Hardware & Software Verification

John Wickerson

Lecture 4: Dafny
28 October 2025

The need for Dafny

- We need to be able to **reason about** the programs we write, not merely **test** them. There is a large and growing need for this.

- Dafny is a **verification-oriented** programming language. Its compiler will refuse to produce executable code until it has proven the code to be **correct**.

But what does <u>correct</u> mean?

Demo: max of a pair

- named output parameters
- postconditions
- overly weak/strong specifications

Straight-line code

true

`x:=5;`

x=5

`y:=8;`

x=5 ∧ y=8

`z:=x+y;`

x=5 ∧ y=8 ∧ z=x+y

`x:=x+1;`

x=6 ∧ y=8 ∧ z=x-1+y

(x+1)=6 ∧ y=8 ∧ z=(x+1)-1+y

`x:=x+1;`

x=6 ∧ y=8 ∧ z=x-1+y

P

`x:=E;`

P ∧ x=E

✗ (works if E and P don't mention x)

P[E/x]

`x:=E;`

P

✓

If-statements

$x=5 \wedge y=8 \wedge z=x+y$

```
if (w > 5) {
```

$x=5 \wedge y=8 \wedge z=x+y \wedge w>5$

```
w:=5;
```

$x=5 \wedge y=8 \wedge z=x+y \wedge w=5$

```
} else {
```

$x=5 \wedge y=8 \wedge z=x+y \wedge w\leq5$

```
v:=10;
```

$x=5 \wedge y=8 \wedge z=x+y \wedge w\leq5 \wedge v=10$

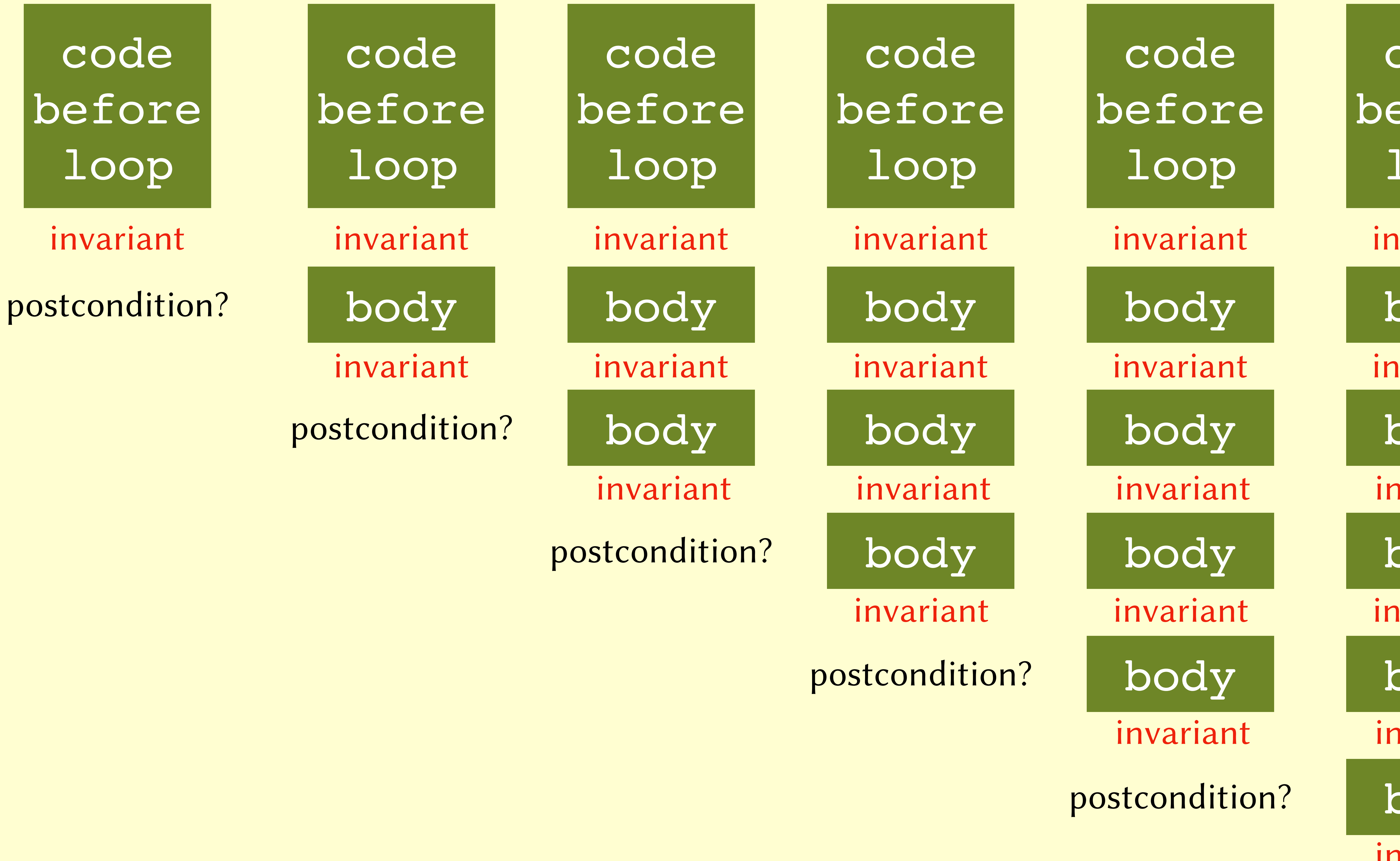
```
}
```

$(x=5 \wedge y=8 \wedge z=x+y \wedge w=5) \vee (x=5 \wedge y=8 \wedge z=x+y \wedge w\leq5 \wedge v=10)$

$x=5 \wedge y=8 \wedge z=x+y \wedge (w=5 \vee (w<5 \wedge v=10))$

Demo: max of an array

The problem with loops



Loop invariants

1.

`code before loop`

$\Downarrow$

invariant

2.

invariant

`body`

$\Downarrow$

invariant

3.

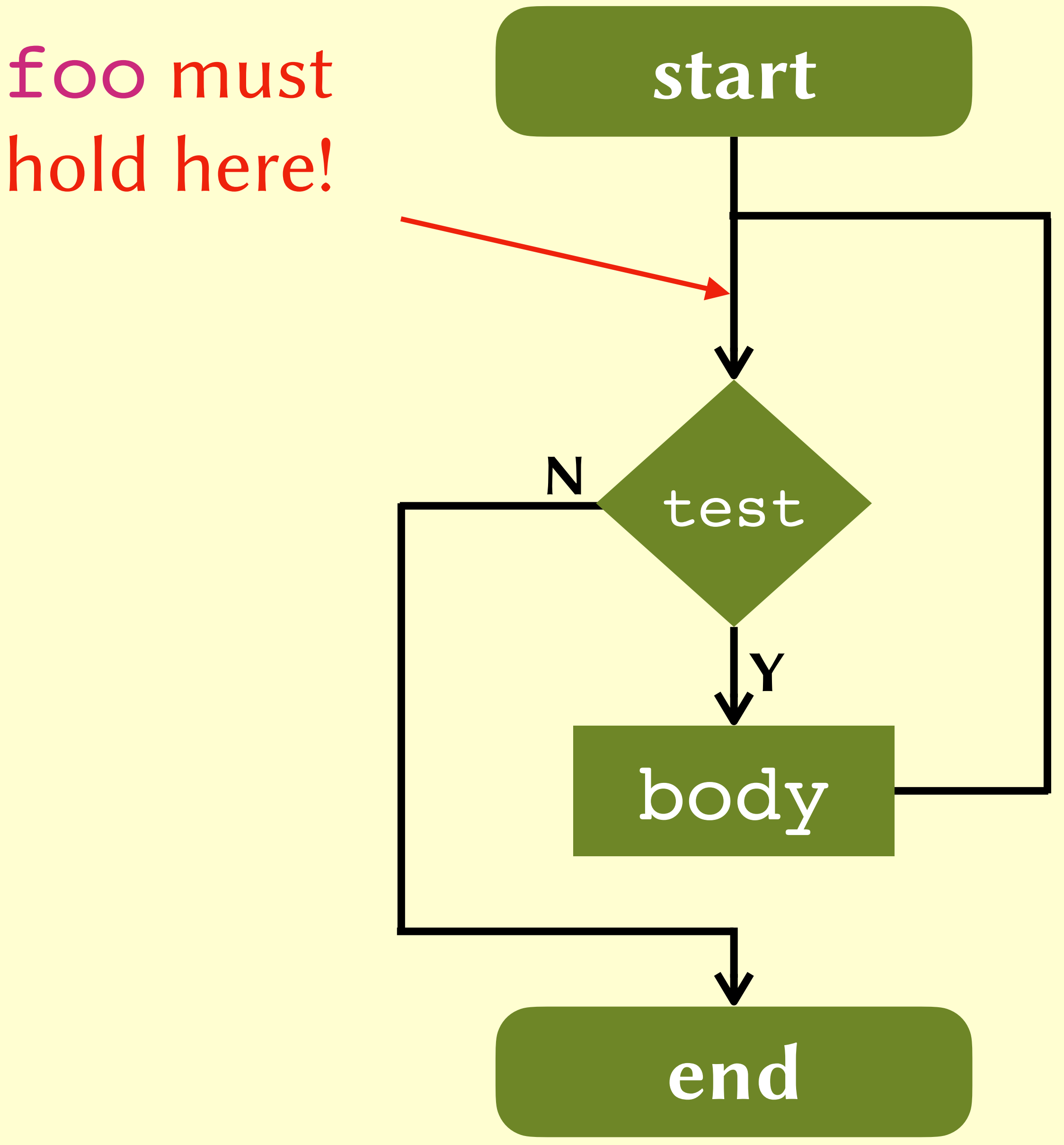
invariant

$\Downarrow$

postcondition

Loop invariants

```
while test
  invariant foo
{
  body
}
```



Finding invariants

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
4	0	1	9	7	1	2

i	r
1	4
2	4
3	4
4	9
5	9
6	9
7	9

```
r := A[0];
var i := 1;
while i < A.Length {
  if r < A[i] {
    r := A[i];
  }
  i := i+1;
}
```

Finding invariants

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
4	0	1	9	7	1	2

i	r	∃j. 0≤j<i ∧ r=A[j]
1	4	
2	4	
3	4	
4	9	
5	9	
6	9	
7	9	

```
r := A[0];
var i := 1;
while i < A.Length {
  if r < A[i] {
    r := A[i];
  }
  i := i+1;
}
```

Finding invariants

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
4	0	1	9	7	1	2

i	r	∃j. 0≤j<i ∧ r=A[j]
1	4	✔
2	4	
3	4	
4	9	
5	9	
6	9	
7	9	

```
r := A[0];
var i := 1;
while i < A.Length {
  if r < A[i] {
    r := A[i];
  }
  i := i+1;
}
```

Finding invariants

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
4	0	1	9	7	1	2

i	r	∃j. 0≤j<i ∧ r=A[j]
1	4	✔
2	4	✔
3	4	
4	9	
5	9	
6	9	
7	9	

```
r := A[0];
var i := 1;
while i < A.Length {
  if r < A[i] {
    r := A[i];
  }
  i := i+1;
}
```

Finding invariants

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
4	0	1	9	7	1	2

```
r := A[0];
var i := 1;
while i < A.Length {
  if r < A[i] {
    r := A[i];
  }
  i := i+1;
}
```

i	r	∃j. 0≤j<i ∧ r=A[j]
1	4	✔
2	4	✔
3	4	✔
4	9	
5	9	
6	9	
7	9	

Finding invariants

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
4	0	1	9	7	1	2

i	r	∃j. 0≤j<i ∧ r=A[j]
1	4	✔
2	4	✔
3	4	✔
4	9	✔
5	9	
6	9	
7	9	

```
r := A[0];
var i := 1;
while i < A.Length {
  if r < A[i] {
    r := A[i];
  }
  i := i+1;
}
```

Finding invariants

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
4	0	1	9	7	1	2

```
r := A[0];
var i := 1;
while i < A.Length {
  if r < A[i] {
    r := A[i];
  }
  i := i+1;
}
```

i	r	∃j. 0≤j<i ∧ r=A[j]
1	4	✔
2	4	✔
3	4	✔
4	9	✔
5	9	✔
6	9	
7	9	

Finding invariants

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
4	0	1	9	7	1	2

i	r	∃j. 0≤j<i ∧ r=A[j]
1	4	✔
2	4	✔
3	4	✔
4	9	✔
5	9	✔
6	9	✔
7	9	

```
r := A[0];
var i := 1;
while i < A.Length {
  if r < A[i] {
    r := A[i];
  }
  i := i+1;
}
```

Finding invariants

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
4	0	1	9	7	1	2

i	r	∃j. 0≤j<i ∧ r=A[j]
1	4	✔
2	4	✔
3	4	✔
4	9	✔
5	9	✔
6	9	✔
7	9	✔

```
r := A[0];
var i := 1;
while i < A.Length {
  if r < A[i] {
    r := A[i];
  }
  i := i+1;
}
```

Finding invariants

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
4	0	1	9	7	1	2

```
r := A[0];
var i := 1;
while i < A.Length {
  if r < A[i] {
    r := A[i];
  }
  i := i+1;
}
```

i	r	∃j. 0≤j<i ∧ r=A[j]	1 ≤ i ≤ A.Length
1	4	✔	
2	4	✔	
3	4	✔	
4	9	✔	
5	9	✔	
6	9	✔	
7	9	✔	

Finding invariants

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
4	0	1	9	7	1	2

```
r := A[0];
var i := 1;
while i < A.Length {
  if r < A[i] {
    r := A[i];
  }
  i := i+1;
}
```

i	r	∃j. 0≤j<i ∧ r=A[j]	1 ≤ i ≤ A.Length
1	4	✔	✔
2	4	✔	
3	4	✔	
4	9	✔	
5	9	✔	
6	9	✔	
7	9	✔	

Finding invariants

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
4	0	1	9	7	1	2

```
r := A[0];
var i := 1;
while i < A.Length {
  if r < A[i] {
    r := A[i];
  }
  i := i+1;
}
```

i	r	∃j. 0≤j<i ∧ r=A[j]	1 ≤ i ≤ A.Length
1	4	✔	✔
2	4	✔	✔
3	4	✔	
4	9	✔	
5	9	✔	
6	9	✔	
7	9	✔	

Finding invariants

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
4	0	1	9	7	1	2

```
r := A[0];
var i := 1;
while i < A.Length {
  if r < A[i] {
    r := A[i];
  }
  i := i+1;
}
```

i	r	∃j. 0≤j<i ∧ r=A[j]	1 ≤ i ≤ A.Length
1	4	✔	✔
2	4	✔	✔
3	4	✔	✔
4	9	✔	
5	9	✔	
6	9	✔	
7	9	✔	

Finding invariants

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
4	0	1	9	7	1	2

```
r := A[0];
var i := 1;
while i < A.Length {
  if r < A[i] {
    r := A[i];
  }
  i := i+1;
}
```

i	r	∃j. 0≤j<i ∧ r=A[j]	1 ≤ i ≤ A.Length
1	4	✔	✔
2	4	✔	✔
3	4	✔	✔
4	9	✔	✔
5	9	✔	
6	9	✔	
7	9	✔	

Finding invariants

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
4	0	1	9	7	1	2

```
r := A[0];
var i := 1;
while i < A.Length {
  if r < A[i] {
    r := A[i];
  }
  i := i+1;
}
```

i	r	∃j. 0≤j<i ∧ r=A[j]	1 ≤ i ≤ A.Length
1	4	✔	✔
2	4	✔	✔
3	4	✔	✔
4	9	✔	✔
5	9	✔	✔
6	9	✔	
7	9	✔	

Finding invariants

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
4	0	1	9	7	1	2

```
r := A[0];
var i := 1;
while i < A.Length {
  if r < A[i] {
    r := A[i];
  }
  i := i+1;
}
```

i	r	∃j. 0≤j<i ∧ r=A[j]	1 ≤ i ≤ A.Length
1	4	✔	✔
2	4	✔	✔
3	4	✔	✔
4	9	✔	✔
5	9	✔	✔
6	9	✔	✔
7	9	✔	

Finding invariants

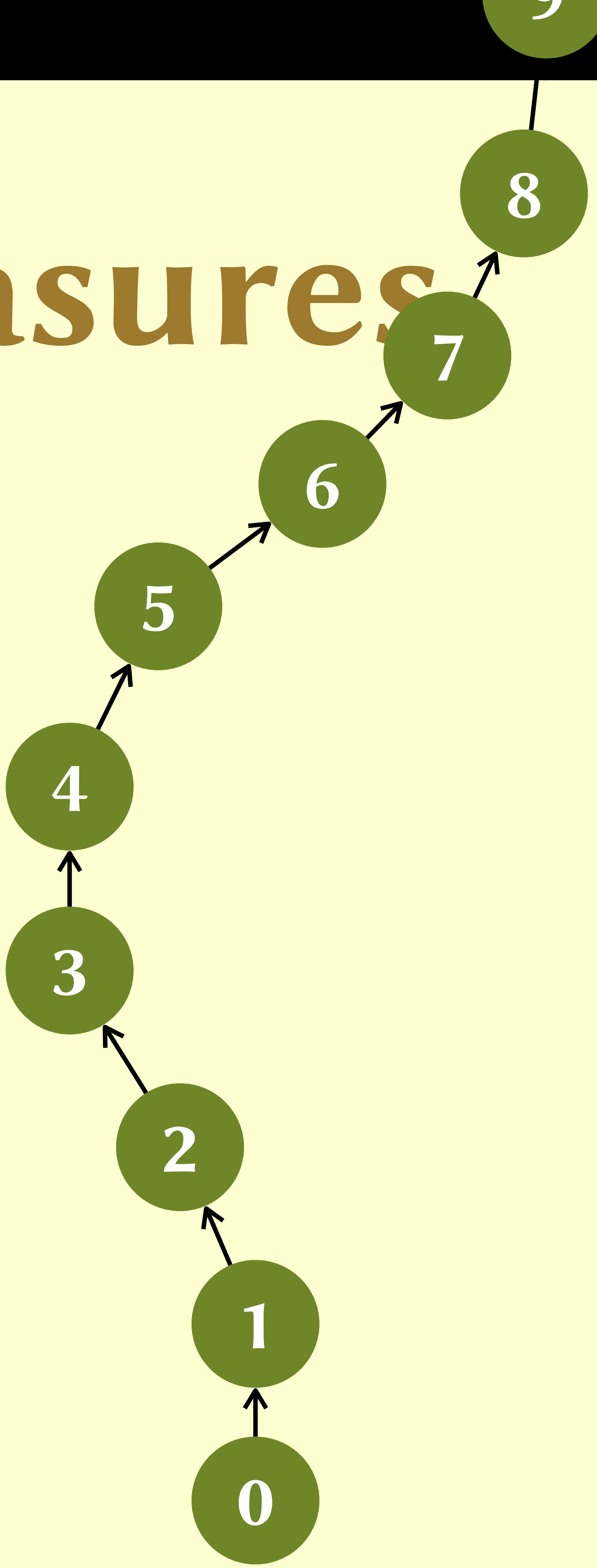
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
4	0	1	9	7	1	2

i	r	∃j. 0≤j<i ∧ r=A[j]	1 ≤ i ≤ A.Length
1	4	✔	✔
2	4	✔	✔
3	4	✔	✔
4	9	✔	✔
5	9	✔	✔
6	9	✔	✔
7	9	✔	✔

```
r := A[0];
var i := 1;
while i < A.Length {
  if r < A[i] {
    r := A[i];
  }
  i := i+1;
}
```

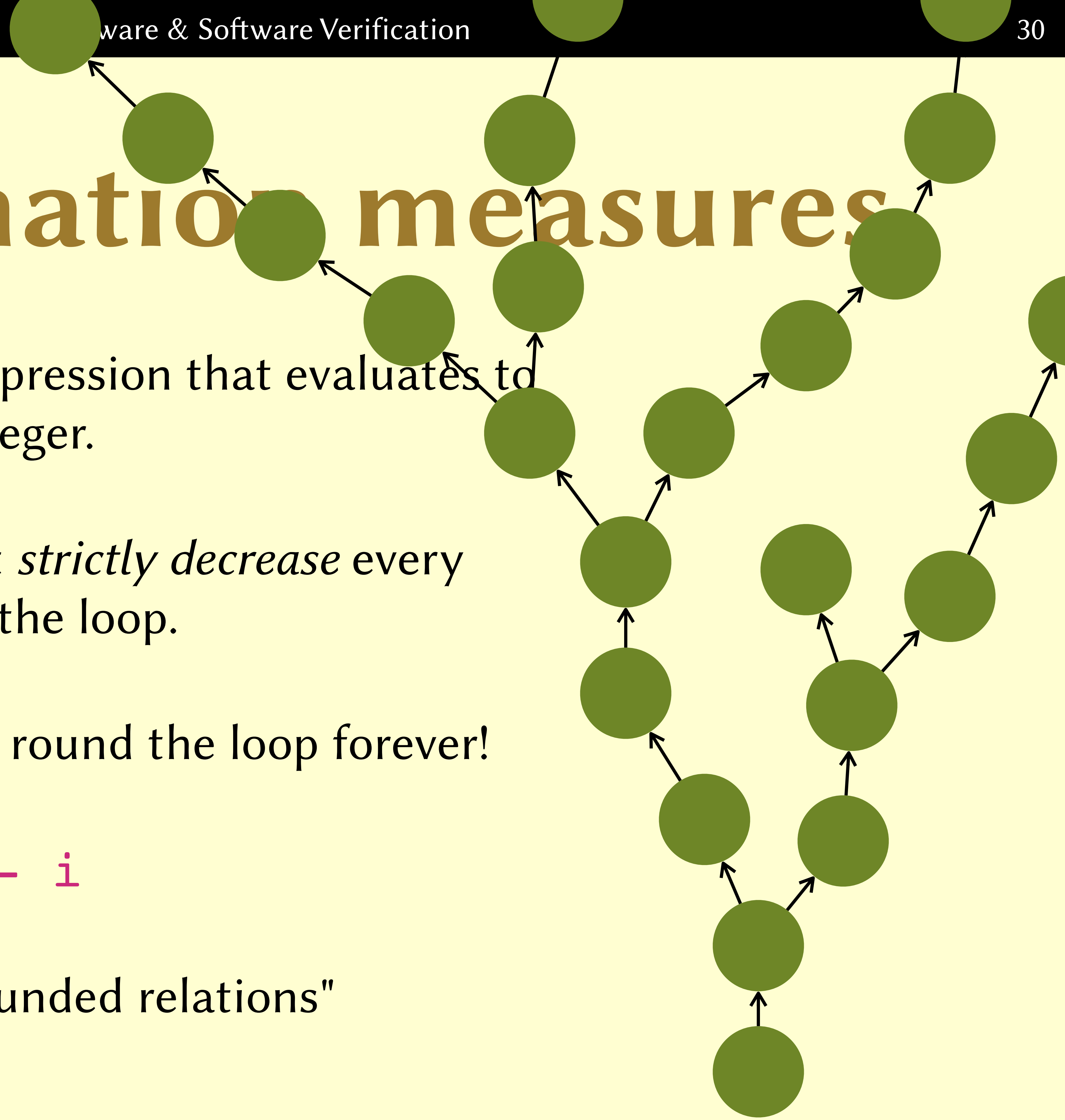
Demo: max of an array

Termination measures



- A *measure* is an expression that evaluates to a non-negative integer.
- The measure must *strictly decrease* every time we go round the loop.
- Hence we can't go round the loop forever!
- E.g.: `A.Length - i`
- "Theory of well-founded relations"

Termination measures



- A *measure* is an expression that evaluates to a non-negative integer.
- The measure must *strictly decrease* every time we go round the loop.
- Hence we can't go round the loop forever!
- E.g.: `A.Length - i`
- "Theory of well-founded relations"

Demo: max of an array

- syntax for variables (**`var`**) and arrays (**`array<...>`**)
- preconditions (**`requires`**)
- termination measures (**`decreases`**)
- universal (**`forall`**) and existential (**`exists`**) quantification
- loop invariants (**`invariant`**)
- predicates (**`predicate`**)