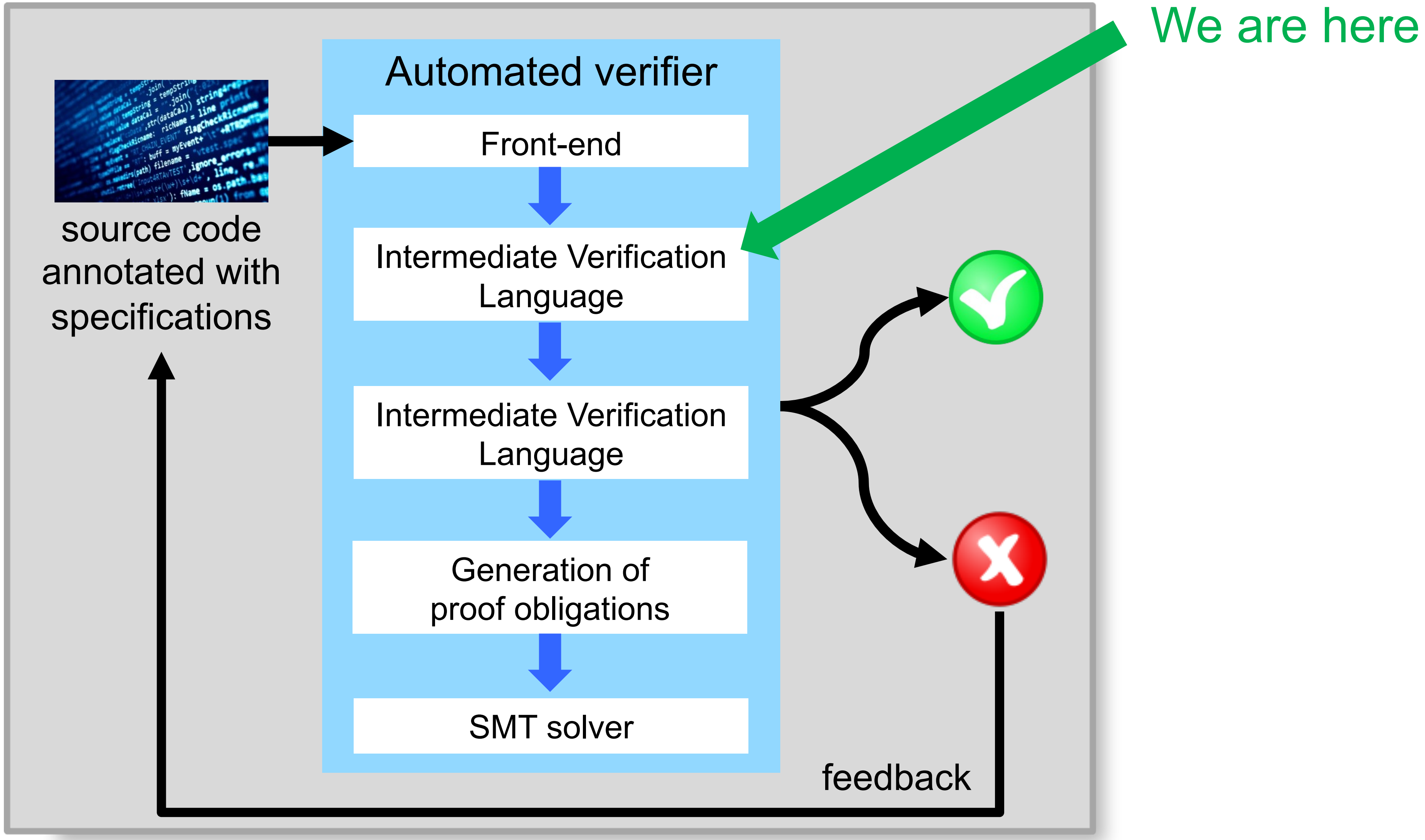
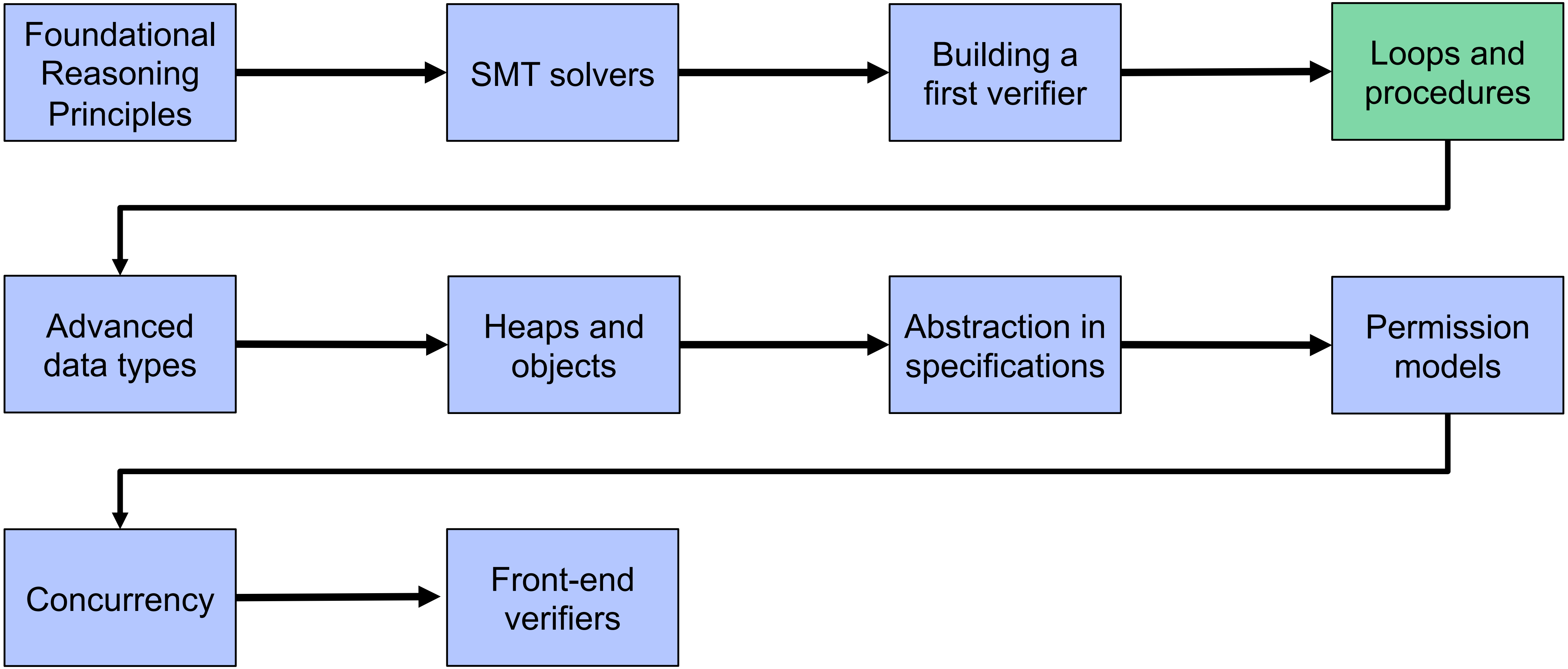
02245 – Chapter 4

LOOPS & PROCEDURES

Roadmap


We are here
Automated verifier
Front-end
Intermediate Verification Language
Intermediate Verification Language
Generation of proof obligations
SMT solver
source code annotated with specifications
feedback

Tentative course outline



02245 – Chapter 4.1

LOOPS

Loops – operationally

Statements

```
S  ::=  ...  |  while (b) { S }
```

- If guard b holds, execute S and run loop again
- If b does not hold, terminate without an effect

Semantics

$$\frac{}{\texttt{while (b) \{ S \}}, \sigma \Rightarrow \texttt{if (b) \{ S; while (b) \{ S \} \} else \{ skip \}}, \sigma}$$

```
assert true
```

Loops – by example

Statements

```
S  ::=  ...  |  while (b) { S }
```

- If guard b holds, execute loop body S and repeat
- If guard b does not hold, terminate

```
assume n >= 0

var i: Int := 1
var r: Int := 0

while (i <= n) {
  r := r + i
  i := i + 1
}

assert ???
```

What should hold after the loop?

n = 5 (before guard)	
i	r
1	0
2	1
3	3
4	6
5	10

Reminder

(see 1.3)

`{ P } S { Q }` is valid for total correctness iff

1. Safety:

   executing `S` on any state in `P` never fails an assertion

2. Partial correctness:

   every terminating execution of `S` on a state in `P` ends in a state in `Q`

3. Termination:

   every execution of `S` on a state from `P` stops after finitely many steps

iff verification condition `P ==> WP(S, Q)` is valid

Loops – by example

- Safety: loop execution does not fail

- Partial correctness: postcondition is satisfied if the loop terminates

- Termination of the loop

```
assume n >= 0

var i: Int := 1
var r: Int := 0

while (i <= n) {
  r := r + i
  i := i + 1
}

assert r == n * (n+1) / 2
assert n >= 0
```

Outline

- Weakest preconditions of loops
- Partial correctness reasoning
- Termination
- Encoding to PL0

Loops – operationally (reminder)

Statements

```
S  ::=  ...  |  while (b) { S }
```

- If guard b holds, execute S and run loop again
- If b does not hold, terminate without an effect

Semantics

$$\frac{}{\texttt{while (b) \{ S \}},\ \sigma \ \Rightarrow\ \texttt{if (b) \{ S; while (b) \{ S \} \} else \{ skip \}},\ \sigma}$$

`assert true`

Loops – via unrolling

```
WP(while (b) { S }, Q)

=

WP(if (b) { S; while (b) { S } } else { skip }, Q)

=

(b ==> WP(S, WP(while (b) { S }, Q))) && (!b ==> Q)

::=  Φ(WP(while (b) { S }, Q))
```

➔ Solution is a fixed point of $X = \Phi(X)$

Running example

```
Φ(X) ::= (b ==> WP(S, X)) && (!b ==> Q)
```

```
Φ(X) ::=
  (i <= n ==> X[i / i+1][r / r+i]) &&
  (!(i <= n) ==> n >= 0 &&
                 r == n * (n+1) / 2)
```

```
while (i <= n) {
  { X[i / i+1][r / r+i] }
  r := r + i
  { X[i / i+1] }
  i := i + 1
  { X }
}

assert n >= 0
assert r == n * (n+1) / 2
```

Loops – as fixed points

`WP(while (b) { S }, Q)` must be a fixed point of

`Φ(X) ::= b ==> WP(S, X) && !b ==> Q`

- `(Pred, ==>)` is a complete lattice
- `WP(S,_), b ==> _, &&` are monotone and continuous
- $\Phi(X)$ is monotone and continuous
- Tarski-Knaster Theorem: $\Phi(X)$ has at least one fixed point
- Which fixed point do we choose if there is more than one?

reading assignment

Exercise

1. Determine *all* fixed points of $\Phi(X)$ for the loop on the right and an arbitrary `Q`.

2. Which fixed point corresponds to the weakest precondition of the loop, that is, what is

   WP`(`**`while(true) { skip }`**`, Q) ?`

Hint: recall that ***WP***`(S, Q)` is the largest predicate P such that `{ P } S { Q }` is valid for total correctness.

3. Does your answer change if we reason about *partial* instead of *total* correctness? Why (not)?

```
while (true) {
  skip // assert true
}
```

$\Phi(X)$ `::= b ==>` ***WP***`(S,` X`)`
`&& !b ==> Q`

Loops – via weakest precondition

continuous predicate transformer

Weakest precondition of loops

```
WP(while (b) { S }, Q) ::= fix(Φ)

Φ (X) ::= b ==> WP(S, X) && !b ==> Q
```

Relative Completeness Theorem (Cook, 1974).

For PL0 programs and predicates, there exists a predicate that is logically equivalent to `fix(Φ)`.

Loops – via weakest precondition

Weakest precondition of loops

```
WP(while (b) { S }, Q) ::= fix(Φ)

Φ (X) ::= b ==> WP(S, X) && !b ==> Q
```

continuous predicate transformer that depends on `b, S, Q`

Kleene's fixed point theorem (applied to loops)

$$\texttt{fix}(\Phi) = \texttt{sup}\ \{\Phi^n(\texttt{false}) \mid n \in \mathbb{N}\ \}$$

least fixed point may only be reached *in the limit*

$\Phi^\infty(\texttt{false})$

$\uparrow$

. . .

$\Phi^3(\texttt{false})$

$\uparrow$

$\Phi(\Phi(\texttt{false}))$

$\uparrow$

$\Phi(\texttt{false})$

$\uparrow$

$\texttt{false}$

Loops – a proof rule using Kleene's theorem

If we can find a parameterized predicate `I(k)` such that

1. `I(0) ==> Φ(false)`
2. `I(k+1) ==> Φ(I(k))`
3. `P ==>` $\left(\lim_{k\to\infty} \texttt{I(k)}\right)$,

then `P ==>` $\underbrace{\texttt{wp(\textbf{while} (b) \{ S \}}}_{=\ \texttt{fix}(\Phi)}\texttt{, Q)}$.

$$\texttt{false} \;\to\; \Phi\,\texttt{(false)} \;\to\; \texttt{I(0)} \;\to\; \Phi^2\,\texttt{(false)} \;\to\; \texttt{I(1)} \;\to\; \cdots \;\to\; \Phi^\infty\,\texttt{(false)} \;\to\; \lim_{k\to\infty} \texttt{I(k)}$$

Example – via Kleene's theorem

If we can find a parameterized predicate `I(k)` such that

1. `I(0) ==> Φ(false)`
2. `I(k+1) ==> Φ(I(k))`
3. `P ==>` $\left(\lim_{k \to \infty} \texttt{I(k)}\right)$,

then `P ==> wp(while (b) { S }, Q).`

```
I(k) ::= n >= 0 &&
         (i > n ==> r == n * (n+1) / 2) &&
         forall j:Int ::
             1 <= j && j <= k ==>
                 i == n – j + 1 ==>
                     r == (n-j) * (n-j+1) / 2
```

```
assume n >= 0

var i: Int := 1
var r: Int := 0

while (i <= n) {
  r := r + i
  i := i + 1
}

assert r == n * (n-1)/2
```

Example – via Kleene's theorem

If we can find a parameterized predicate `I(k)` such that

1. `I(0) ==> Φ(false)`
2. `I(k+1) ==> Φ(I(k))`
3. `P ==>` $\left(\lim_{k\to\infty} \texttt{I(k)}\right)$,

then `P ==> wp(while (b) { S }, Q)`.

$\lim_{k\to\infty}$
```
I(k) = n >= 0 &&
          (i > n ==> r == n * (n+1) / 2) &&
          forall j:Int ::
              1 <= j ~~&& j <= k~~ ==>
                  i == n - j + 1 ==>
                      r == (n-j) * (n-j+1) / 2
```

```
assume n >= 0

var i: Int := 1
var r: Int := 0

while (i <= n) {
  r := r + i
  i := i + 1
}

assert r == n * (n-1)/2
```

➔ Proves total correctness

➔ Finding I(k) is challenging

➔ Step 3 is hard to automate

Outline

- Weakest preconditions of loops
- Partial correctness reasoning
- Termination
- Encoding to PL0

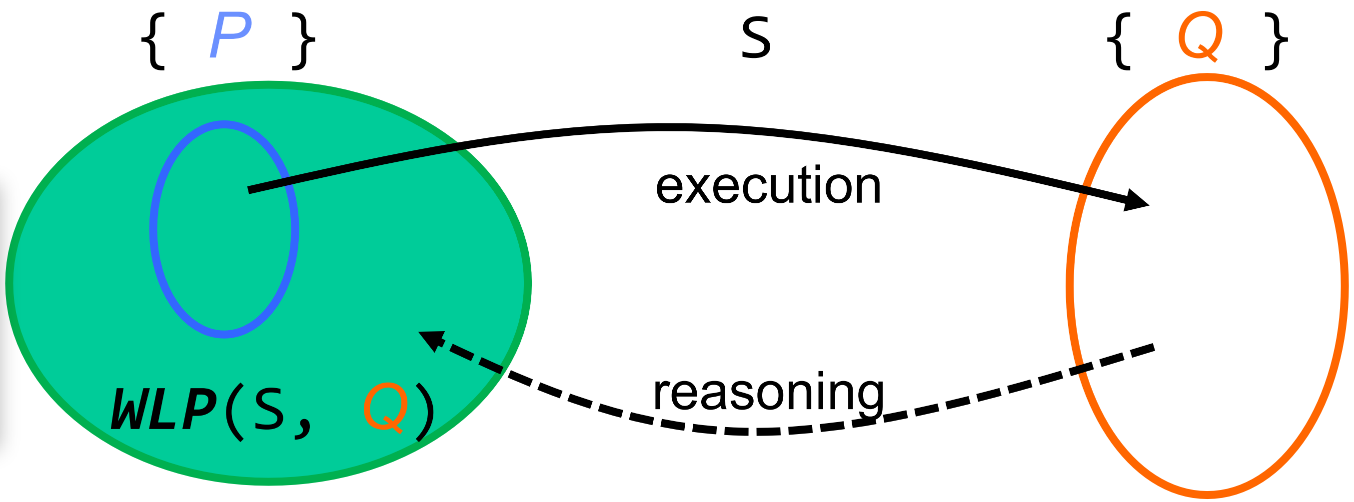
Loops – by example with proof arguments

- Safety: loop execution does not fail
  - No assertion (failure) in the loop
- Partial correctness: postcondition is satisfied if the loop terminates
  - Before every loop iteration: `r == (i - 1) * i / 2`
  - Upon termination we also know `i == n + 1`

```
assume n >= 0

var i: Int := 1
var r: Int := 0

while (i <= n)
   invariant ...
{

  r := r + i
  i := i + 1

}

assert n >= 0
assert r == n * (n+1) / 2
```

Loops – fixed points for partial correctness

Backward VC: P ==> ***WLP***(S, Q)
(are all initial states from which every terminating execution of S ends in Q)




```
while (true) {
  skip
}
```

WLP(**while**(**true**) { **skip** }, Q)
=
true
=
FIX(Φ)

$\Phi(X) = X$

→ Pick *greatest* fixed point FIX(Φ)

Loops – weakest liberal preconditions

Backward VC: *P* ==> ***WLP***(S, Q)
(are all initial states from which every terminating execution of S ends in Q)

S	*WLP*(S, Q)
`var x`	`forall x :: Q`
`x := a`	`Q[x / a]`
`assert R`	`R && Q`
`assume R`	`R ==> Q`
`S1; S2`	***WLP***`(S1, `***WLP***`(S2, Q))`
`S1 [] S2`	***WLP***`(S1, Q) && `***WLP***`(S2, Q)`

Weakest *liberal* precondition of loops

WLP`(while (b) { S }, Q) ::= FIX(Φ)`

`Φ(X) ::= b ==> `***WLP***`(S, X) && !b ==> Q`

Loops – inductive invariants

greatest fixed point

Weakest *liberal* precondition of loops

```
WLP(while (b) { S }, Q) ::= FIX(Φ)
Φ (X) ::= b ==> WLP(S, X) && !b ==> Q
```

pre-fixed point

Tarski-Knaster fixed point theorem

```
FIX(Φ) = sup { I | I ==> Φ(I) }
```

Inductive invariant rule

loop invariant

$$\frac{\texttt{I ==> }\Phi\texttt{(I)}}{\texttt{I ==> }\textit{WLP}\texttt{(while (b) \{ S \}, Q)}}$$

Loop invariants

- Predicate that holds before every iteration

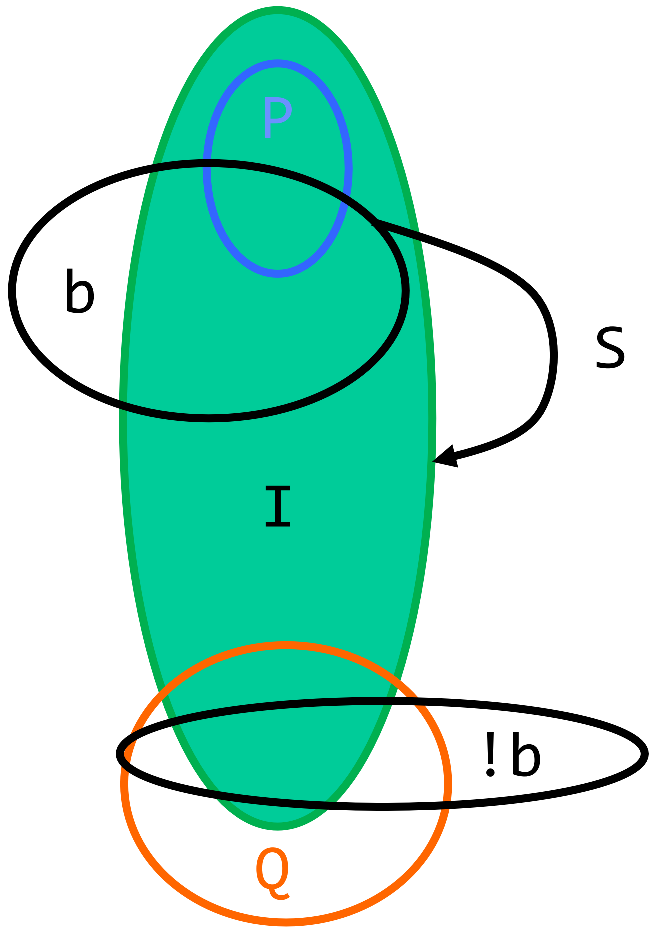
invariant `I` is preserved by one iteration

```
         { I && b } S { I }
------------------------------------
{ I } while (b) { S } { I && !b }
```

How can we derive this rule?

- Can be viewed as an induction proof
  - **Base:** invariant holds before the loop
  - **Hypothesis:** invariant holds before a fixed number of loop iterations
  - **Step:** invariant is preserved after performing one more iteration

{ P } S { Q }



Loop invariants

- Predicate that holds before every iteration

loop invariant I is preserved by one iteration

$$\frac{\{\ I\ \&\&\ b\ \}\ S\ \{\ I\ \}}{\{\ I\ \}\ \texttt{while (b) \{ S \}}\ \{\ I\ \&\&\ !b\ \}}$$

- Can be viewed as an induction proof
  - **Base:** invariant holds before the loop
  - **Hypothesis:** invariant holds before a fixed number of loop iterations
  - **Step:** invariant is preserved after performing one more iteration

```
i := 1
r := 0

{ 0 <= r && 1 <= i }
while (i <= n) {
{ 0 <= r && 1 <= i && i <= n }
==>
{ 0 <= r + i && 1 <= i + 1 }
  r := r + i
{ 0 <= r && 1 <= i + 1 }
  i := i + 1
{ 0 <= r && 1 <= i }
}
{ 0 <= r && 1 <= i && !(i <= n) }
==>
{ 0 <= r }
```

Inductive loop invariants

```
{ I && b } S { I }
─────────────────────────────────
{ I } while (b) { S } { I && !b }
```

- Some predicates hold before every iteration but are not loop invariants
- We must be able to prove that the invariant is preserved
- Often requires strengthening the proposed invariant

```
i := 1
r := 0


while (i <= n) {
{ 0 <= r && i <= n }
==>  // proof fails
{ 0 <= r + i }
  r := r + i
{ 0 <= r }
  i := i + 1
{ 0 <= r }
}
{ 0 <= r && !(i <= n) }
==>
{ 0 <= r }
```

PL1: PL0 + loops with invariants

PL1 Statements

```
S ::= PL0... | while (b) invariant I { S }
```

Approximation of WLP with invariants

```
WLP(while (b) invariant I { S }, Q) ::= I
```

if predicate `I` is a loop invariant

```
i := 1; r := 0

while (i <= n)
  invariant 0 <= r && 1 <= i
{
  r := r + i
  i := i + 1
}
```

- We require loop invariants to be provided by the programmer
- Writing loop invariants is one of the main challenges for program verification
- Preservation of invariants needs to be checked as a side condition
  - invariant wrong ➔ failure

Loops – in Viper

- Viper supports multiple invariants
  - all invariants are conjoined

```
while (0 < x)
  invariant 0 < x
  invariant x < 10
{ … }
```

- Error messages indicate why an invariant does not hold

```
var x: Int

while (0 < x)
  invariant 0 < x
{ … }
```

"Loop invariant might not hold on entry"

```
var x: Int
x := 5

while (0 < x)
  invariant 0 < x
{
  x := x - 1
}
```

"Loop invariant might not be preserved"

Demo

```
method main() {
  var n: Int
  var i: Int
  var r: Int

  assume n >= 0

  i := 1
  r := 0

  while (i <= n)
    invariant ??
  {
    r := r + i
    i := i + 1
  }

  assert r == n * (n+1) / 2
}
```

Exercise

```
method main() {
  var M: Int
  var N: Int
  var res: Int

  assume N > 0 && M >= 0

  var m: Int := M
  res := 0

  while (m >= N)
    invariant ??
  {
    m := m - N
    res := res + 1
  }

  assert M == res * N + m
}
```

```
method main() {
  var n: Int; var m: Int; var res: Int

  assume n >= 0 && m >= 0

  var x: Int := n
  var y: Int := m
  res := 0

  while (x > 0)
    invariant ??
  {
    if (x % 2 == 1) {
      res := res + y
    }
    x := x / 2  // right shift
    y := y * 2  // left shift
  }

  assert res == n * m
}
```

Loops – by example with proof arguments

- Safety: loop execution does not fail
  - No assertion (failure) in the loop

- Partial correctness: postcondition is satisfied if the loop terminates
  - Before every loop iteration: `r == (i - 1) * i / 2`
  - Upon termination we also know `i == n + 1`

```
assume n >= 0

var i: Int := 1
var r: Int := 0

while (i <= n)
   invariant ...
{


  r := r + i
  i := i + 1


}

assert n >= 0
assert r == n * (n+1) / 2
```

Outline

- Weakest preconditions of loops
- Partial correctness reasoning
- Termination
- Encoding to PL0

Proving termination

A loop **variant** is an an expression V that decreases in every loop iteration (for some well-founded ordering <).

< has no infinite descending chains

Well-founded	Not-well-founded
< over Nat	< over Int
⊂ over finite sets	< over positive reals

A loop terminates iff there exists a loop variant.

\# loop iterations

$V_1 > V_2 > V_3 > V_4 > \ ... > \ V_k$

Loop must stop after some finite number k of iterations because < has no infinite descending chains

old value of variant (>= 0)

V decreases

```
{ I && b && V == z } S { I && V < z }
-------------------------------------
    { I } while (b) { S } { I && !b }
```

Example – loops with variants

old value of variant (>= 0)

V decreases

```
{ I && b && V == z } S { I && V < z }
--------------------------------------
{ I } while (b) { S } { I && !b }
```

- Termination is experimental in Viper
- We can model variants with ghost code
  - code that does not affect execution
  - can be safely removed again
  - example: variables that keep track of old values

```
assume n >= 0

var i: Int := 1
var r: Int := 0

while (i <= n)
{
  var z: Int := n - i + 1
  assert z >= 0
  r := r + i
  i := i + 1
  assert n - i + 1 >= 0
  assert n - i + 1 < z
}

assert n >= 0
assert r == n * (n+1) / 2
```

V = n – i + 1

Loops – by example with proof arguments

- Safety: loop execution does not fail
  - No assertion (failure) in the loop
- Partial correctness: postcondition is satisfied if the loop terminates
  - Before every loop iteration: `r == (i - 1) * i / 2`
  - Upon termination we also know `i == n + 1`
- Termination of the loop
  - `n – i + 1 >= 0`, always
  - `n – i + 1` decreases in every loop iteration

```
assume n >= 0

var i: Int := 1
var r: Int := 0

while (i <= n)
   invariant ...
{
  z := variant

  r := r + i
  i := i + 1
  assert variant < z

}

assert n >= 0
assert r == n * (n+1) / 2
```

Outline

- Weakest preconditions of loops
- Partial correctness reasoning
- Termination
- Encoding to PL0

Encoding of loops: naive attempt

$$\frac{\{\ I\ \&\&\ b\ \}\ S\ \{\ I\ \}}{\{\ I\ \}\ \texttt{while (b) \{ S \}}\ \{\ I\ \&\&\ !b\ \}}$$

- Check that loop invariant is preserved via a separate proof obligation

```
assume I
assume b

// encoding of S

assert I
```

- Verify the surrounding code by replacing the loop with statements that check and use the loop invariant

```
assert I

// havoc (reset) the state
var x; var y; // ...

assume I
assume !b
```

Loop framing

```
{ I && b } S { I }
─────────────────────────────────────
{ I } while (b) { S } { I && !b }
```

```
assert I

// havoc (reset) the state
var x; var y; // ...

assume I
assume !b
```

```
x := 0

while (false)
  invariant true
{ skip }

assert x == 0
```

- We often need to prove that a property is not affected by a loop
- Proving the preservation of a property across operations is called framing
- Our rule and our preliminary encoding require all framed properties to be conjoined to the loop invariant

Improved encoding for surrounding code

- It is sufficient to havoc those variables that get assigned to in the loop body
  - all other variables will not change
  - we do not forget their values

Frame rule

$$\frac{\{\ P\ \}\ S\ \{\ Q\ \} \qquad S \text{ modifies no var. in } R}{\{\ P\ \&\&\ R\ \}\ S\ \{\ Q\ \&\&\ R\ \}}$$

- We call the assigned variables loop targets

```
assert I

// havoc all loop targets

assume I
assume !b
```

```
x := 0

while (false)
  invariant true
{ skip }

assert x == 0
```

Improved encoding of invariant preservation

- If we check the invariant in a separate proof, we also check it for states we can never reach given the remaining code

```
assume I
assume b

// encoding of S

assert I
```

```
x := 0

while (true)
  invariant true
{ assert x == 0 }
```

invariant is checked for x == -1

- Solution check loop preservation after prior code

```
// prior code
// reset all loop targets
assume I
assume b

// encoding of S

assert I
```

```
x := 0

while (true)
  invariant true
{ assert x == 0 }
```

Final loop encoding

```
// prior code
// havoc all loop targets
assume I
assume b

// encoding of S

assert I
```

```
// prior code
assert I

// havoc all loop targets
assume I

assume !b
// subsequent code
```

```
// prior code

assert I

// havoc all loop targets

assume I

{
  assume b

  // encoding of S

  assert I
  assume false
} [] {
  assume !b
}

// subsequent code
```

Exercise

- Explain why the right program verifies for the final loop encoding but not for the naive one.

- More exercises online and in code files (13-homework.vpr)

```
assume x > 17
var z: Int := 1
var y: Int := x

while (y > 0)
  invariant y >= 0
{
    y := y - z
}

assert y >= 0
```

Loops: wrap-up