Bounded Verification

Emina Torlak
emina@cs.washington.edu
Today

Last lecture
• Full functional verification with Dafny, Boogie, and Z3

Today
• Bounded verification with Kodkod (Forge, Miniatur, TACO)

Announcements
• HW3 is out; start early.
The spectrum of program verification tools

Confidence

Cost (programmer effort, time, expertise)

- Static Analysis
- Verification
- Bounded Verification & Symbolic Execution
- Concolic Testing & Whitebox Fuzzing
- Ad-hoc Testing
- Extended Static Checking
The spectrum of program verification tools

- E.g., Dafny, Coq, Leon:
  - support for rich (FOL+) correctness properties
  - high annotation overhead (pre/post conditions, loop invariants, etc.)
  - total correctness

Confidence vs. Cost (programmer effort, time, expertise)
The spectrum of program verification tools

- **Concolic Testing & Whitebox Fuzzing**
- **Ad-hoc Testing**
- **Bounded Verification & Symbolic Execution**
- **Extended Static Checking**
- **Static Analysis**
- **Verification**

### E.g., Astree:
- Small set of fixed properties (e.g., “no null dereferences”)
- No annotations but must deal with false positives
- No false negatives
The spectrum of program verification tools

- **Static Analysis**
  - user-defined assertions supported but optional
  - no annotations
  - some/low false positives
  - false negatives

- **Verification**
  - E.g., Calysto, Saturn:

**Cost (programmer effort, time, expertise)**

- **Confidence**

**Techniques**

- Ad-hoc Testing
- Concolic Testing & Whitebox Fuzzing
- Bounded Verification & Symbolic Execution
- Extended Static Checking
The spectrum of program verification tools

- **Confidence**
- **Cost (programmer effort, time, expertise)**

E.g., CBMC, Miniatur, Forge, TACO, JPF, Klee:
- optional user-defined harnesses, assertions, and/or FOL+ properties
- no/low annotations
- no/low false positives
- false negatives

Verification tools:
- Concolic Testing & Whitebox Fuzzing
- Ad-hoc Testing
- Bounded Verification & Symbolic Execution
- Extended Static Checking
- Static Analysis
The spectrum of program verification tools

- Static Analysis
  - E.g., SAGE, Pex, CUTE, DART:
    - test harnesses and/or user-defined assertions
    - no annotations
    - no false positives
    - false negatives

- Verification

Cost (programmer effort, time, expertise)

Confidence

Bounded Verification & Symbolic Execution

Concolic Testing & Whitebox Fuzzing

Ad-hoc Testing

Extended Static Checking
The spectrum of program verification tools

- Static Analysis
- Verification
- Bounded Verification & Symbolic Execution
- Extended Static Checking
- Concolic Testing & Whitebox Fuzzing
- Ad-hoc Testing

Confidence vs. Cost (programmer effort, time, expertise)
Bounded verification

Bound everything
  • Execution length
  • Bitwidth
  • Heap size (number of objects per type)

Sound counterexamples but no proof
  • Exhaustive search within bounded scope

Empirical “small-scope hypothesis”
  • Bugs usually have small manifestations
Bounded verification by example

```java
class List {
    Node head;

    void reverse() {
        Node near = head;
        Node mid = near.next;
        Node far = mid.next;

        near.next = far;
        while (far != null) {
            mid.next = near;
            near = mid;
            mid = far;
            far = far.next;
        }

        mid.next = near;
        head = mid;
    }
}

class Node {
    Node next;
    String data; }
```
Bounded verification by example

```java
class List {
    Node head;

    void reverse() {
        Node near = head;
        Node mid = near.next;
        Node far = mid.next;

        near.next = far;
        while (far != null) {
            mid.next = near;
            near = mid;
            mid = far;
            far = far.next;
        }

        mid.next = near;
        head = mid;
    }
}

class Node {
    Node next;
    String data;
}
```

Express the property either by writing a test harness or by providing FOL+ contracts.
Specifying contracts: class invariants

class List {
    Node head;

    void reverse() {
        Node near = head;
        Node mid = near.next;
        Node far = mid.next;

        near.next = far;
        while (far != null) {
            mid.next = near;
            near = mid;
            mid = far;
            far = far.next;
        }
        mid.next = near;
        head = mid;
    }
}

class Node {
    Node next;
    String data;
}
class List {
    Node head;

    void reverse() {
        Node near = head;
        Node mid = near.next;
        Node far = mid.next;

        near.next = far;
        while (far != null) {
            mid.next = near;
            near = mid;
            mid = far;
            far = far.next;
        }
        mid.next = near;
        head = mid;
    }
}

class Node {
    Node next;
    String data;
}
Specifying contracts: postconditions

```java
class List {
    Node head;

    void reverse() {
        Node near = head;
        Node mid = near.next;
        Node far = mid.next;

        near.next = far;
        while (far != null) {
            mid.next = near;
            near = mid;
            mid = far;
            far = far.next;
        }
        mid.next = near;
        head = mid;
    }
}

class Node {
    Node next;
    String data;
}
```

@invariant
    no ^next n iden

@requires
    this.head != null and
    this.head.next != null

@ensures
    this.head.*next = this.old(head).*old(next) and
    let N = this.old(head).*old(next) - null |
    next = old(next) ++
    this.old(head)×null ++
    ~(old(next) n N×N)
Specifying contracts: postconditions

```java
class List {
    Node head;

    void reverse() {
        Node near = head;
        Node mid = near.next;
        Node far = mid.next;

        near.next = far;
        while (far != null) {
            mid.next = near;
            near = mid;
            mid = far;
            far = far.next;
        }

        mid.next = near;
        head = mid;
    }
}

class Node {
    Node next;
    String data;
}
```

@invariant Inv(next)

@requires Pre(this, head, next)

@ensures Post(this, old(head), head, old(next), next)
A relational model of memory (heap)

@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

void reverse() {
    Node near = head;
    Node mid = near.next;
    Node far = mid.next;

    near.next = far;
    while (far != null) {
        mid.next = near;
        near = mid;
        mid = far;
        far = far.next;
    }

    mid.next = near;
    head = mid;
}
A relational model of memory (heap)

@invariant \texttt{Inv(next)}
@requires \texttt{Pre(this, head, next)}
@ensures \texttt{Post(this, old(head), head, old(next), next)}

```java
void reverse() {
    Node near = head;
    Node mid = near.next;
    Node far = mid.next;

    near.next = far;
    while (far != \texttt{null}) {
        mid.next = near;
        near = mid;
        mid = far;
        far = far.next;
    }

    mid.next = near;
    head = mid;
}
```

Fields as binary relations

- head : { ⟨this, n2⟩ }, next : { ⟨n2, n1⟩, … }
A relational model of memory (heap)

@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

```java
void reverse() {
    Node near = head;
    Node mid = near.next;
    Node far = mid.next;

    near.next = far;
    while (far != null) {
        mid.next = near;
        near = mid;
        mid = far;
        far = far.next;
    }

    mid.next = near;
    head = mid;
}
```

Fields as binary relations
- `head` : `{ ⟨this, n2⟩ }`, `next` : `{ ⟨n2, n1⟩, ... }`

Types as sets (unary relations)
- `List` : `{ ⟨this⟩ }`, `Node` : `{ ⟨n0⟩, ⟨n1⟩, ⟨n2⟩ }`
A relational model of memory (heap)

@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

void reverse() {
    Node near = head;
    Node mid = near.next;
    Node far = mid.next;

    near.next = far;
    while (far != null) {
        mid.next = near;
        near = mid;
        mid = far;
        far = far.next;
    }

    mid.next = near;
    head = mid;
}
A relational model of memory (heap)

@invariant inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

void reverse() {
    Node near = head;
    Node mid = near.next;
    Node far = mid.next;

    near.next = far;
    while (far != null) {
        mid.next = near;
        near = mid;
        mid = far;
        far = far.next;
    }
    mid.next = near;
    head = mid;
}

Fields as binary relations
- head : {⟨this, n2⟩}, next : { ⟨n2, n1⟩, ... }  

Types as sets (unary relations)
- List: {⟨this⟩}, Node: { ⟨n0⟩, ⟨n1⟩, ⟨n2⟩ }  

Objects as scalars (singleton sets)
- this : {⟨this⟩}, null : { ⟨null⟩ }  

Field read as relational join (.)
- this.head : { ⟨this⟩ }. { ⟨this, n2⟩ } = { ⟨n2⟩ }
A relational model of memory (heap)

@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

void reverse() {
    Node near = head;
    Node mid = near.next;
    Node far = mid.next;

    near.next = far;
    while (far != null) {
        mid.next = near;
        near = mid;
        mid = far;
        far = far.next;
    }

    mid.next = near;
    head = mid;
}
Bounded verification: step 1/4

@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

void reverse() {
    Node near = head;
    Node mid = near.next;
    Node far = mid.next;

    near.next = far;
    while (far != null) {
        mid.next = near;
        near = mid;
        mid = far;
        far = far.next;
    }

    mid.next = near;
    head = mid;
}
@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

```c
void reverse() {
    Node near = head;
    Node mid = near.next;
    Node far = mid.next;
    near.next = far;
    if (far != null) {
        mid.next = near;
        near = mid;
        mid = far;
        far = far.next;
    }
    assume far == null;
    mid.next = near;
    head = mid;
}
```

Execution finitization (inlining, unrolling, SSA)
Bounded verification: step 1/4

@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

```java
void reverse() {
    Node near0 = this.head;
    Node mid0 = near0.next;
    Node far0 = mid0.next;

    next0 = update(next, near0, far0);
    boolean guard = (far0 != null);
    next1 = update(next0, mid0, near0);
    near1 = mid0;
    mid1 = far0;
    far1 = far0.next1;

    near2 = phi(guard, near1, near0);
    mid2 = phi(guard, mid1, mid0);
    far2 = phi(guard, far1, far0);
    next2 = phi(guard, next1, next0);

    assume far2 == null;

    next3 = update(next2, mid2, near2);
    head0 = update(head, this, mid2);
}
```

Execution finitization
(inlining, unrolling, SSA)
Bounded verification: step 2/4

@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

```c
void reverse() {
    Node near0 = this.head;
    Node mid0 = near0.next;
    Node far0 = mid0.next;

    next0 = update(next, near0, far0);
    boolean guard = (far0 != null);
    next1 = update(next0, mid0, near0);
    near1 = mid0;
    mid1 = far0;
    far1 = far0.next1;

    near2 = phi(guard, near1, near0);
    mid2 = phi(guard, mid1, mid0);
    far2 = phi(guard, far1, far0);
    next2 = phi(guard, next1, next0);

    assume far2 == null;

    next3 = update(next2, mid2, near2);
    head0 = update(head, this, mid2);
}
```

Execution finitization (inlining, unrolling, SSA)

Forward VCG

Symbolic interpretation of the code with respect to the relational heap model.
Bounded verification: step 2/4

@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

void reverse() {
    Node near0 = this.head;
    Node mid0 = near0.next;
    Node far0 = mid0.next;

    next0 = update(next, near0, far0);
    boolean guard = (far0 != null);
    next1 = update(next0, mid0, near0);
    near1 = mid0;
    mid1 = far0;
    far1 = far0.next1;

    near2 = phi(guard, near1, near0);
    mid2 = phi(guard, mid1, mid0);
    far2 = phi(guard, far1, far0);
    next2 = phi(guard, next1, next0);

    assume far2 == null;

    next3 = update(next2, mid2, near2);
    head0 = update(head, this, mid2);
}

this ⊆ List ∧ one this ∧
head ⊆ List ↦ (Node ∪ null) ∧
next ⊆ Node ↦ (Node ∪ null) ∧
data ⊆ Node ↦ (String ∪ null) ∧

let near0 = this.head,
    mid0 = near0.next,
    far0 = mid0.next,
next0 = next ++ (near0 × far0),
guard = (far0 != null),
next1 = next0 ++ (mid0 × near0),
near1 = mid0,
    mid1 = far0,
    far1 = far0.next1,
next2 = if guard then near1 else near0,
    mid2 = if guard then mid1 else mid0,
    far2 = if guard then far1 else far0,
    next2 = if guard then next1 else next0,
next3 = next2 ++ (mid2 × near2)
head0 = head ++ (this × mid2) |

far2 = null ∧ Inv(next) ∧ Pre(this, head, next) ∧
    ¬ (Inv(next3) ∧ Post(this, head, head0, next, next3))
Bounded verification: step 3/4

```plaintext
this ⊆ List ∧ one this ∧
head ⊆ List ↦ (Node ∪ null) ∧
next ⊆ Node ↦ (Node ∪ null) ∧
data ⊆ Node ↦ (String ∪ null) ∧

let near₀ = this.head,
    mid₀ = near₀.next,
    far₀ = mid₀.next,

    next₀ = next ++ (near₀ × far₀),
    guard = (far₀ != null),
    next₁ = next₀ ++ (mid₀ × near₀),
    near₁ = mid₀,
    mid₁ = far₀,
    far₁ = far₀.next₁,

    near₂ = if guard then near₁ else near₀,
    mid₂ = if guard then mid₁ else mid₀,
    far₂ = if guard then far₁ else far₀,
    next₂ = if guard then next₁ else next₀,
    next₃ = next₂ ++ (mid₂ × near₂)
head₀ = head ++ (this × mid₂) |

far₂ = null ∧ Inv(next) ∧ Pre(this, head, next) ∧
¬ (Inv(next₃) ∧ Post(this, head, head₀, next, next₃))
```

**Execution finitization** (inlining, unrolling, SSA)

**Forward VCG**

**Heap finitization** (bounds for types, fields)
Bounded verification: step 3/4

this $\subseteq$ List ∧ one this ∧
head $\subseteq$ List $\mapsto$ (Node $\cup$ null) ∧
next $\subseteq$ Node $\mapsto$ (Node $\cup$ null) ∧
data $\subseteq$ Node $\mapsto$ (String $\cup$ null) ∧

let near$_0$ = this.head,
    mid$_0$ = near$_0$.next,
    far$_0$ = mid$_0$.next,
    next$_0$ = next ++ (near$_0$ × far$_0$),
guard = (far$_0$ != null),
    next$_1$ = next$_0$ ++ (mid$_0$ × near$_0$),
    near$_1$ = mid$_0$,
    mid$_1$ = far$_0$,
    far$_1$ = far$_0$.next$_1$,
    near$_2$ = if guard then near$_1$ else near$_0$,
    mid$_2$ = if guard then mid$_1$ else mid$_0$,
    far$_2$ = if guard then far$_1$ else far$_0$,
    next$_2$ = if guard then next$_1$ else next$_0$,
    next$_3$ = next$_2$ ++ (mid$_2$ × near$_2$),
    head$_0$ = head ++ (this × mid$_2$) |

far$_2$ = null ∧ Inv(next) ∧ Pre(this, head, next) ∧
¬ (Inv(next$_3$) ∧ Post(this, head, head$_0$, next, next$_3$))
Bounded verification: step 3/4

this ⊆ List ∧ one this ∧
head ⊆ List ↦ (Node ∪ null) ∧
next ⊆ Node ↦ (Node ∪ null) ∧
data ⊆ Node ↦ (String ∪ null) ∧

let near₀ = this.head,
   mid₀ = near₀.next,
   far₀ = mid₀.next,

next₀ = next ++ (near₀ × far₀),
guard = (far₀ != null),
next₁ = next₀ ++ (mid₀ × near₀),
near₁ = mid₀,
mid₁ = far₀,
far₁ = far₀.next₁,

near₂ = if guard then near₁ else near₀,
mid₂ = if guard then mid₁ else mid₀,
far₂ = if guard then far₁ else far₀,
next₂ = if guard then next₁ else next₀,
next₃ = next₂ ++ (mid₂ × near₂)
head₀ = head ++ (this × mid₂) |

far₂ = null ∧ Inv(next) ∧ Pre(this, head, next) ∧
¬ (Inv(next₃) ∧ Post(this, head, head₀, next, next₃))
Bounded verification: step 3/4

\[
\begin{align*}
\text{this} & \subseteq \text{List} \land \text{one}\ this \land \\
\text{head} & \subseteq \text{List} \mapsto (\text{Node} \cup \text{null}) \land \\
\text{next} & \subseteq \text{Node} \mapsto (\text{Node} \cup \text{null}) \land \\
\text{data} & \subseteq \text{Node} \mapsto (\text{String} \cup \text{null}) \land \\
\text{let near}_0 & = \text{this}.\text{head}, \\
& \quad \text{mid}_0 = \text{near}_0.\text{next}, \\
& \quad \text{far}_0 = \text{mid}_0.\text{next}, \\
& \quad \text{next}_0 = \text{next} \, \mathbf{++} \, (\text{near}_0 \times \text{far}_0), \\
& \quad \text{guard} = (\text{far}_0 \neq \text{null}), \\
& \quad \text{next}_1 = \text{next}_0 \, \mathbf{++} \, (\text{mid}_0 \times \text{near}_0), \\
& \quad \text{near}_1 = \text{mid}_0, \\
& \quad \text{mid}_1 = \text{far}_0, \\
& \quad \text{far}_1 = \text{far}_0.\text{next}_1, \\
& \quad \text{near}_2 = \text{if guard then near}_1 \text{ else near}_0, \\
& \quad \text{mid}_2 = \text{if guard then mid}_1 \text{ else mid}_0, \\
& \quad \text{far}_2 = \text{if guard then far}_1 \text{ else far}_0, \\
& \quad \text{next}_2 = \text{if guard then next}_1 \text{ else next}_0, \\
& \quad \text{next}_3 = \text{next}_2 \, \mathbf{++} \, (\text{mid}_2 \times \text{near}_2) \\
& \quad \text{head}_0 = \text{head} \, \mathbf{++} \, (\text{this} \times \text{mid}_2) \mid \\
& \quad \text{far}_2 = \text{null} \land \text{Inv}(\text{next}) \land \text{Pre}(\text{this}, \text{head}, \text{next}) \land \\
& \quad \neg (\text{Inv}(\text{next}_3) \land \text{Post}(\text{this}, \text{head}, \text{head}_0, \text{next}, \text{next}_3))
\end{align*}
\]

Finite universe of uninterpreted symbols.

\[
\begin{align*}
\{ \text{this}, n0, n1, n2, s0, s1, s2, \text{null} \} \\
\{ \langle \text{null} \rangle \} \subseteq \text{null} \subseteq \{ \langle \text{null} \rangle \} \\
\{ \} \subseteq \text{this} \subseteq \{ \langle \text{this} \rangle \} \\
\{ \} \subseteq \text{List} \subseteq \{ \langle \text{this} \rangle \} \\
\{ \} \subseteq \text{Node} \subseteq \{ \langle n0 \rangle, \langle n1 \rangle, \langle n2 \rangle \} \\
\{ \} \subseteq \text{String} \subseteq \{ \langle s0 \rangle, \langle s1 \rangle, \langle s2 \rangle \} \\
\{ \} \subseteq \text{head} \subseteq \{ \text{this} \} \times \{ n0, n1, n2, \text{null} \} \\
\{ \} \subseteq \text{next} \subseteq \{ n0, n1, n2 \} \times \{ n0, n1, n2, \text{null} \} \\
\{ \} \subseteq \text{data} \subseteq \{ n0, n1, n2 \} \times \{ s0, s1, s2, \text{null} \}
\end{align*}
\]

Upper bound on each relation: tuples it may contain.
Bounded verification: step 3/4

\[
\text{this} \subseteq \text{List} \land \text{one this} \land \\
\text{head} \subseteq \text{List} \mapsto (\text{Node} \cup \text{null}) \land \\
\text{next} \subseteq \text{Node} \mapsto (\text{Node} \cup \text{null}) \land \\
\text{data} \subseteq \text{Node} \mapsto (\text{String} \cup \text{null})
\]

\[
\text{let near}_0 = \text{this.head}, \\
\text{mid}_0 = \text{near}_0.\text{next}, \\
\text{far}_0 = \text{mid}_0.\text{next}, \\
\text{next}_0 = \text{next} + + (\text{near}_0 \times \text{far}_0), \\
\text{guard} = (\text{far}_0 \neq \text{null}), \\
\text{next}_1 = \text{next}_0 + + (\text{mid}_0 \times \text{near}_0), \\
\text{near}_1 = \text{mid}_0, \\
\text{mid}_1 = \text{far}_0, \\
\text{far}_1 = \text{far}_0.\text{next}_1,
\]

\[
\text{near}_2 = \text{if guard then near}_1 \text{ else near}_0, \\
\text{mid}_2 = \text{if guard then mid}_1 \text{ else mid}_0, \\
\text{far}_2 = \text{if guard then far}_1 \text{ else far}_0, \\
\text{next}_2 = \text{if guard then next}_1 \text{ else next}_0, \\
\text{next}_3 = \text{next}_2 + + (\text{mid}_2 \times \text{near}_2), \\
\text{head}_0 = \text{head} + + (\text{this} \times \text{mid}_2) \\
\]

\[
\text{far}_2 = \text{null} \land \text{Inv(next)} \land \text{Pre(this, head, next)} \land \\
\neg (\text{Inv(next}_3) \land \text{Post(this, head, head}_0, \text{next, next}_3))
\]

Finite universe of uninterpreted symbols.

\[
\{ \text{this, n0, n1, n2, s0, s1, s2, null} \}
\]

\[
\{ \langle \text{null} \rangle \} \subseteq \text{null} \subseteq \{ \langle \text{null} \rangle \}
\]

\[
\{ \langle \text{null} \rangle \} \subseteq \text{this} \subseteq \{ \langle \text{this} \rangle \}
\]

\[
\{ \langle \text{null} \rangle \} \subseteq \text{List} \subseteq \{ \langle \text{this} \rangle \}
\]

\[
\{ \langle \text{n0}, \langle \text{n1}, \langle \text{n2}, \{ \langle \text{s0}, \langle \text{s1}, \langle \text{s2} \rangle \} \} \} \} \}
\]

\[
\{ \langle \text{n0}, \langle \text{n1}, \langle \text{n2} \rangle \} \} \subseteq \text{Node} \subseteq \{ \langle \text{n0}, \langle \text{n1}, \langle \text{n2} \rangle \} \}
\]

\[
\{ \langle \text{s0}, \langle \text{s1}, \langle \text{s2} \rangle \} \} \subseteq \text{String} \subseteq \{ \langle \text{s0}, \langle \text{s1}, \langle \text{s2} \rangle \} \}
\]

\[
\{ \langle \text{n0}, \langle \text{n1}, \langle \text{n2} \rangle \} \} \subseteq \text{data} \subseteq \{ \langle \text{n0}, \langle \text{n1}, \langle \text{n2} \rangle \} \} \times \{ \langle \text{s0}, \langle \text{s1}, \langle \text{s2} \rangle \} \}
\]

\[
\{ \langle \text{n0}, \langle \text{n1}, \langle \text{n2} \rangle \} \} \subseteq \text{head} \subseteq \{ \langle \text{n0}, \langle \text{n1}, \langle \text{n2} \rangle \} \} \times \{ \langle \text{n0}, \langle \text{n1}, \langle \text{n2} \rangle \} \}
\]

\[
\{ \langle \text{n0}, \langle \text{n1}, \langle \text{n2} \rangle \} \} \subseteq \text{next} \subseteq \{ \langle \text{n0}, \langle \text{n1}, \langle \text{n2} \rangle \} \} \times \{ \langle \text{n0}, \langle \text{n1}, \langle \text{n2} \rangle \} \}
\]

\[
\{ \langle \text{n0}, \langle \text{n1}, \langle \text{n2} \rangle \} \} \subseteq \text{data} \subseteq \{ \langle \text{n0}, \langle \text{n1}, \langle \text{n2} \rangle \} \} \times \{ \langle \text{s0}, \langle \text{s1}, \langle \text{s2} \rangle \} \}
\]

Lower bound on each relation: tuples it must contain.

Upper bound on each relation: tuples it may contain.
Bounded verification: step 4/4

this ⊆ List ∧ one this ∧
head ⊆ List → (Node ∪ null) ∧
ext ⊆ Node → (Node ∪ null) ∧
data ⊆ Node → (String ∪ null) ∧

let near,
mid0 = near0.next,
far0 = near0.next
next0 = next ++ (near0 × far0)
head0 = head ++ (this × mid0) |

far2 = null ∧ Inv(next) ∧ Pre(this, head, next) ∧
¬ (Inv(next3) ∧ Post(this, head, head0, next, next3))

Execution finitization (inlining, unrolling, SSA)
Forward VCG
Heap finitization (bounds for types, fields)
Solver
Bounded verification: counterexample

```
this
  head
  data: s1
  next
n2
  data: s2
  next
n1
  data: null
  next
n0
  next
null
```
Bounded verification: optimization

Execution finitization (inlining, unrolling, SSA)

Forward VCG

Heap finitization (bounds for types, fields)

Solver

Finitized program after inlining may be huge.

Full inlining is rarely needed to check partial correctness.

Optimization: Counterexample-Guided Abstraction Refinement with Unsatisfiable Cores [Taghdiri, 2004]
From bounded verification to fault localization

@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

void reverse() {
    Node near0 = this.head;
    Node mid0 = near0.next;
    Node far0 = mid0.next;

    next0 = update(next, near0, far0);
    boolean guard = (far0 != null);
    next1 = update(next0, mid0, near0);
    near1 = mid0;
    mid1 = far0;
    far1 = far0.next1;

    near2 = phi(guard, near1, near0);
    mid2 = phi(guard, mid1, mid0);
    far2 = phi(guard, far1, far0);
    next2 = phi(guard, next1, next0);

    assume far2 == null;

    next3 = update(next2, mid2, near2);
    head0 = update(head, this, mid2);
}
From bounded verification to fault localization

Given a buggy program and a failure-triggering input, find a minimal subset of program statements that prevents the execution on the given input from producing a correct output.

```java
@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

void reverse() {
    Node near0 = this.head;
    Node mid0 = near0.next;
    Node far0 = mid0.next;

    next0 = update(next, near0, far0);
    boolean guard = (far0 != null);
    next1 = update(next0, mid0, near0);
    near1 = mid0;
    mid1 = far0;
    far1 = far0.next1;

    near2 = phi(guard, near1, near0);
    mid2 = phi(guard, mid1, mid0);
    far2 = phi(guard, far1, far0);
    next2 = phi(guard, next1, next0);

    assume far2 == null;

    next3 = update(next2, mid2, near2);
    head0 = update(head, this, mid2);
}
```
From bounded verification to fault localization

@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

void reverse() {
    Node near0 = this.head;
    Node mid0 = near0.next;
    Node far0 = mid0.next;

    next0 = update(next, near0, far0);
    boolean guard = (far0 != null);
    next1 = update(next0, mid0, near0);
    near1 = mid0;
    mid1 = far0;
    far1 = far0.next1;

    near2 = phi(guard, near1, near0);
    mid2 = phi(guard, mid1, mid0);
    far2 = phi(guard, far1, far0);
    next2 = phi(guard, next1, next0);

    assume far2 == null;

    next3 = update(next2, mid2, near2);
    head0 = update(head, this, mid2);
}

Given a buggy program and a failure-triggering input, find a minimal subset of program statements that prevents the execution on the given input from producing a correct output.

Introduce additional “indicator” relations into the encoding.
Given a buggy program and a failure-triggering input, find a minimal subset of program statements that prevents the execution on the given input from producing a correct output.

Introduce additional “indicator” relations into the encoding.

The resulting formula, together with the input partial model, is unsatisfiable.
From bounded verification to fault localization

void reverse() {
    Node near0 = this.head;
    Node mid0 = near0.next;
    Node far0 = mid0.next;

    next0 = update(next, near0, far0);
    boolean guard = (far0 != null);
    next1 = update(next0, mid0, near0);
    near1 = mid0;
    mid1 = far0;
    far1 = far0.next1;

    near2 = phi(guard, near1, near0);
    mid2 = phi(guard, mid1, mid0);
    far2 = phi(guard, far1, far0);
    next2 = phi(guard, next1, next0);

    assume far2 == null;

    next3 = update(next2, mid2, near2);
    head0 = update(head, this, mid2);
}

Given a buggy program and a failure-triggering input, find a minimal subset of program statements that prevents the execution on the given input from producing a correct output.

Introduce additional “indicator” relations into the encoding.

The resulting formula, together with the input partial model, is unsatisfiable.

A minimal unsatisfiable core of this formula represents an irreducible cause of the program’s failure to meet the specification.
Fault localization: encoding

```java
void reverse() {
    Node near0 = this.head;
    Node mid0 = near0.next;
    Node far0 = mid0.next;

    next0 = update(next, near0, far0);
    boolean guard = (far0 != null);
    next1 = update(next0, mid0, near0);
    near1 = mid0;
    mid1 = far0;
    far1 = far0.next1;

    near2 = phi(guard, near1, near0);
    mid2 = phi(guard, mid1, mid0);
    far2 = phi(guard, far1, far0);
    next2 = phi(guard, next1, next0);

    assume far2 == null;

    next3 = update(next2, mid2, near2);
    head0 = update(head, this, mid2);
}
```

Start with the encoding for bounded verification.

```java
this ⊆ List ∧ one this ∧
head ⊆ List ↦ (Node ∪ null) ∧
next ⊆ Node ↦ (Node ∪ null) ∧
data ⊆ Node ↦ (String ∪ null) ∧

let near0 = this.head,
    mid0 = near0.next,
    far0 = mid0.next,

    next0 = next ++ (near0 × far0),
    guard = (far0 != null),
    next1 = next0 ++ (mid0 × near0),
    near1 = mid0,
    mid1 = far0,
    far1 = far0.next1,

    near2 = if guard then near1 else near0,
    mid2 = if guard then mid1 else mid0,
    far2 = if guard then far1 else far0,
    next2 = if guard then next1 else next0,
    next3 = next2 ++ (mid2 × near2)

    head0 = head ++ (this × mid2) |

    far2 = null ∧ Inv(next) ∧ Pre(this, head, next) ∧
    ¬ (Inv(next3) ∧ Post(this, head, head0, next, next3))
```
Fault localization: encoding

```java
@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

void reverse() {
    Node near0 = this.head;
    Node mid0 = near0.next;
    Node far0 = mid0.next;

    next0 = update(next, near0, far0);
    boolean guard = (far0 != null);
    next1 = update(next0, mid0, near0);
    near1 = mid0;
    mid1 = far0;
    far1 = far0.next;

    near2 = phi(guard, near1, mid1, far1);
    mid2 = phi(guard, next1, next0);
    next2 = phi(guard, near1, next1, next0);

    assume far2 == null;

    next3 = update(next2, mid2, near2);
    head0 = update(head, this, mid2);
}
```

Introduce fresh relations for source-level expressions.

```
this ⊆ List ∧ one this ∧
head ⊆ List ↔ (Node ∪ null) ∧
next ⊆ Node ↔ (Node ∪ null) ∧
data ⊆ Node ↔ (String ∪ null) ∧
near0 = this.head ∧
mid0 = near0.next ∧
far0 = mid0.next ∧

next0 = next ++ (near0 × far0) ∧
next1 = next0 ++ (mid0 × near0) ∧
near1 = mid0 ∧
mid1 = far0 ∧
far1 = far0.next1 ∧

let guard = (far0 != null),
    near2 = if guard then near1 else near0,
    mid2 = if guard then mid1 else mid0,
    far2 = if guard then far1 else far0,
    next2 = if guard then next1 else next0 |

next3 = next2 ++ (mid2 × near2) ∧
head0 = head ++ (this × mid2) ∧
far2 = null ∧ Inv(next) ∧ Pre(this, head, next) ∧
Inv(next3) ∧ Post(this, head, head0, next, next3)
```
Fault localization: bounds

\[ \text{this} \subseteq \text{List} \land \text{one} \\text{this} \land \]
\[ \text{head} \subseteq \text{List} \mapsto (\text{Node} \cup \text{null}) \land \]
\[ \text{next} \subseteq \text{Node} \mapsto (\text{Node} \cup \text{null}) \land \]
\[ \text{data} \subseteq \text{Node} \mapsto (\text{String} \cup \text{null}) \land \]

\[ \text{near}_0 = \text{this}.\text{head} \land \]
\[ \text{mid}_0 = \text{near}_0.\text{next} \land \]
\[ \text{far}_0 = \text{mid}_0.\text{next} \land \]

\[ \text{next}_0 = \text{next} \mathrel{++} (\text{near}_0 \times \text{far}_0) \land \]
\[ \text{next}_1 = \text{next}_0 \mathrel{++} (\text{mid}_0 \times \text{near}_0) \land \]
\[ \text{near}_1 = \text{mid}_0 \land \]
\[ \text{mid}_1 = \text{far}_0 \land \]
\[ \text{far}_1 = \text{far}_0.\text{next}_1 \land \]

\[ \text{let} \ \text{guard} = (\text{far}_0 \neq \text{null}), \]
\[ \text{near}_2 = \text{if} \ \text{guard} \ \text{then} \ \text{near}_1 \ \text{else} \ \text{near}_0, \]
\[ \text{mid}_2 = \text{if} \ \text{guard} \ \text{then} \ \text{mid}_1 \ \text{else} \ \text{mid}_0, \]
\[ \text{far}_2 = \text{if} \ \text{guard} \ \text{then} \ \text{far}_1 \ \text{else} \ \text{far}_0, \]
\[ \text{next}_2 = \text{if} \ \text{guard} \ \text{then} \ \text{next}_1 \ \text{else} \ \text{next}_0 \mid \]

\[ \text{next}_3 = \text{next}_2 \mathrel{++} (\text{mid}_2 \times \text{near}_2) \land \]
\[ \text{head}_0 = \text{head} \mathrel{++} (\text{this} \times \text{mid}_2) \land \]
\[ \text{far}_2 = \text{null} \land \text{Inv}(\text{next}) \land \text{Pre}(\text{this}, \text{head}, \text{next}) \land \]
\[ \text{Inv}(\text{next}_3) \land \text{Post}(\text{this}, \text{head}, \text{head}_0, \text{next}, \text{next}_3) \]

\[ \{ \text{this}, \text{n0}, \text{n1}, \text{n2}, \text{s0}, \text{s1}, \text{s2}, \text{null} \} \]

\[ \text{null} = \{ \langle \text{null} \rangle \} \]
\[ \text{this} = \{ \langle \text{this} \rangle \} \]
\[ \text{List} = \{ \langle \text{this} \rangle \} \]
\[ \text{Node} = \{ \langle \text{n0} \rangle, \langle \text{n1} \rangle, \langle \text{n2} \rangle \} \]
\[ \text{String} = \{ \langle \text{s1} \rangle, \langle \text{s2} \rangle \} \]

\[ \text{head} = \{ \langle \text{this}, \text{n2} \rangle \} \]
\[ \text{next} = \{ \langle \text{n2}, \text{n1} \rangle, \langle \text{n1}, \text{n0} \rangle, \langle \text{n0}, \text{null} \rangle \} \]
\[ \text{data} = \{ \langle \text{n2}, \text{s1} \rangle, \langle \text{n1}, \text{s2} \rangle, \langle \text{n0}, \text{null} \rangle \} \]

\[ \{ \} \subseteq \text{head}_0 \subseteq \{ \langle \text{this} \rangle \} \times \{ \langle \text{n0}, \text{n1}, \text{n2}, \text{null} \rangle \} \]
\[ \{ \} \subseteq \text{next}_0 \subseteq \{ \langle \text{n0}, \text{n1}, \text{n2} \rangle \} \times \{ \langle \text{n0}, \text{n1}, \text{n2}, \text{null} \rangle \} \]
\[ \{ \} \subseteq \text{next}_1 \subseteq \{ \langle \text{n0}, \text{n1}, \text{n2} \rangle \} \times \{ \langle \text{n0}, \text{n1}, \text{n2}, \text{null} \rangle \} \]
\[ \{ \} \subseteq \text{next}_3 \subseteq \{ \langle \text{n0}, \text{n1}, \text{n2} \rangle \} \times \{ \langle \text{n0}, \text{n1}, \text{n2}, \text{null} \rangle \} \]
\[ \{ \} \subseteq \text{near}_0 \subseteq \{ \langle \text{n0}, \text{n1}, \text{n2}, \text{null} \rangle \} \]
\[ \{ \} \subseteq \text{near}_1 \subseteq \{ \langle \text{n0}, \text{n1}, \text{n2}, \text{null} \rangle \} \]
\[ \{ \} \subseteq \text{near}_2 \subseteq \{ \langle \text{n0}, \text{n1}, \text{n2}, \text{null} \rangle \} \]
\[ \{ \} \subseteq \text{near}_3 \subseteq \{ \langle \text{n0}, \text{n1}, \text{n2}, \text{null} \rangle \} \]
\[ \{ \} \subseteq \text{mid}_0 \subseteq \{ \langle \text{n0}, \text{n1}, \text{n2}, \text{null} \rangle \} \]
\[ \{ \} \subseteq \text{mid}_1 \subseteq \{ \langle \text{n0}, \text{n1}, \text{n2}, \text{null} \rangle \} \]
\[ \{ \} \subseteq \text{far}_0 \subseteq \{ \langle \text{n0}, \text{n1}, \text{n2}, \text{null} \rangle \} \]
\[ \{ \} \subseteq \text{far}_1 \subseteq \{ \langle \text{n0}, \text{n1}, \text{n2}, \text{null} \rangle \} \]
Fault localization: minimal unsat core

\[
\begin{align*}
\text{this} & \subseteq \text{List} \land \text{one this} \land \\
\text{head} & \subseteq \text{List} \mapsto (\text{Node} \cup \text{null}) \land \\
\text{next} & \subseteq \text{Node} \mapsto (\text{Node} \cup \text{null}) \land \\
\text{data} & \subseteq \text{Node} \mapsto (\text{String} \cup \text{null}) \land \\
\text{near} & = 0 \land \\
\text{mid} & = 0 \land \\
\text{far} & = 0 \land \\
\text{next} & = 0 \land \\
\text{next} & = 0 \land \\
\text{next} & = 0 \land \\
\text{this} & = \{ \text{null} \} \\
\text{this} & = \{ \text{this} \} \\
\text{List} & = \{ \text{this} \} \\
\text{Node} & = \{ \text{n0}, \text{n1}, \text{n2} \} \\
\text{String} & = \{ \text{s1}, \text{s2} \} \\
\text{near} & = \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \} \\
\text{mid} & = \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \} \\
\text{far} & = \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \} \\
\text{next} & = \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \} \\
\text{next} & = \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \} \\
\text{near} & = \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \} \\
\text{mid} & = \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \} \\
\text{far} & = \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \} \\
\text{next} & = \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \} \\
\text{head} & = \{ \text{this, n2} \} \\
\text{next} & = \{ \text{n2, n1}, \text{n2, n0}, \text{n0, null} \} \\
\text{data} & = \{ \text{n2, s1}, \text{n1, s2}, \text{n0, null} \} \\

\end{align*}
\]
Fault localization: minimal unsat core

```java
@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

void reverse() {
    Node near0 = this.head;
    Node mid0 = near0.next;
    Node far0 = mid0.next;

    next0 = update(next, near0, far0);
    boolean guard = (far0 != null);
    next1 = update(next0, mid0, near0);
    near1 = mid0;
    mid1 = far0;
    far1 = far0.next1;

    near2 = phi(guard, near1, near0);
    mid2 = phi(guard, mid1, mid0);
    far2 = phi(guard, far1, far0);
    next2 = phi(guard, next1, next0);

    assume far2 == null;

    next3 = update(next2, mid2, near2);
    head0 = update(head, this, mid2);
}
```
Fault localization: minimal unsat core

@invariant Inv(next)
@requires Pre(this, head, next)
@ensures Post(this, old(head), head, old(next), next)

void reverse() {
    Node near = head;
    Node mid = near.next;
    Node far = mid.next;

    near.next = far;
    while (far != null) {
        mid.next = near;
        near = mid;
        mid = far;
        far = far.next;
    }

    mid.next = near;
    head = mid;
}
Summary

Today

• Bounded verification
  • A relational model of the heap
  • CEGAR with unsat cores
  • Fault localization

Next lecture

• Symbolic execution and concolic testing