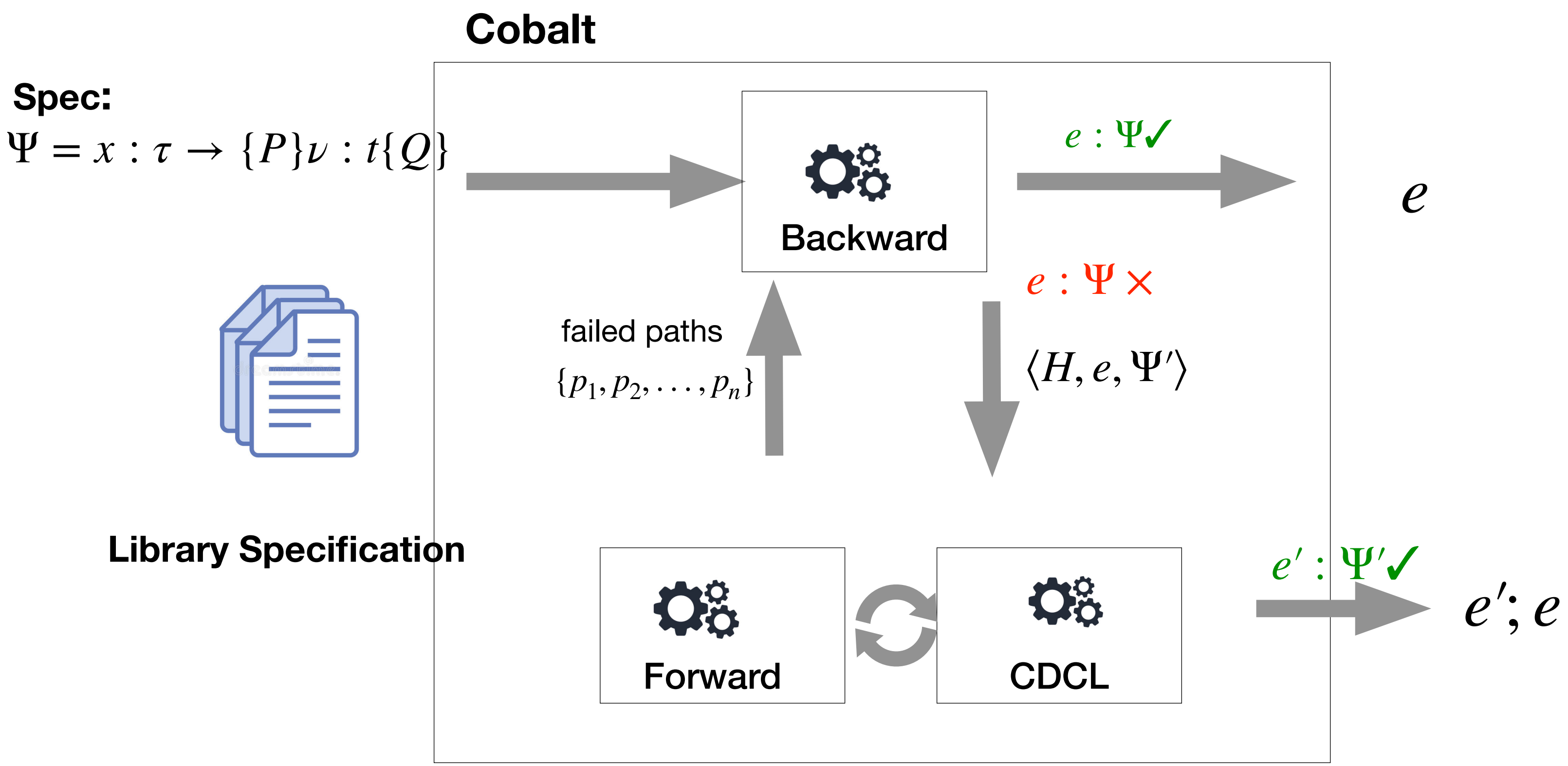
violates uniqueness property of add_tbl

```
add_and_incr (tbl : table ∗ s : string) =
  _ ← add_tbl (tbl, s);
  x1 ← average_len_tbl (tbl);
  y1 ← size_tbl (tbl);
  return Pair (x1, y1)
```

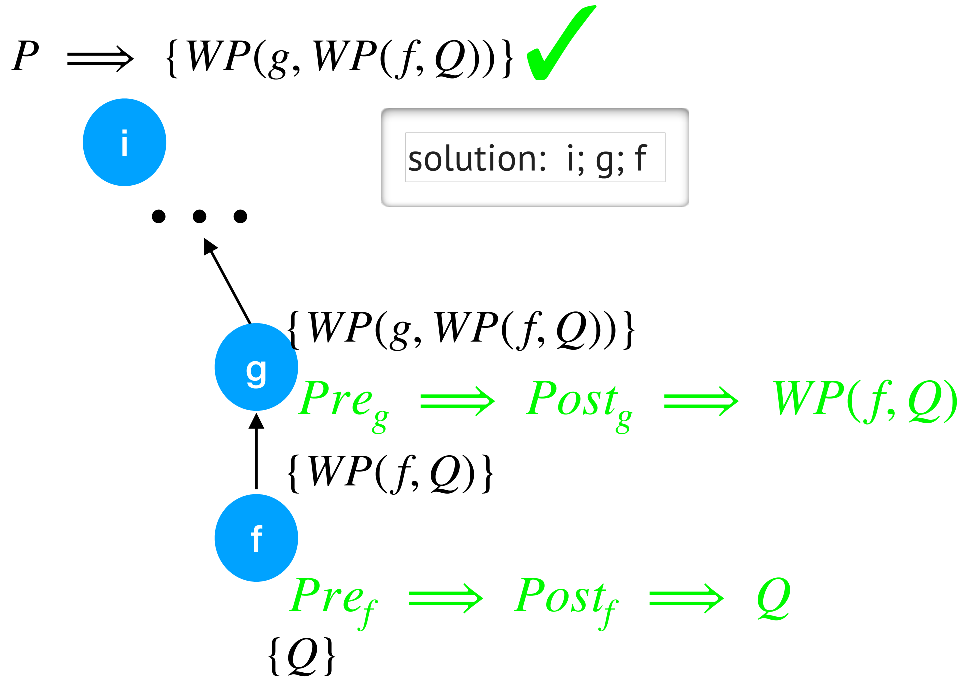
splits control flow

```
add_and_incr (tbl : table ∗ s : string) =
 b1 ← mem (s);
 if (b1) then
  s1 ← fresh_str (tbl);
  _ ← add_tbl(tbl, s1);
  x1 ← average_len_tbl (tbl);
  y1 ← size_tbl (tbl);
  return Pair (x1, y1)
else
  _ ← add_tbl (tbl, s);
  x1 ← average_len_tbl (tbl);
  y1 ← size_tbl (tbl);
  return Pair (x1, y1)
```

Overview: Cobalt

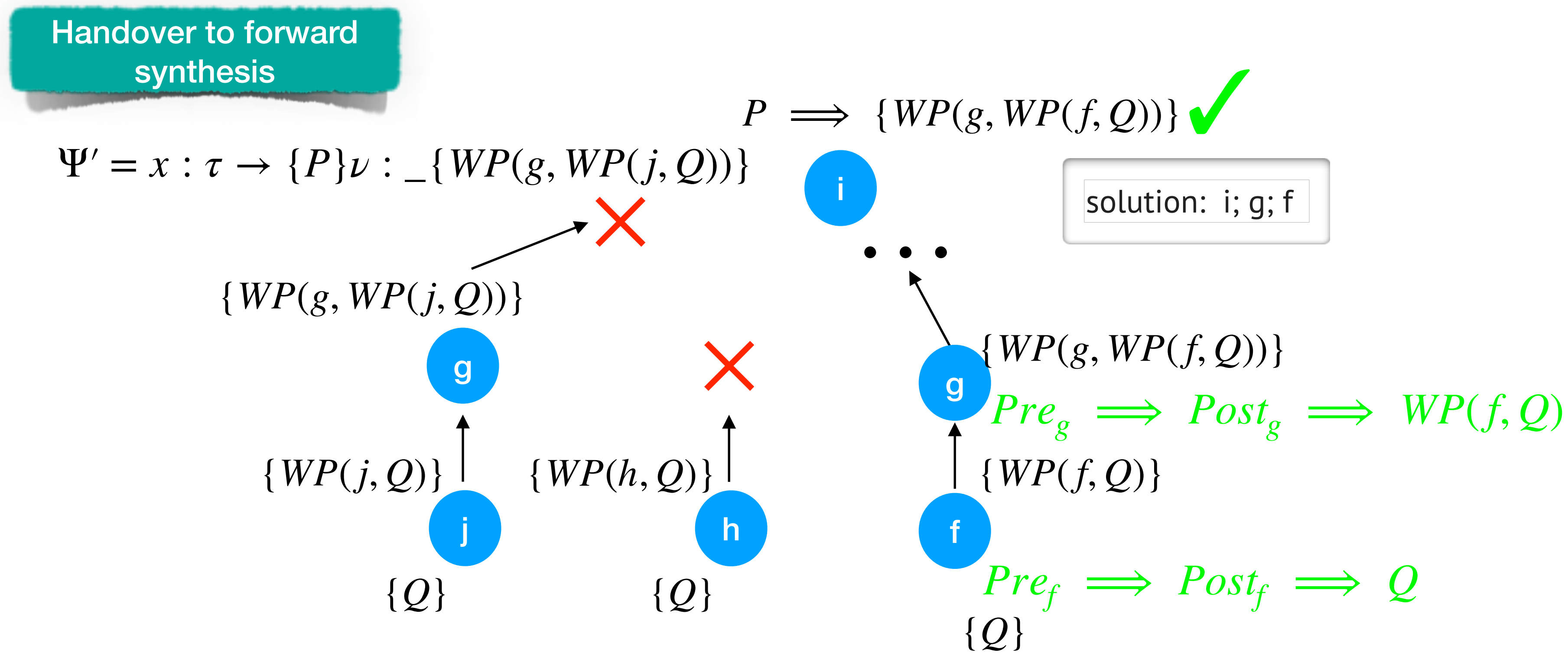


Backward synthesis



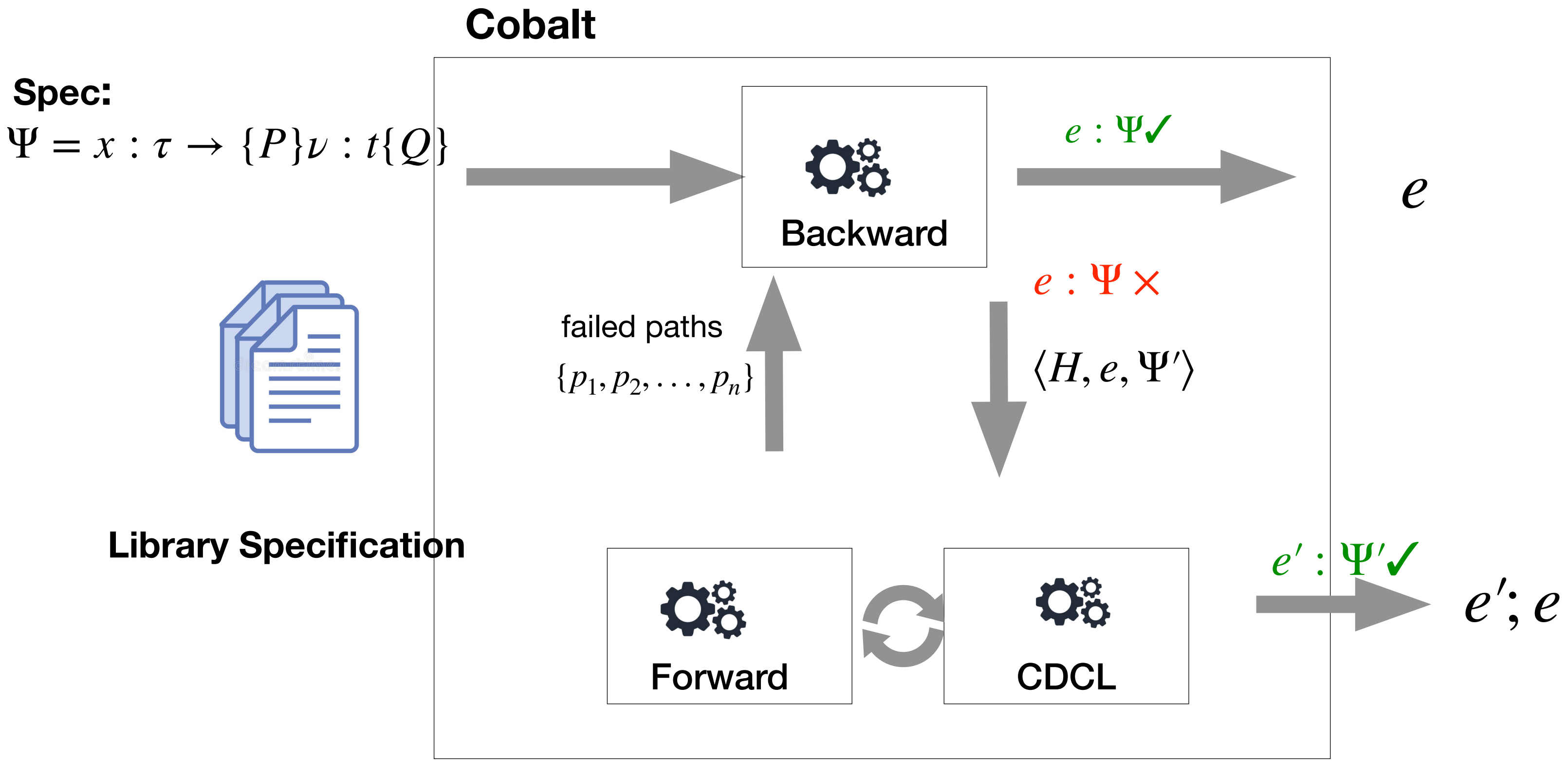
$$\text{Spec } \Psi = x : \tau \rightarrow \{P\}\nu : t\{Q\}$$

Backward synthesis



Spec $\Psi = x : \tau \to \{P\}\nu : t\{Q\}$

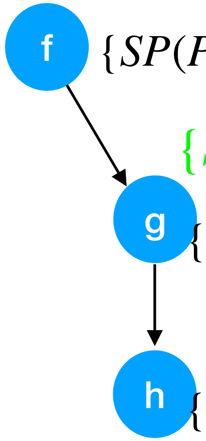
Forward synthesis



finite-depth Forward synthesis

Spec $\Psi = x : \tau \rightarrow \{P\}\nu : t\{Q\}$

$$\{P\} \implies Pre_f$$



$$\{SP(P, f)\}\nu : t\{Q\}$$

$$\{SP(P, f)\} \implies Pre_g$$

$$\{SP(SP(P, f), g)\}\nu : t\{Q\}$$

$$\{SP(SP(P, f), g, h)\}\nu : t\{Q\}$$

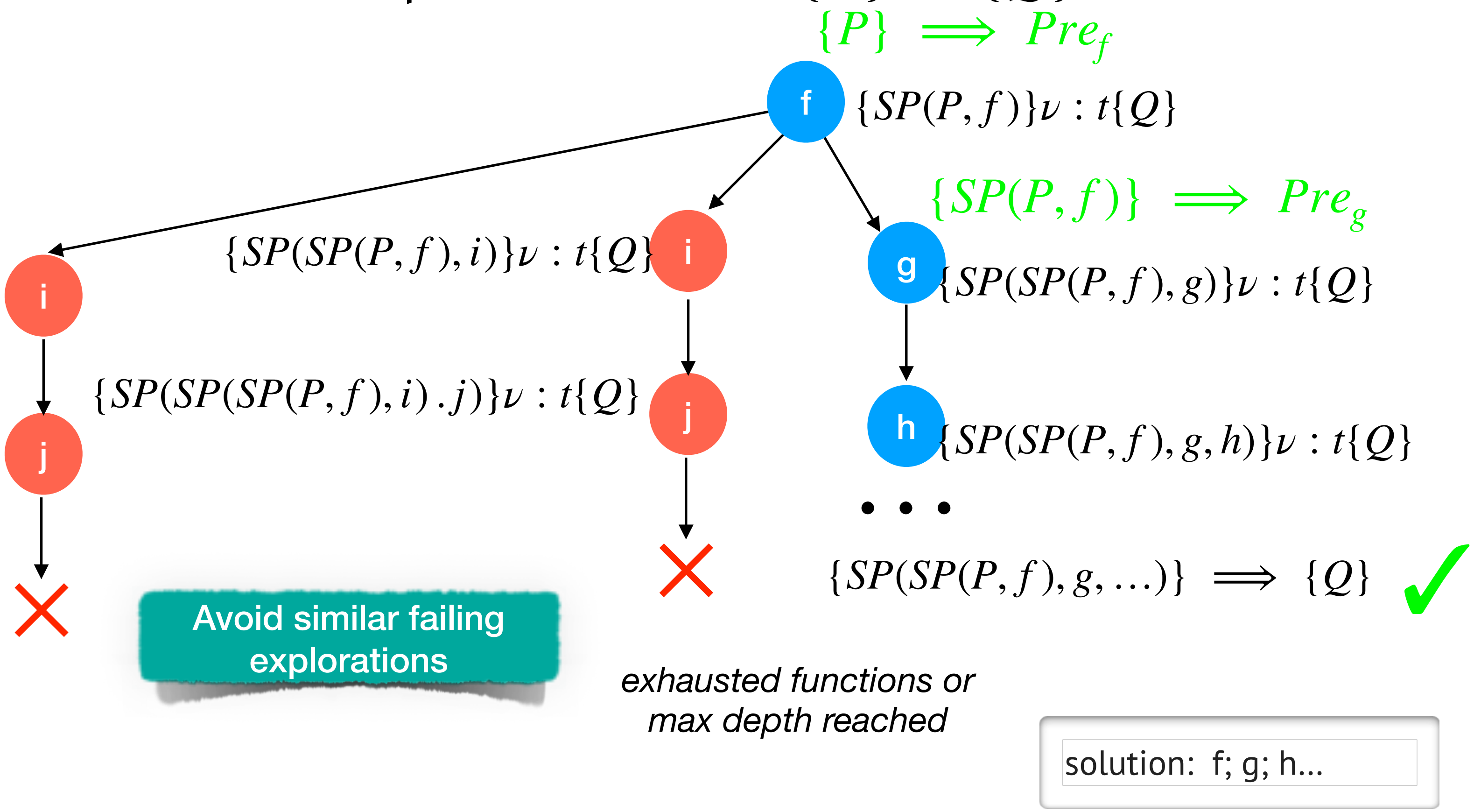
• • •

$$\{SP(SP(P, f), g, \ldots)\} \implies \{Q\}$$ ✓

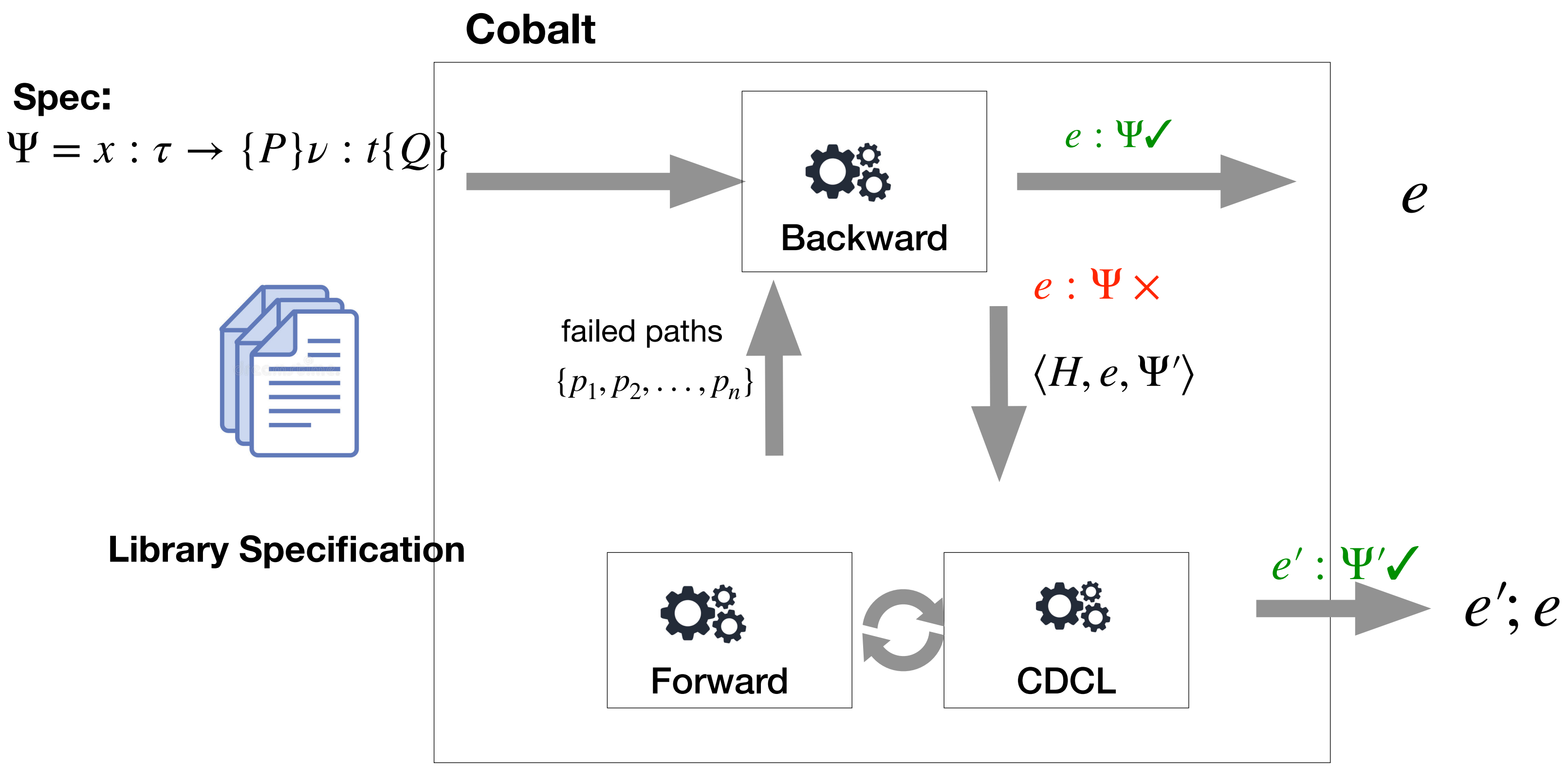
solution: f; g; h...

finite-depth forward synthesis

Spec $\Psi = x : \tau \rightarrow \{P\}\nu : t\{Q\}$



CDCL search


Cobalt
Spec:
$\Psi = x:\tau \to \{P\}\nu : t\{Q\}$
Library Specification
Backward
$e : \Psi$ ✓
e
$e : \Psi$ ×
failed paths
$\{p_1, p_2, \ldots, p_n\}$
$\langle H, e, \Psi' \rangle$
Forward
CDCL
$e' : \Psi'$ ✓
$e'; e$

Revisiting the query

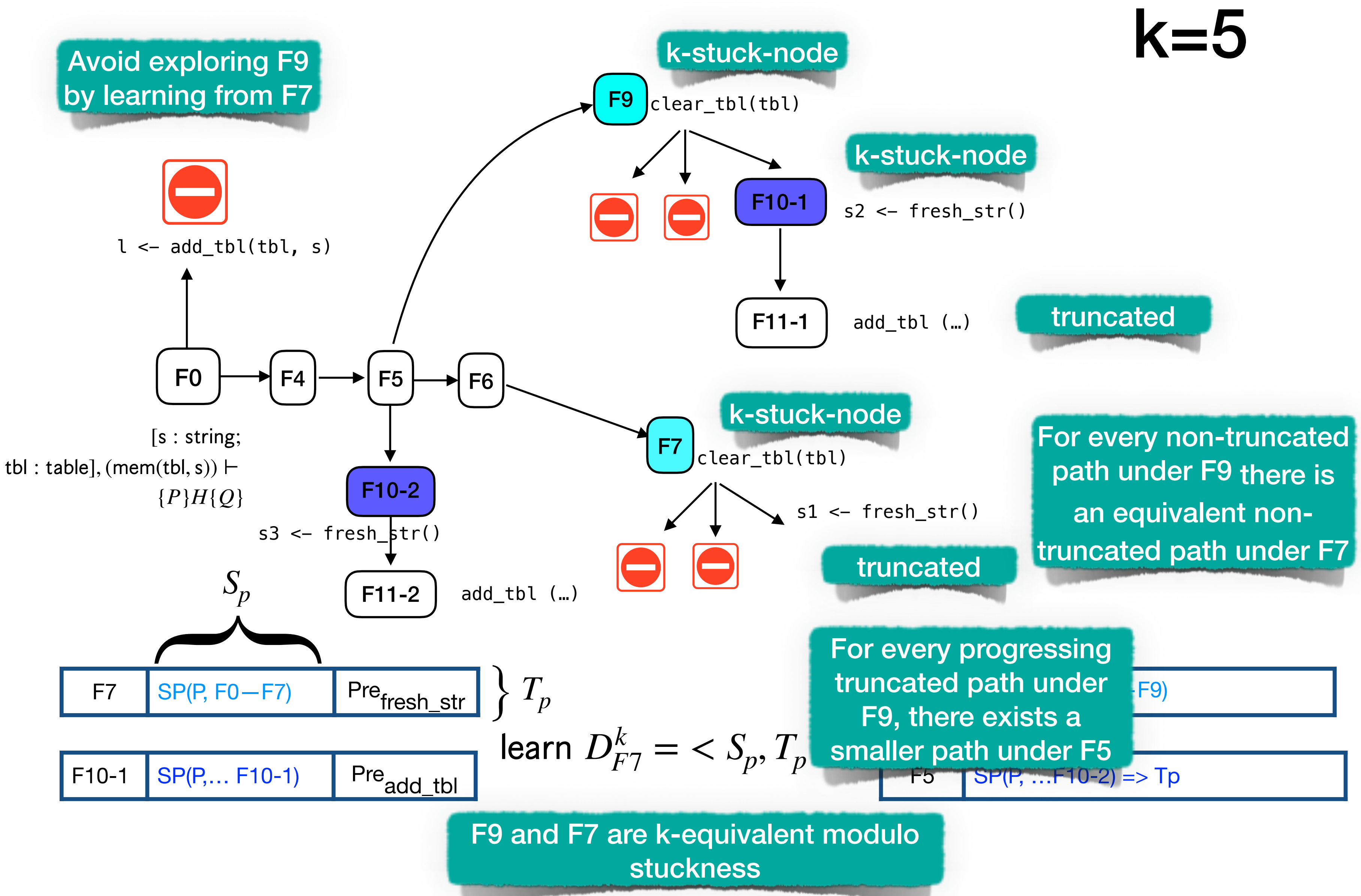
subquery for when s is in the table

Query

add_and_incr : (tbl : table ∗ s : string)→
(*requires*)
{true}
v : pair
(*ensures*)
{ mem (Tbl', s) ∧
size (Tbl') = size (Tbl) + 1};

add_and_incr (*tbl* : table ∗ s :string) **=**
b1 ← mem (s)**;**
if (b1) **then**
$?? : \{(\text{mem}(\text{Tbl}, s)\}$
$v : \text{pair}$
$\{\text{mem}(\text{Tbl}', s)$
$\wedge\ \text{size}(\text{Tbl}') =$
$\text{size}(\text{Tbl}) + 1\}$
else
?? : ...

finite-depth CDCL search



Synthesis guarantees

- The synthesis algorithm is *sound* and *complete.*

Theorem (Soundness): For a given (Γ, Σ, Ψ), if Cobalt synthesizes a term *e* then $\Gamma \vdash e : \Psi$

Type Environment, Library and Specification

Theorem (Completeness): For all k, for a given (Γ, Σ, Ψ), if Cobalt fails to find a solution then there exists no *e* of size (e) <= k, such that $\Gamma \vdash e : \Psi$