Computer-Aided Reasoning for Software

CSE507

Angelic Execution

courses.cs.washington.edu/courses/cse507/14au/

Emina Torlak
emina@cs.washington.edu

Today

Today

Last lecture

- Verifying compiler optimizations with SMT solvers

Today

Last lecture

- Verifying compiler optimizations with SMT solvers

Today

- Beyond verification: solvers as interpreters

Today

Last lecture

- Verifying compiler optimizations with SMT solvers

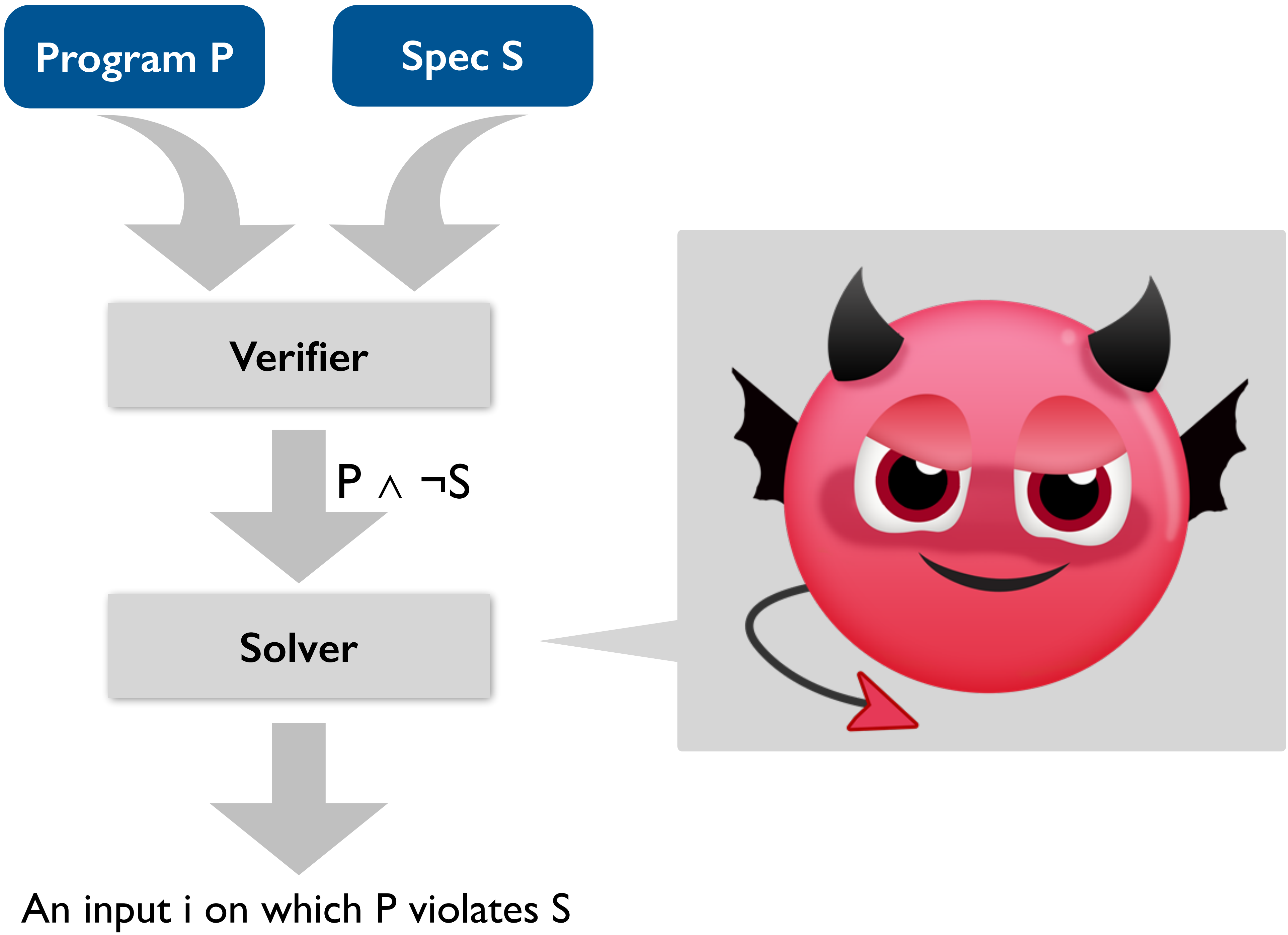
Today

- Beyond verification: solvers as interpreters

Announcements

- Project presentation logistics:
  - 8 min talk (problem statement, demo, results)
  - An electronic poster (single slide)

So far, we have used solvers as demonic oracles



But solvers can also act as angelic oracles

```
P() {
 y = choose();
 …
 assert S;
}
```

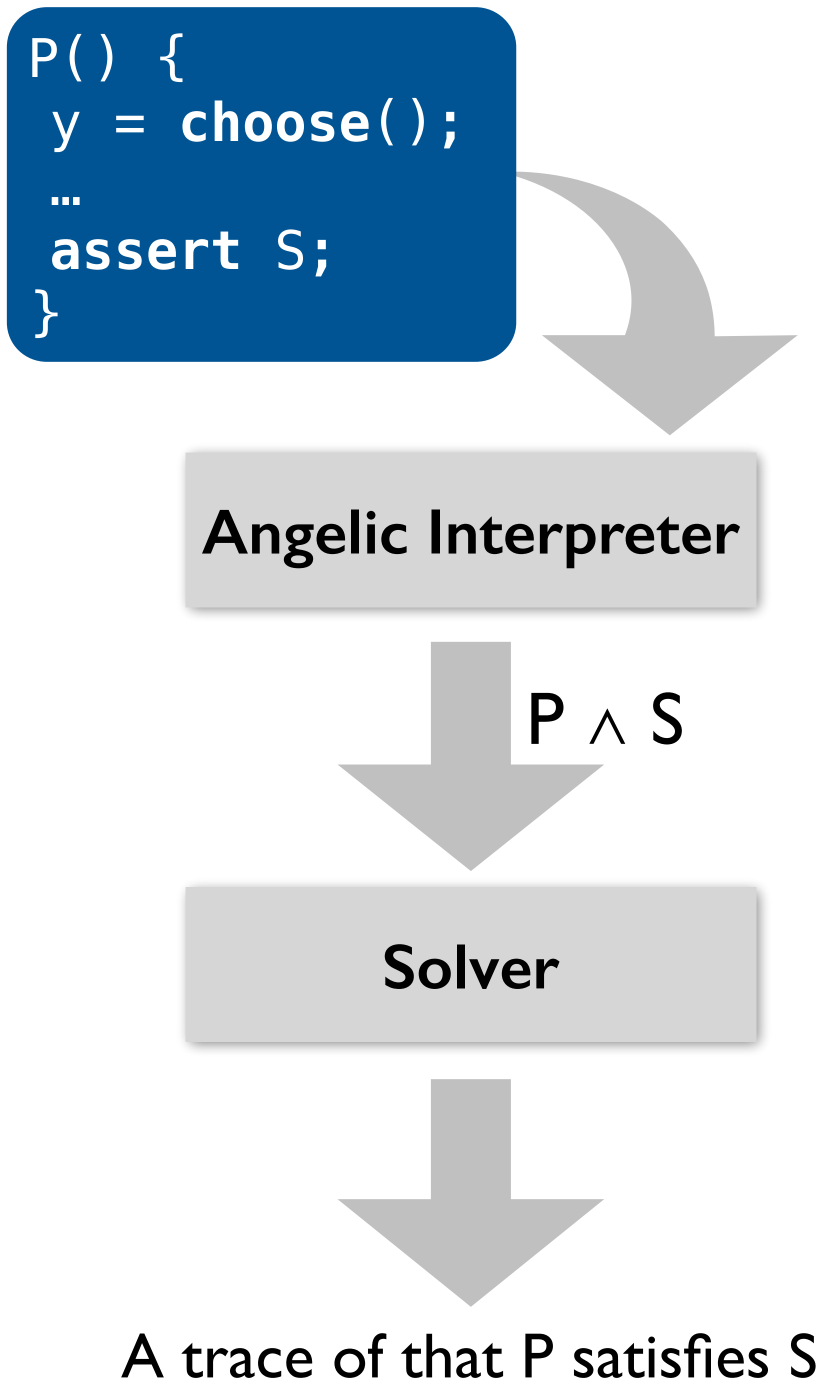
Angelic Interpreter

$P \wedge S$

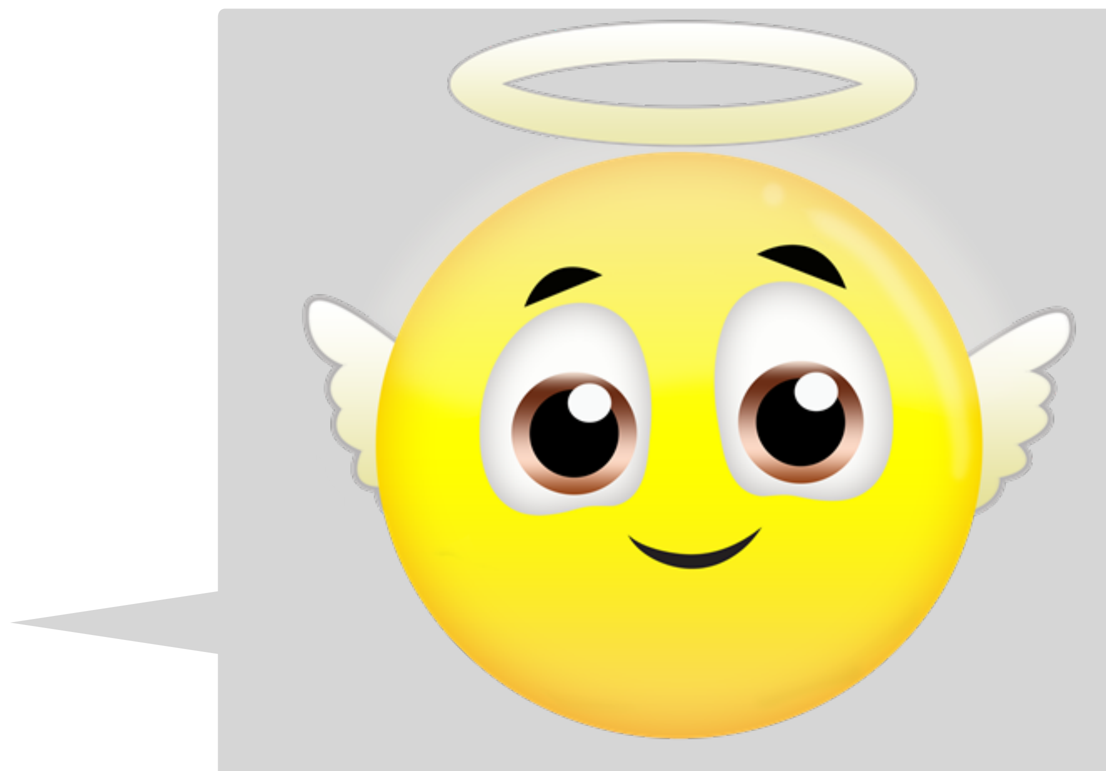
Solver

A trace of that P satisfies S

But solvers can also act as angelic oracles



1. Definitions
2. Implementations
3. Applications



Angelic non-determinism, two ways

Angelic choice:

```
choose(T)
```

Robert Floyd, 1966

Specification statement:

$x_1, \ldots, x_n \leftarrow [pre, post]$

Carroll Morgan, 1988

Angelic non-determinism, two ways

Angelic choice:

`choose(T)`

Non-deterministically chooses a value of (finite) type T so that the rest of the program terminates successfully.

Designed to abstract away the details of backtracking search.

Robert Floyd, 1966

Specification statement:

$x_1, \ldots, x_n \leftarrow [pre, post]$

Carroll Morgan, 1988

A programming abstraction

Angelic non-determinism, two ways

Angelic choice:

`choose(T)`

Robert Floyd, 1966

A programming abstraction

Specification statement:

$x_1, \ldots, x_n \leftarrow$ `[pre, post]`

Non-deterministically modifies the values of frame variables $x_1, \ldots, x_n$ so that *post* holds in the next state if *pre* holds in the current state.

Designed to enable derivation of programs from specifications via step-wise refinement.

Carroll Morgan, 1988

A refinement abstraction

Angelic non-determinism, two ways: an example

Angelic choice:

`choose(T)`

Specification statement:

$x_1, \ldots, x_n \leftarrow$ [pre, post]

```
s = 16
r = choose(int)
if (r ≥ 0)
  assert r*r ≤ s < (r+1)*(r+1)
else
  assert r*r ≤ s < (r−1)*(r−1)
```

```
s = 16
r ← [(r ≥ 0 ∧
       r*r ≤ s < (r+1)*(r+1)) ∨
      (r < 0 ∧
       r*r ≤ s < (r−1)*(r−1))]
```

Angelic non-determinism, two ways: an example

Angelic choice:

`choose(T)`

Specification statement:

$x_1, \ldots, x_n \leftarrow$ [pre, post]

```
s = 16
r = choose(int)
if (r ≥ 0)
  assert r*r ≤ s < (r+1)*(r+1)
else
  assert r*r ≤ s < (r-1)*(r-1)
```

Interleaves imperative and angelic execution. As a result, implementation requires global constraint solving.

```
s = 16
r ← [(r ≥ 0 ∧
      r*r ≤ s < (r+1)*(r+1)) ∨
     (r < 0 ∧
      r*r ≤ s < (r-1)*(r-1))]
```

Alternates between angelic and imperative execution. As a result, implementation requires only local constraint solving.

Angelic non-determinism, two ways: an example

Angelic choice:
`choose(T)`

Specification statement:
$x_1, \ldots, x_n \leftarrow$ [pre, post]

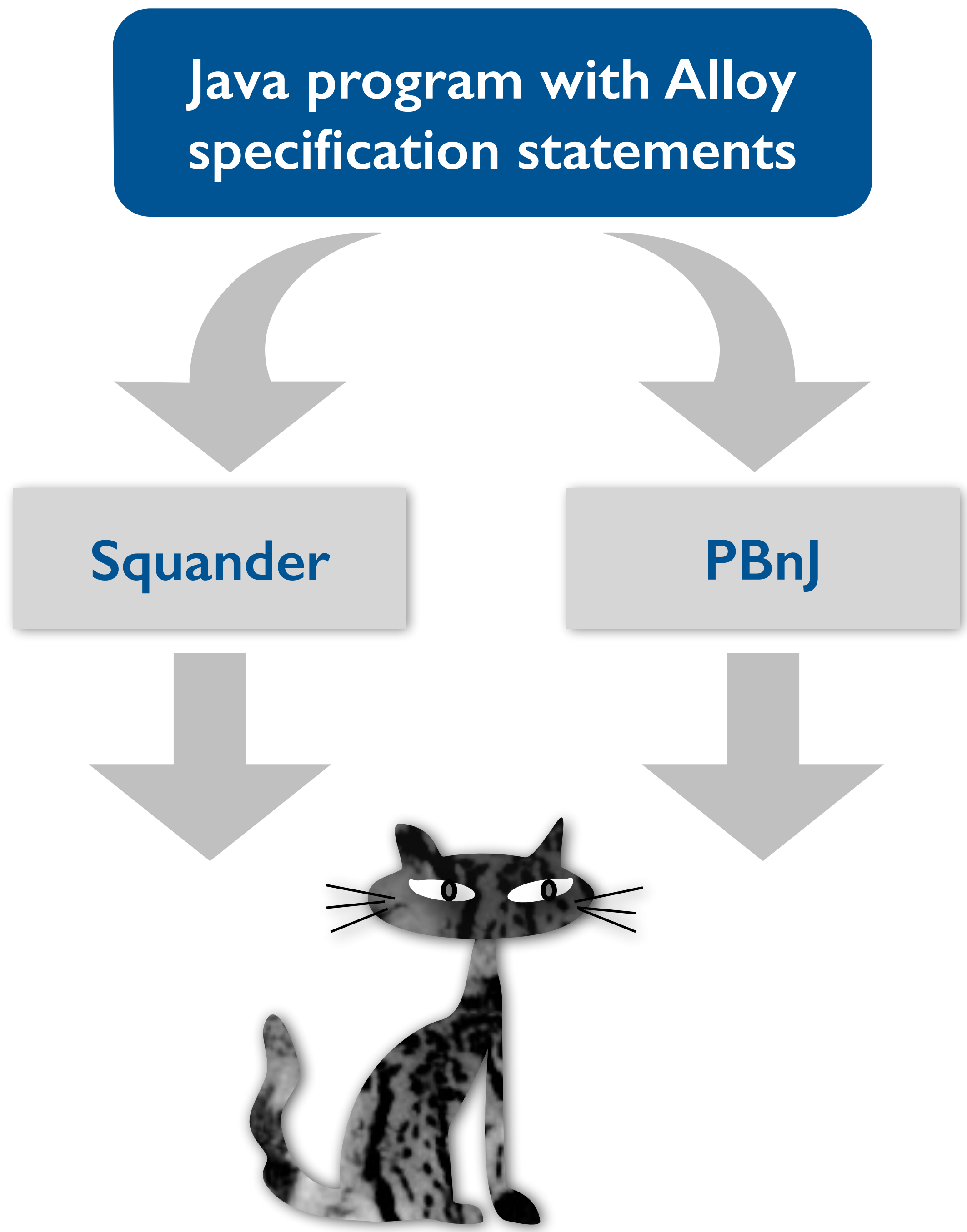
```
s = 16
r = choose(int)
if (r ≥ 0)
  assert r*r ≤ s < (r+1)*(r+1)
else
  assert r*r ≤ s < (r−1)*(r−1)
```

"Angelic Interpretation"

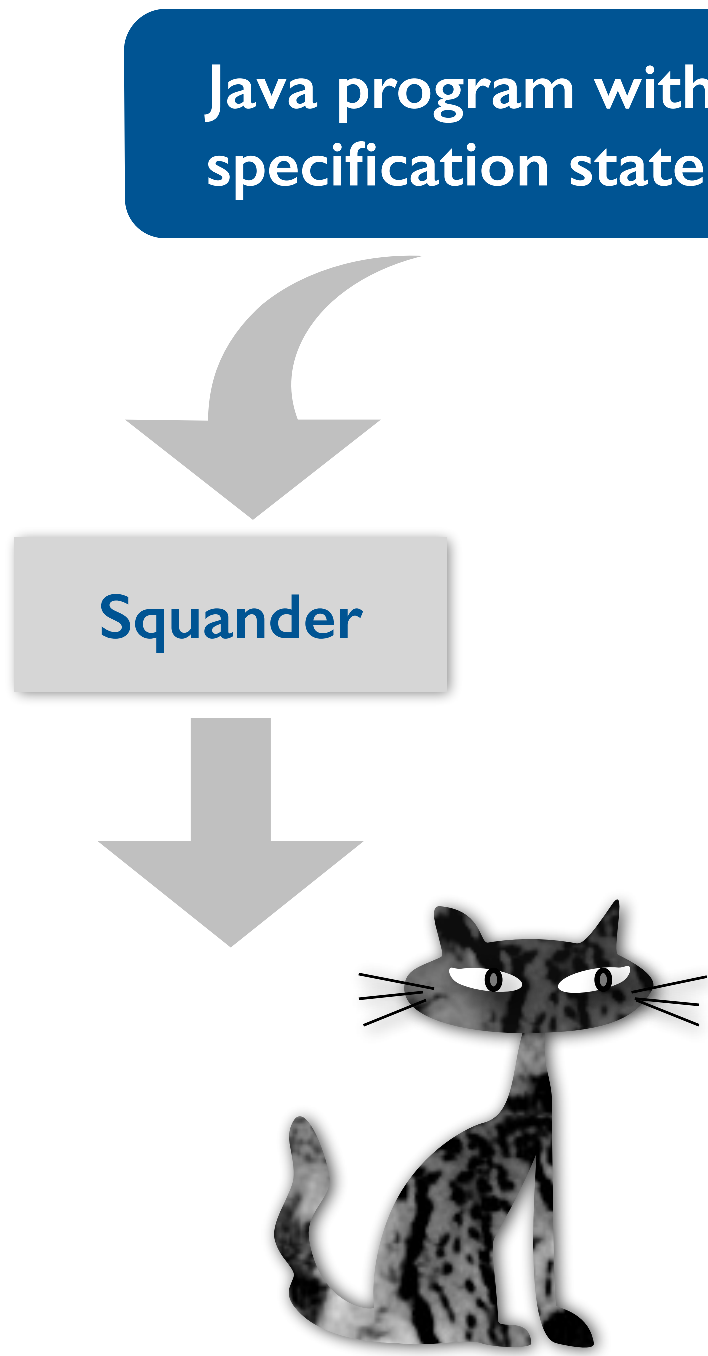
```
s = 16
r ← [(r ≥ 0 ∧
      r*r ≤ s < (r+1)*(r+1)) ∨
     (r < 0 ∧
      r*r ≤ s < (r−1)*(r−1))]
```

"Mixed Interpretation"

Mixed interpretation with a model finder (1/4)



Mixed interpretation with a model finder (1/4)



Mixed interpretation with a model finder (2/4)

```
@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old(this.nodes) + z")
@Modifies("this.root,
           this.nodes.left | _<1> = null,
           this.nodes.right | _<1> = null")

public void insert(Node z) {
  Squander.exe(this, z); }
```

Mixed interpretation with a model finder (2/4)

```
@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old(this.nodes) + z")
@Modifies("this.root,
           this.nodes.left | _<1> = null,
           this.nodes.right | _<1> = null")

public void insert(Node z) {
  Squander.exe(this, z); }
```

Specification statements describing insertion of a new node z into a binary search tree.

Mixed interpretation with a model finder (2/4)

```
@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old(this.nodes) + z")
@Modifies("this.root,
           this.nodes.left | _<1> = null,
           this.nodes.right | _<1> = null")

public void insert(Node z) {
  Squander.exe(this, z); }
```

Specification statements describing insertion of a new node z into a binary search tree.

Call to the Squander mixed interpreter ensures that the state of this tree and the node z is mutated so that the insertion specification is satisfied when the insert method returns.

Mixed interpretation with a model finder (2/4)

```
@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old(this.nodes) + z")
@Modifies("this.root,
           this.nodes.left | _<1> = null,
           this.nodes.right | _<1> = null")

public void insert(Node z) {
  Squander.exe(this, z); }
```

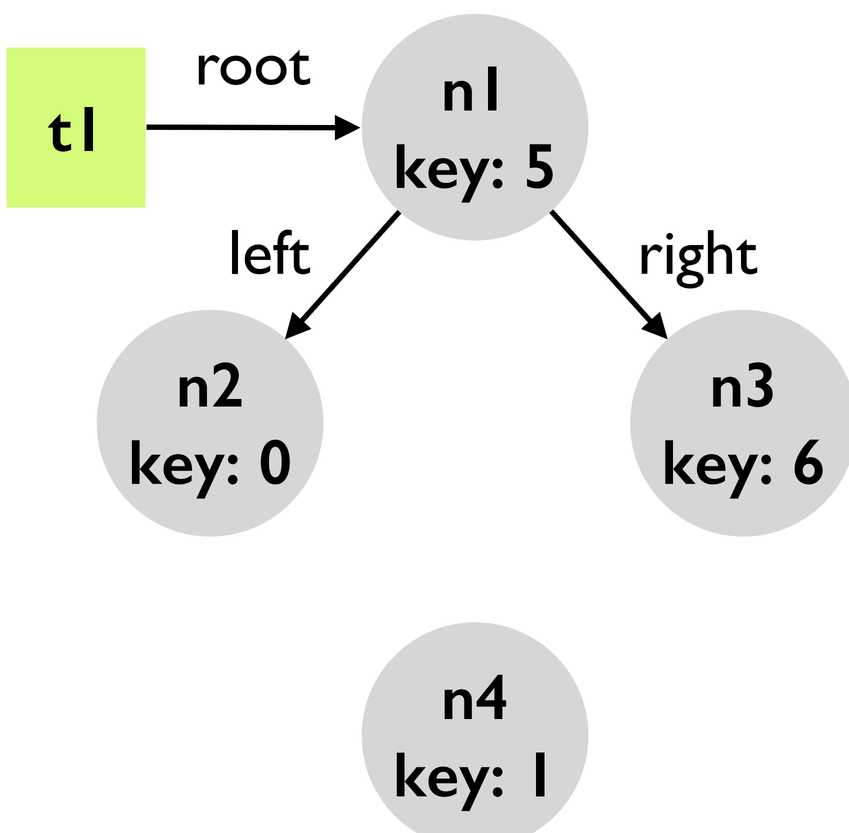
Specification statements describing insertion of a new node z into a binary search tree.

Execution steps:

- Serialize the relevant part of the heap to a universe and bounds
- Use Kodkod to solve the specs against the resulting universe / bounds
- Deserialize the solution (if any) and update the heap accordingly

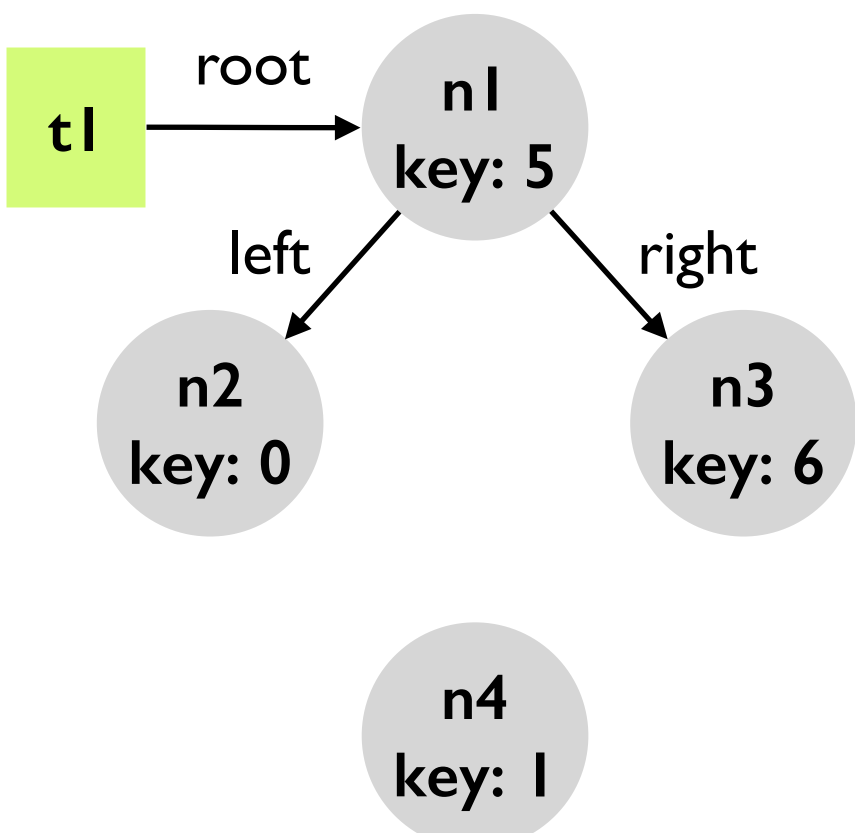
Mixed interpretation with a model finder (3/4)

```
@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old(this.nodes) + z")
@Modifies("this.root,
           this.nodes.left | _<1> = null,
           this.nodes.right | _<1> = null")

public void insert(Node z) {
  Squander.exe(this, z); }
```



Mixed interpretation with a model finder (3/4)

```
@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old(this.nodes) + z")
@Modifies("this.root,
           this.nodes.left | _<1> = null,
           this.nodes.right | _<1> = null")

public void insert(Node z) {
  Squander.exe(this, z); }
```



reachable objects

$T = \{\langle t_1 \rangle\}$

$N = \{\langle n_1 \rangle, \ldots, \langle n_4 \rangle\}$

$null = \{\langle null \rangle\}$

$this = \{\langle t_1 \rangle\}$

$z = \{\langle n_4 \rangle\}$

$ints = \{\langle 0 \rangle, \langle 1 \rangle, \langle 5 \rangle, \langle 6 \rangle\}$

Mixed interpretation with a model finder (3/4)

```
@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old(this.nodes) + z")
@Modifies("this.root,
           this.nodes.left | _<1> = null,
           this.nodes.right | _<1> = null")

public void insert(Node z) {
  Squander.exe(this, z); }
```

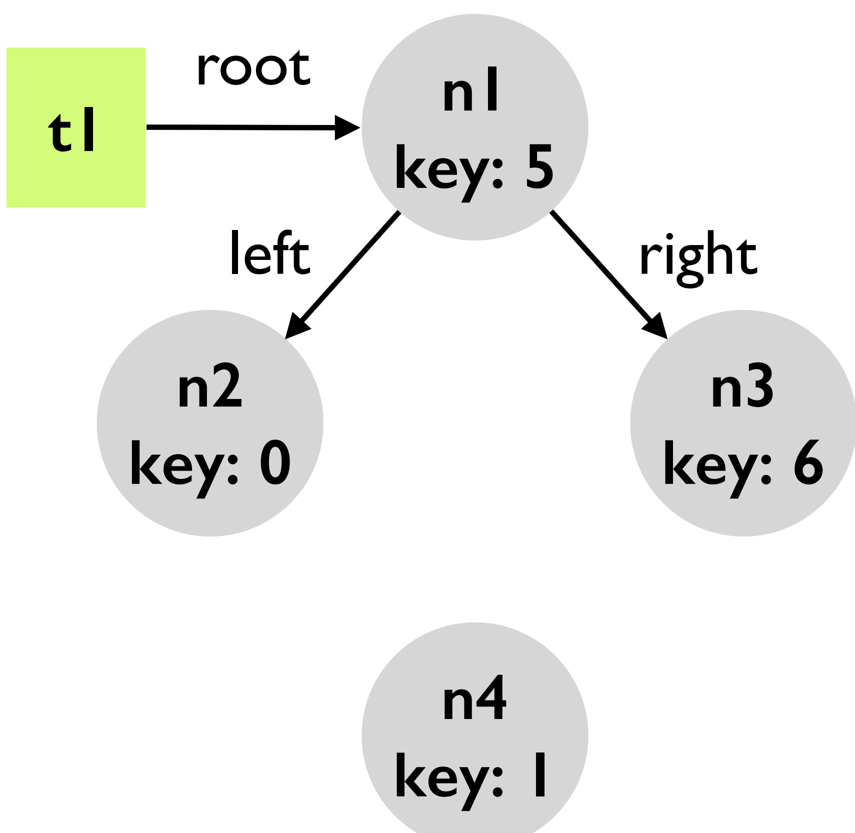
pre-state

$key_{old} = \{\langle n_1, 5\rangle, \ldots, \langle n_4, 1\rangle\}$

$root_{old} = \{\langle t_1, n_1\rangle\}$

$left_{old} = \{\langle n_1, n_2\rangle, \ldots, \langle n_4, null\rangle\}$

$right_{old} = \{\langle n_1, n_3\rangle, \ldots, \langle n_4, null\rangle\}$



reachable objects

$T = \{\langle t_1\rangle\}$

$N = \{\langle n_1\rangle, \ldots, \langle n_4\rangle\}$

$null = \{\langle null\rangle\}$

$this = \{\langle t_1\rangle\}$

$z = \{\langle n_4\rangle\}$

$ints = \{\langle 0\rangle, \langle 1\rangle, \langle 5\rangle, \langle 6\rangle \}$

Mixed interpretation with a model finder (3/4)

```
@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old(this.nodes) + z")
@Modifies("this.root,
           this.nodes.left | _<1> = null,
           this.nodes.right | _<1> = null")

public void insert(Node z) {
  Squander.exe(this, z); }
```

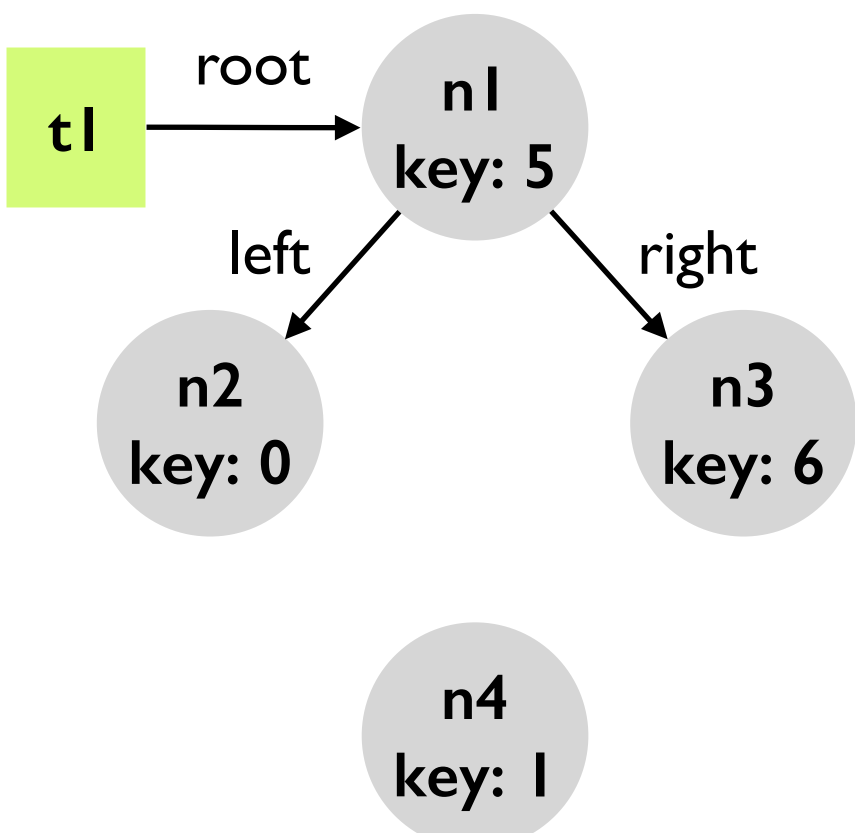
pre-state

$\mathbf{key}_{old} = \{\langle n_1, 5\rangle, \ldots, \langle n_4, 1\rangle\}$

$\mathbf{root}_{old} = \{\langle t_1, n_1\rangle\}$

$\mathbf{left}_{old} = \{\langle n_1, n_2\rangle, \ldots, \langle n_4, null\rangle\}$

$\mathbf{right}_{old} = \{\langle n_1, n_3\rangle, \ldots, \langle n_4, null\rangle\}$



reachable objects

$\mathbf{T} = \{\langle t_1\rangle\}$

$\mathbf{N} = \{\langle n_1\rangle, \ldots, \langle n_4\rangle\}$

$\mathbf{null} = \{\langle null\rangle\}$

$\mathbf{this} = \{\langle t_1\rangle\}$

$\mathbf{z} = \{\langle n_4\rangle\}$

$\mathbf{ints} = \{\langle 0\rangle, \langle 1\rangle, \langle 5\rangle, \langle 6\rangle\}$

post-state

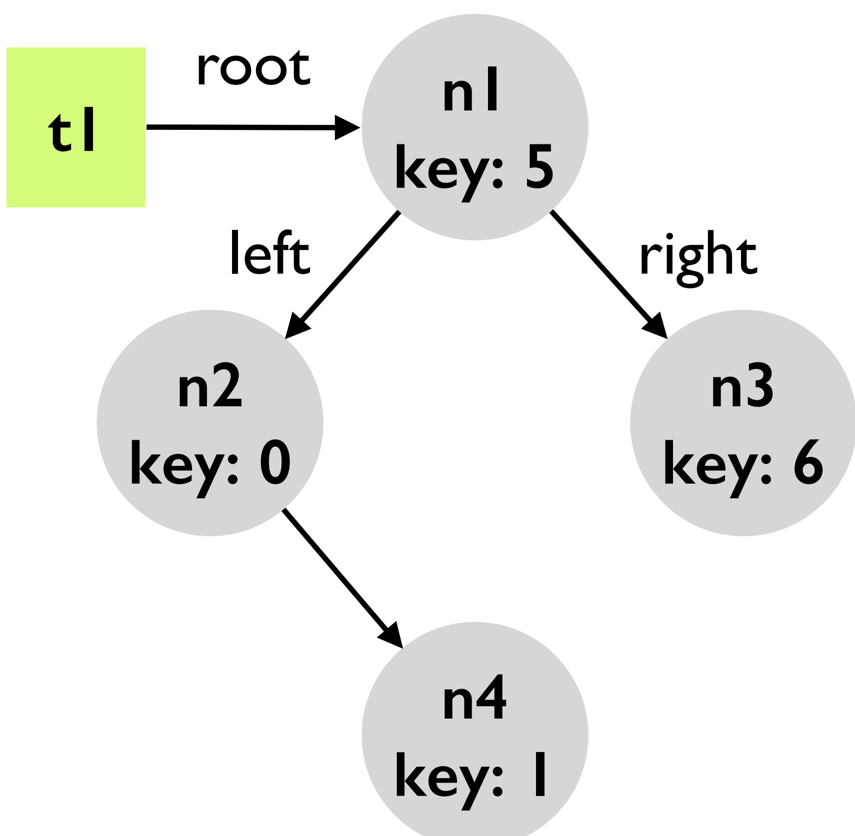
$\{\} \subseteq \mathbf{root} \subseteq \{t_1\} \times \{n_1, \ldots, n_4, null\}$

$\{\langle n_1, n_2\rangle\} \subseteq \mathbf{left} \subseteq \{n_2, n_3, n_4\} \times \{n_1, \ldots, n_4, null\}$

$\{\langle n_1, n_3\rangle\} \subseteq \mathbf{right} \subseteq \{n_2, n_3, n_4\} \times \{n_1, \ldots, n_4, null\}$

Mixed interpretation with a model finder (3/4)

```
@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old(this.nodes) + z")
@Modifies("this.root,
           this.nodes.left | _<1> = null,
           this.nodes.right | _<1> = null")

public void insert(Node z) {
  Squander.exe(this, z); }
```



Mixed interpretation with a model finder (4/4)

```
@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old(this.nodes) + z")
@Modifies("this.root,
           this.nodes.left | _<1> = null,
           this.nodes.right | _<1> = null")

public void insert(Node z) {
  Squander.exe(this, z); }
```

Many more features (e.g., support for obtaining all solutions, support for data abstraction, etc.).

See **Unifying Execution of Declarative and Imperative Code** for details.

Mixed interpretation with a model finder (4/4)

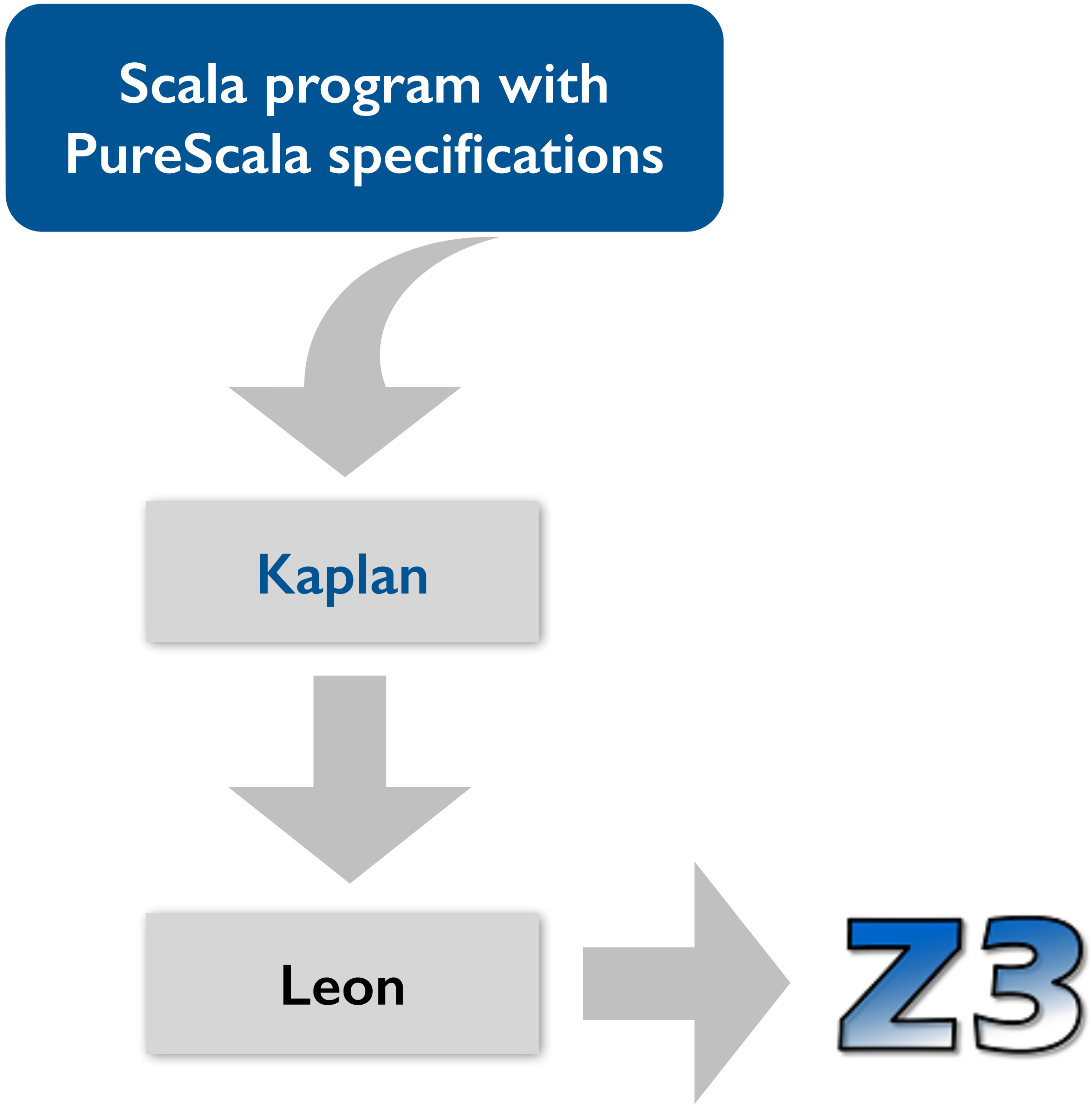
```
@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old(this.nodes) + z")
@Modifies("this.root,
           this.nodes.left | _<1> = null,
           this.nodes.right | _<1> = null")

public void insert(Node z) {
  Squander.exe(this, z); }
```

Incompleteness due to finitization: Squander bounds the number of new instances of a given type that Kodkod can create to satisfy the specification.

Many more features (e.g., support for obtaining all solutions, support for data abstraction, etc.).

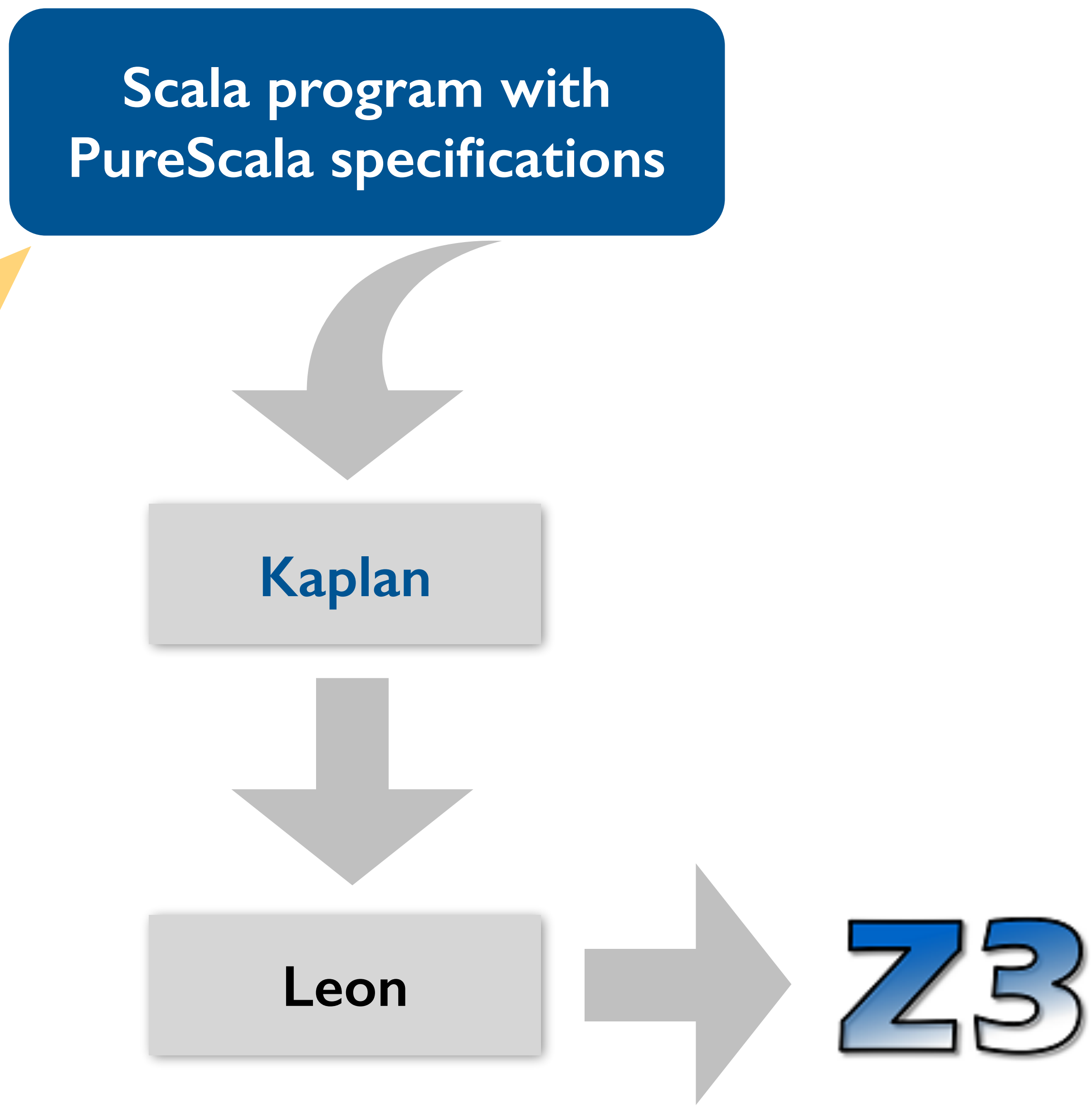
See Unifying Execution of Declarative and Imperative Code for details.

Mixed interpretation with an SMT solver (1/3)



Mixed interpretation with an SMT solver (1/3)

PureScala is a pure, Turing complete subset of Scala that supports unbounded datatypes and arbitrary recursive functions.



Mixed interpretation with an SMT solver (2/3)

```
@spec def noneDivides(from: Int, j: Int) : Boolean {
  from == j ||
  (j % from != 0 && noneDivides(from+1, j))
}

@spec def isPrime(i: Int) : Boolean {
  i >= 2 && noneDivides(2, i)
}

val primes =
((isPrime(_Int)) minimizing
 ((x:Int) => x)).findAll

> primes.take(10).toList
List(2, 3, 4, 5, 11, 17, 19, 23, 29)
```

Mixed interpretation with an SMT solver (2/3)

```
@spec def noneDivides(from: Int, j: Int) : Boolean {
  from == j ||
  (j % from != 0 && noneDivides(from+1, j))
}

@spec def isPrime(i: Int) : Boolean {
  i >= 2 && noneDivides(2, i)
}

val primes =
((isPrime(_Int)) minimizing
 ((x:Int) => x)).findAll

> primes.take(10).toList
List(2, 3, 4, 5, 11, 17, 19, 23, 29)
```

Recursive specification functions. Mutual recursion also allowed.

Mixed interpretation with an SMT solver (2/3)

```
@spec def noneDivides(from: Int, j: Int) : Boolean {
  from == j ||
  (j % from != 0 && noneDivides(from+1, j))
}

@spec def isPrime(i: Int) : Boolean {
  i >= 2 && noneDivides(2, i)
}

val primes =
((isPrime(_Int)) minimizing
 ((x:Int) => x)).findAll

> primes.take(10).toList
List(2, 3, 4, 5, 11, 17, 19, 23, 29)
```

Recursive specification functions. Mutual recursion also allowed.

Call the Kaplan mixed interpreter to obtain the first 10 primes.

Mixed interpretation with an SMT solver (2/3)

```
@spec def noneDivides(from: Int, j: Int) : Boolean {
  from == j ||
  (j % from != 0 && noneDivides(from+1, j))
}

@spec def isPrime(i: Int) : Boolean {
  i >= 2 && noneDivides(2, i)
}

val primes =
((isPrime(_Int)) minimizing
 ((x:Int) => x)).findAll

> primes.take(10).toList
List(2, 3, 4, 5, 11, 17, 19, 23, 29)
```

Recursive specification functions. Mutual recursion also allowed.

Two execution modes:

- Eager: uses Leon to find a satisfying assignment for a given specification.
- Lazy: accumulates specifications, checking their feasibility, until the programmer asks for the *value* of a logical variable. The variable is then frozen (permanently bound) to the returned value.

Call the Kaplan mixed interpreter to obtain the first 10 primes.

Mixed interpretation with an SMT solver (3/3)

```
@spec def noneDivides(from: Int, j: Int) : Boolean {
  from == j ||
  (j % from != 0 && noneDivides(from+1, j))
}

@spec def isPrime(i: Int) : Boolean {
  i >= 2 && noneDivides(2, i)
}

val primes =
((isPrime(_Int)) minimizing
 ((x:Int) => x)).findAll

> primes.take(10).toList
List(2, 3, 4, 5, 11, 17, 19, 23, 29)
```

Many more features (e.g., support for optimization).

See Constraints as Control for details.

Mixed interpretation with an SMT solver (3/3)

```
@spec def noneDivides(from: Int, j: Int) : Boolean {
  from == j ||
  (j % from != 0 && noneDivides(from+1, j))
}

@spec def isPrime(i: Int) : Boolean {
  i >= 2 && noneDivides(2, i)
}

val primes =
((isPrime(_Int)) minimizing
 ((x:Int) => x)).findAll

> primes.take(10).toList
List(2, 3, 4, 5, 11, 17, 19, 23, 29)
```

Incompleteness due to undecidability of PureScala.

Many more features (e.g., support for optimization).

See Constraints as Control for details.

Angelic interpretation with a solver

```
s = 16
r = choose(int)
if (r ≥ 0)
  assert r*r ≤ s < (r+1)*(r+1)
else
  assert r*r ≤ s < (r−1)*(r−1)
```

Angelic interpretation with a solver

```
s = 16
r = choose(int)
if (r ≥ 0)
  assert r*r ≤ s < (r+1)*(r+1)
else
  assert r*r ≤ s < (r−1)*(r−1)
```

Execution steps:

- Translate to the entire program to constraints using either BMC or SE.
- Query the solver for one or all solutions that satisfy the constraints.
- Convert each solution to a valid program trace (represented, e.g., as a sequence of choices made by the oracle in a given execution).

Applications of angelic execution

Applications of angelic execution

Declarative mocking [Samimi et al., ISSTA'13]

Applications of angelic execution

Declarative mocking [Samimi et al., ISSTA'13]

Angelic debugging [Chandra et al., ICSE'11]

Applications of angelic execution

Declarative mocking [Samimi et al., ISSTA'13]

Angelic debugging [Chandra et al., ICSE'11]

Imperative/declarative programming [Milicevic et al., ICSE'11]

Applications of angelic execution

Declarative mocking [Samimi et al., ISSTA'13]

Angelic debugging [Chandra et al., ICSE'11]

Imperative/declarative programming [Milicevic et al., ICSE'11]

Algorithm development [Bodik et al., POPL'10]

Applications of angelic execution

Declarative mocking [Samimi et al., ISSTA'13]

Angelic debugging [Chandra et al., ICSE'11]

Imperative/declarative programming [Milicevic et al., ICSE'11]

Algorithm development [Bodik et al., POPL'10]

Dynamic program repair [Samimi et al., ECOOP'10]

Applications of angelic execution

Declarative mocking [Samimi et al., ISSTA'13]

Angelic debugging [Chandra et al., ICSE'11]

Imperative/declarative programming [Milicevic et al., ICSE'11]

Algorithm development [Bodik et al., POPL'10]

Dynamic program repair [Samimi et al., ECOOP'10]

Test case generation [Khurshid et al., ASE'01]

...

Summary

Today

- Angelic nondeterminism with specifications statements and angelic choice
- Angelic execution with model finders and SMT solvers
- Applications of angelic execution

Next lecture

- Program synthesis