Angelic Execution

Emina Torlak
emina@cs.washington.edu
Today

Last lecture

• Symbolic execution

Today

• Solvers as angelic oracles

Announcements

• No office hours with Emina on Thursday
• No lecture on Friday
So far, we have used solvers as demonic oracles

Program P  Spec S

Verifier

P \land \neg S

Solver

An input i on which P violates S
But solvers can also act as angelic oracles

A trace of P that satisfies S

\[
P() \{
    y = \text{choose}();
    ...
    \text{assert } S;
\}
\]
But solvers can also act as angelic oracles

\[ P() \{ \\
\quad y = \text{choose}() ; \\
\quad \ldots \\
\quad \text{assert } S ; \\
\} \]

1. Definitions
2. Implementations
3. Applications
Angelic non-determinism, two ways

**Angelic choice:**
choose(T)

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

Robert Floyd, 1967

Carroll Morgan, 1988
Angelic non-determinism, two ways

**Angelic choice:**
\[
\text{choose}(T)
\]

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

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

A programming abstraction
Angelic non-determinism, two ways

**Angelic choice:**
choose(T)

**Specification statement:**
x\(_1\), \(\ldots\), x\(_n\) ← [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:**
\(x_1, \ldots, x_n \leftarrow [\text{pre, post}]\)

\[
\begin{align*}
s &= 16 \\
r &= \text{choose}(\text{int}) \\
\text{if } (r \geq 0) \\
\quad \text{assert } r*r \leq s < (r+1)*(r+1) \\
\text{else} \\
\quad \text{assert } r*r \leq s < (r-1)*(r-1)
\end{align*}
\]
Angelic non-determinism, two ways: an example

**Angelic choice:**
choose(T)

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

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 ← [true,
    (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:**
\[ x_1, ..., x_n \leftarrow \text{[pre, post]} \]

\[
\begin{align*}
    s &= 16 \\
    r &= \text{choose(int)} \\
    \text{if } (r \geq 0) &\text{ then assert } r^2 \leq s < (r+1)(r+1) \\
    \text{else } &\text{ assert } r^2 \leq s < (r-1)(r-1)
\end{align*}
\]

“Angelic Interpretation”

\[
\begin{align*}
    s &= 16 \\
    r &\leftarrow \text{[true,} \\
    &\begin{cases}
    (r \geq 0 \land r^2 \leq s < (r+1)(r+1) &\text{or}
    \\
    (r < 0 \land r^2 \leq s < (r-1)(r-1)) &\text{end}
    \end{cases}
\end{align*}
\]

“Mixed Interpretation”

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

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

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

Java program with Alloy specification statements

Squander

PBnJ
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);
}

*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

Specification statements describing insertion of a new node z into a binary search tree.
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 = \{⟨t₁⟩\}
N = \{⟨n₁⟩, ..., ⟨n₄⟩\}
null = \{⟨null⟩\}
this = \{⟨t₁⟩\}
z = \{⟨n₄⟩\}
isnts = \{⟨0⟩, ⟨1⟩, ⟨5⟩, ⟨6⟩ \}
Mixed interpretation with a model finder (3/4)

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

@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")

pre-state

key\textsubscript{old} = \{\langle n_1, 5 \rangle, \ldots, \langle n_4, 1 \rangle\}
root\textsubscript{old} = \{\langle t_1, n_1 \rangle\}
left\textsubscript{old} = \{\langle n_1, n_2 \rangle, \ldots, \langle n_4, \text{null} \rangle\}
right\textsubscript{old} = \{\langle n_1, n_3 \rangle, \ldots, \langle n_4, \text{null} \rangle\}

reachable objects

\( T = \{\langle t_1 \rangle\} \)
\( N = \{\langle n_1 \rangle, \ldots, \langle n_4 \rangle\} \)
null = \{\langle \text{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)

```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); }
```

### pre-state

- **key**: `{⟨n1, 5⟩, ..., ⟨n4, 1⟩}`
- **root**: `{⟨t1, n1⟩}`
- **left**: `{⟨n1, n2⟩, ..., ⟨n4, null⟩}`
- **right**: `{⟨n1, n3⟩, ..., ⟨n4, null⟩}`

### reachable objects

- **T**: `{⟨t1⟩}`
- **N**: `{⟨n1⟩, ..., ⟨n4⟩}`
- **null**: `{⟨null⟩}`
- **this**: `{⟨t1⟩}`
- **z**: `{⟨n4⟩}`
- **ints**: `{⟨0⟩, ⟨1⟩, ⟨5⟩, ⟨6⟩}`

### post-state

- `{} ⊆ root ⊆ {t1} × {n1, ..., n4, null}`
- `{n1, n2} ⊆ left ⊆ {n2, n3, n4} × {n1, ..., n4, null}`
- `{n1, n3} ⊆ right ⊆ {n2, n3, n4} × {n1, ..., n4, null}`
@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); }

[Diagram of a tree structure with nodes and keys]
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)

Scala program with PureScala specifications

Kaplan

Leon

Z3
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)

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

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

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, 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.

Recursive specification functions. Mutual recursion also allowed.
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)
s = 16
r = \text{choose(int)}
\textbf{if} (r \geq 0)
   \textbf{assert} \ r \cdot r \leq s < (r+1) \cdot (r+1)
\textbf{else}
   \textbf{assert} \ r \cdot r \leq s < (r-1) \cdot (r-1)

\textbf{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

• No lecture Friday

• Next Wednesday: Program synthesis