Angelic Execution
Today

Last lecture

• Symbolic execution

Today

• Solvers as angelic oracles
So far, we have used solvers as demonic oracles.
But solvers can also act as angelic oracles

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

1. Definitions
2. Implementations
3. Applications

A trace of P that satisfies S
A programming abstraction

**Angelic non-determinism, two ways**

Angelic choice:

```latex
\text{choose}(T)
```

Specifyation statement:

```latex
x_1, \ldots, x_n \leftarrow [\text{pre}, \text{post}]
```

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

Designed to abstract away the details of backtracking search.

Robert Floyd, 1967

Carroll Morgan, 1988

Robert Floyd, 1967

Carroll Morgan, 1988

A programming abstraction
Angelic non-determinism, two ways

**Angelic choice:**
choose(T)

**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.

Robert Floyd, 1967

Carroll Morgan, 1988

A programming abstraction

A refinement abstraction
Angelic non-determinism, two ways: an example

**Angelic choice:**
```
choose(T)
```

**Specification statement:**
```
\(x_1, \ldots, x_n \leftarrow \text{[pre, post]}\)
```

```
s = 16
r = \text{choose(int)}
\text{if } (r \geq 0)
    \text{assert } r \times r \leq s < (r+1) \times (r+1)
\text{else}
    \text{assert } r \times r \leq s < (r-1) \times (r-1)
```

```
s = 16
r \leftarrow [\text{true,}
    (r \geq 0 \land
        r \times r \leq s < (r+1) \times (r+1)) \lor
    (r < 0 \land
        r \times r \leq s < (r-1) \times (r-1))]
```

“Angelic Interpretation”

“Mixed Interpretation”
Mixed interpretation with a model finder (1/4)

Java program with Alloy specification statements

Squander  PBnJ
Mixed interpretation with a model finder (2/4)

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

```java
@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); }
```

**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); }

---

**pre-state**

- key\textsubscript{old} = \{⟨n\textsubscript{1}, 5⟩, ..., ⟨n\textsubscript{4}, 1⟩\}
- root\textsubscript{old} = \{⟨t\textsubscript{1}, n\textsubscript{1}⟩\}
- left\textsubscript{old} = \{⟨n\textsubscript{1}, n\textsubscript{2}⟩, ..., ⟨n\textsubscript{4}, null⟩\}
- right\textsubscript{old} = \{⟨n\textsubscript{1}, n\textsubscript{3}⟩, ..., ⟨n\textsubscript{4}, null⟩\}

---

**reachable objects**

- T = \{⟨t\textsubscript{1}⟩\}
- N = \{⟨n\textsubscript{1}⟩, ..., ⟨n\textsubscript{4}⟩\}
- null = \{⟨null⟩\}
- this = \{⟨t\textsubscript{1}⟩\}
- z = \{⟨n\textsubscript{4}⟩\}
- ints = \{⟨0⟩, ⟨1⟩, ⟨5⟩, ⟨6⟩ \}

---

**post-state**

- \{\} \subseteq root \subseteq \{t\textsubscript{1}\} \times \{n\textsubscript{1}, ..., n\textsubscript{4}, null\}
- \{⟨n\textsubscript{1}, n\textsubscript{2}⟩\} \subseteq left \subseteq \{n\textsubscript{2}, n\textsubscript{3}, n\textsubscript{4}\} \times \{n\textsubscript{1}, ..., n\textsubscript{4}, null\}
- \{⟨n\textsubscript{1}, n\textsubscript{3}⟩\} \subseteq right \subseteq \{n\textsubscript{2}, n\textsubscript{3}, n\textsubscript{4}\} \times \{n\textsubscript{1}, ..., n\textsubscript{4}, null\}

---

![Diagram](image-url)
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.

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

```haskell
@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, 5, 7, 11, 13, 17, 19, 23, 29)
```

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, 5, 7, 11, 13, 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)

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

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