Advanced encoding of programs
CS294: Program Synthesis for Everyone

Ras Bodik
Emina Torlak
Division of Computer Science
University of California, Berkeley
We show, by example, how to encode the four programming problems from Lecture 2 in bounded relational logic and solve them with Kodkod.
We show, by example, how to encode the four programming problems from Lecture 2 in bounded relational logic and solve them with Kodkod.

- An eager SAT-based solver optimized for reasoning over finite domains, as used in inductive synthesis

<table>
<thead>
<tr>
<th>ENCODING</th>
<th>SOLVER</th>
<th>TIME (SEC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>QF_AUFLIA</td>
<td>CVC3</td>
<td>&gt;600</td>
</tr>
<tr>
<td></td>
<td>Z3</td>
<td>159</td>
</tr>
<tr>
<td>QF_AUFBV</td>
<td>Boolector</td>
<td>409</td>
</tr>
<tr>
<td></td>
<td>Z3</td>
<td>287</td>
</tr>
<tr>
<td></td>
<td>CVC3</td>
<td>119</td>
</tr>
<tr>
<td>QF_AUFBV (non extensional)</td>
<td>CVC3</td>
<td>&gt;600</td>
</tr>
<tr>
<td></td>
<td>Boolector</td>
<td>&gt;600</td>
</tr>
<tr>
<td></td>
<td>Z3</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>STP</td>
<td>11</td>
</tr>
<tr>
<td>REL_BV</td>
<td>Rosette</td>
<td>9</td>
</tr>
<tr>
<td>REL</td>
<td>Kodkod</td>
<td>5</td>
</tr>
</tbody>
</table>

Lecture 3
We show, by example, how to encode the four programming problems from Lecture 2 in bounded relational logic and solve them with Kodkod.

- An *eager* SAT-based solver optimized for reasoning over *finite domains*, as used in inductive synthesis
- Logic designed for easy modeling of graph-like structures such as *heaps* and *linked data structures*
- First-order logic with relational algebra, transitive closure, bitvector arithmetic and partial models

<table>
<thead>
<tr>
<th>ENCODING</th>
<th>SOLVER</th>
<th>TIME (SEC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>QF_AUFLIA</td>
<td>CVC3</td>
<td>&gt;600</td>
</tr>
<tr>
<td></td>
<td>Z3</td>
<td>159</td>
</tr>
<tr>
<td>QF_AUFBV</td>
<td>Boolector</td>
<td>409</td>
</tr>
<tr>
<td></td>
<td>Z3</td>
<td>287</td>
</tr>
<tr>
<td></td>
<td>CVC3</td>
<td>119</td>
</tr>
<tr>
<td>QF_AUFBV (non extensional)</td>
<td>CVC3</td>
<td>&gt;600</td>
</tr>
<tr>
<td></td>
<td>Boolector</td>
<td>&gt;600</td>
</tr>
<tr>
<td></td>
<td>Z3</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>STP</td>
<td>11</td>
</tr>
<tr>
<td>REL_BV</td>
<td>Rosette</td>
<td>9</td>
</tr>
<tr>
<td>REL</td>
<td>Kodkod</td>
<td>5</td>
</tr>
</tbody>
</table>
We show, by example, how to encode the four programming problems from Lecture 2 in bounded relational logic and solve them with Kodkod.

- An *eager* SAT-based solver optimized for reasoning over *finite domains*, as used in inductive synthesis
- Logic designed for easy modeling of graph-like structures such as heaps and linked data structures
- First-order logic with relational algebra, transitive closure, bitvector arithmetic and partial models
- Provides *minimal unsatisfiable cores* as well as models, enabling both synthesis and diagnosis of synthesis failures

<table>
<thead>
<tr>
<th>ENCODING</th>
<th>SOLVER</th>
<th>TIME (SEC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>QF_AUFLIA</td>
<td>CVC3</td>
<td>&gt;600</td>
</tr>
<tr>
<td></td>
<td>Z3</td>
<td>159</td>
</tr>
<tr>
<td>QF_AUFBV</td>
<td>Boolector</td>
<td>409</td>
</tr>
<tr>
<td></td>
<td>Z3</td>
<td>287</td>
</tr>
<tr>
<td></td>
<td>CVC3</td>
<td>119</td>
</tr>
<tr>
<td>QF_AUFBV (non extensional)</td>
<td>CVC3</td>
<td>&gt;600</td>
</tr>
<tr>
<td></td>
<td>Boolector</td>
<td>&gt;600</td>
</tr>
<tr>
<td></td>
<td>Z3</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>STP</td>
<td>11</td>
</tr>
<tr>
<td>REL_BV</td>
<td>Rosette</td>
<td>9</td>
</tr>
<tr>
<td>REL</td>
<td>Kodkod</td>
<td>5</td>
</tr>
</tbody>
</table>
We show, by example, how to encode the four programming problems from Lecture 2 in bounded relational logic and solve them with Kodkod.

- An eager SAT-based solver optimized for reasoning over finite domains, as used in inductive synthesis
- Logic designed for easy modeling of graph-like structures such as heaps and linked data structures
- First-order logic with relational algebra, transitive closure, bitvector arithmetic and partial models
- Provides minimal unsatisfiable cores as well as models, enabling both synthesis and diagnosis of synthesis failures

Next lecture: why small languages are useful

Subsequent lecture: project problem statements (what we want to synthesize)
KODKOD
a constraint solver for relational logic

about kodkod
Kodkod is an efficient SAT-based constraint solver for first order logic with relations, transitive closure, bit-vector arithmetic, and partial models. It provides analyses for both satisfiable and unsatisfiable problems: a finite model finder for the former and a minimal unsatisfiable core extractor for the latter. Kodkod is used in a wide range of applications, including code checking, test-case generation, declarative execution, declarative configuration, and lightweight analysis of Alloy, UML, and Isabelle/HOL.

Designed as a plugin component that can be easily incorporated into other tools, Kodkod provides a clean Java interface for constructing, manipulating, and solving constraints. The implementation is open-source and available for download under the MIT license. The source code is extensively documented, and the distribution includes many examples demonstrating the use of the Kodkod API.

contact
Emina Torlak (emina at alum.mit.edu)
some applications of kodkod

checking theorems & designs
- Alloy4 (Alloy), Nitpick (Isabelle/HOL), ProB (B, Event-B, Z and TLA\textsuperscript{+}), ExUML (UML)

checking code & memory models
- Forge, Karun, Miniatur, TACO, MemSAT

declarative programming, fault recovery & data structure repair
- Squander, PBnJ, Tarmeem, Cobbler

declarative configuration
- ConfigAssure (networks), Margrave (policies)

test-case generation
- Kesit, Whispec
example: reversing a linked list

class List {
    Node head;

    void reverse() {

    }
}

class Node {
    Node next;
    String data;
}
example: reversing a linked list

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;
}
invariants, pre and post conditions

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;
}
invariants, pre and post conditions

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;
}
invariants, pre and post conditions

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;
}
invariants, pre and post conditions

```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;
}
```

@requires Pre(this, head, next)

@ensures Post(this, old(head), head, old(next), next)

@invariant Inv(next)
a relational view of the 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 view of the heap

@invariant \text{Inv}(\text{next})
@requires \text{Pre}(\text{this}, \text{head}, \text{next})
@ensures \text{Post}(\text{this}, \text{old(head)}, \text{head}, \text{old(next)}, \text{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 view of the heap

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

\textbf{void} \text{reverse}() \{ \\
\quad \text{Node near} = \text{head}; \\
\quad \text{Node mid} = \text{near}.\text{next}; \\
\quad \text{Node far} = \text{mid}.\text{next}; \\
\quad \text{near}.\text{next} = \text{far}; \\
\quad \textbf{while} (\text{far} \neq \text{null}) \{ \\
\qquad \text{mid}.\text{next} = \text{near}; \\
\qquad \text{near} = \text{mid}; \\
\qquad \text{mid} = \text{far}; \\
\qquad \text{far} = \text{far}.\text{next}; \\
\quad \}
\quad \text{mid}.\text{next} = \text{near}; \\
\quad \text{head} = \text{mid}; \\
\}

fields as binary relations
\begin{itemize}
\item \textbf{head} \equiv \{ <\text{this}, n2> \}, \textbf{next} \equiv \{ <n2, n1>, \ldots \}
\end{itemize}

types as sets (unary relations)
\begin{itemize}
\item \textbf{List} \equiv \{ <\text{this}> \}, \textbf{Node} \equiv \{ <n0>, <n1>, <n2> \}
\end{itemize}
a relational view of the 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> }

objects as scalars (singleton unary relations)
- `this` ≡ { <this> }, `null` ≡ { <null> }
a relational view of the 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> }

objects as scalars (singleton unary relations)
- this ≡ { <this> }, null ≡ { <null> }

field access as relational join (.)
- this.head ≡ { <this> } . { <this, n2> } = { <n2> }

- fields as binary relations
- types as sets (unary relations)
- objects as scalars (singleton unary relations)
- field access as relational join (.)
a relational view of the 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> }

objects as scalars (singleton unary relations)
- `this` ≡ { <this> }, `null` ≡ { <null> }

field access as relational join (.)
- `this.head` ≡ { <this> } . { <this, n2> } = { <n2> }

field update as relational override (++)
- `this.head = null` ≡ head ++ (this × null) = { <this, n2> } ++ { <this, null> } = { <this, null> }

---

![Diagram of a relational view of the heap with nodes and links representing data and connections.](image-url)
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;
}
code checking with kodkod

@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;
    if (far != null) {
        mid.next = near;
        near = mid;
        mid = far;
        far = far.next;
    }
    assume far == null;
    mid.next = near;
    head = mid;
}
```

finitize loops

› e.g., unwind once
code checking with kodkod

@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;
}
```

finitize loops
› e.g., unwind once

convert to SSA
› SSA for both locals and fields
code checking with kodkod

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

void reverse() {
    Node near = this.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;
}
```

finitize loops
- e.g., unwind once

convert to SSA
- SSA for both locals and fields

encode program semantics in relational logic
void reverse() {
    Node near₀ = this.head;
    Node mid₀ = near₀.next;
    Node far₀ = mid₀.next;

    // Update variables
    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);
}
**code checking with kodkod**

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

    mid0.next = near0;
    head0 = mid0;
}
```

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

- **finitize loops**
  - e.g., unwind once
- **convert to SSA**
  - SSA for both locals and fields
- **encode program semantics in relational logic**
- **specify analysis bounds**
- **for details see**
  - Forge [Dennis 2006, Dennis 2009]
  - Miniatur [Dolby et al. 2007]
  - MemSAT [Torlak et al. 2010]
translating code to relational logic

@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);
}
translating code to relational logic

@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);
}
translating code to relational logic

```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);
}
```

this ⊆ List ∧ one this ∧
head ⊆ List × (Node ∪ null) ∧ (∀ l: List | one l.head)
next ⊆ Node → (Node ∪ null) ∧
data ⊆ Node → (String ∪ null) ∧

encode the post-state relations in terms of the pre-state, using relational joins and overrides

use the pre- and post-state relations to encode invariants, preconditions, and negated postconditions
translating code to relational logic

```plaintext
@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,

    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))
```
specifying analysis bounds

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))

finite universe of uninterpreted elements
› e.g., 1 List object, 3 of everything else

upper bound on each relation
› set of tuples, drawn from the universe, that the relation may contain

lower bound on each relation
› set of tuples, drawn from the universe, that the relation must contain
› lower bounds collectively form a partial model
specifying analysis bounds

\[
\begin{align*}
\text{this} & \subseteq \text{List} \land \text{one} \text{ this} \\
\text{head} & \subseteq \text{List} \rightarrow (\text{Node} \cup \text{null}) \\
\text{next} & \subseteq \text{Node} \rightarrow (\text{Node} \cup \text{null}) \\
\text{data} & \subseteq \text{Node} \rightarrow (\text{String} \cup \text{null})
\end{align*}
\]

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

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

\[
\begin{align*}
\{ \text{this, n0, n1, n2, s0, s1, s2, null } \} \\
\{ <\text{null}> \} \subseteq \text{null} \subseteq \{ <\text{null}> \}
\end{align*}
\]
specifying analysis bounds

\[
\begin{align*}
\text{this} & \subseteq \text{List} \land \text{one this} \\
\text{head} & \subseteq \text{List} \rightarrow (\text{Node} \cup \text{null}) \\
\text{next} & \subseteq \text{Node} \rightarrow (\text{Node} \cup \text{null}) \\
\text{data} & \subseteq \text{Node} \rightarrow (\text{String} \cup \text{null}) \\
\text{let} \quad & \\
\text{nearer0} & = \text{this}.\text{head}, \\
\text{mid0} & = \text{nearer0} . \text{next}, \\
\text{far0} & = \text{mid0} . \text{next}, \\
\text{next0} & = \text{next} ++ (\text{nearer0} \times \text{far0}), \\
\text{guard} & = (\text{far0} \neq \text{null}), \\
\text{next1} & = \text{next0} ++ (\text{mid0} \times \text{nearer0}), \\
\text{nearer1} & = \text{mid0}, \\
\text{mid1} & = \text{far0}, \\
\text{far1} & = \text{far0} . \text{next1}, \\
\text{nearer2} & = \text{if guard then nearer1 else nearer0}, \\
\text{mid2} & = \text{if guard then mid1 else mid0}, \\
\text{far2} & = \text{if guard then far1 else far0}, \\
\text{next2} & = \text{if guard then next1 else next0}, \\
\text{next3} & = \text{next2} ++ (\text{mid2} \times \text{nearer2}) \\
\text{head0} & = \text{head} ++ (\text{this} \times \text{mid2}) \\
\text{far2} & = \text{null} \land \text{Inv(next)} \land \text{Pre(this, head, next)} \land \\
\neg (\text{Inv(next3)} \land \text{Post(this, head, head0, next, next3)}) \\
\end{align*}
\]

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

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,
ext₂ = if guard then next₁ else next₀,
ext₃ = next₂ ++ (mid₂ × near₂)
head₀ = head ++ (this × mid₂) |

far₂ = null ∧ Inv(next) ∧ Pre(this, head, next) ∧
− (Inv(next₃) ∧ Post(this, head, head₀, next, next₃))
specifying analysis bounds

\[
\begin{align*}
this &\subseteq \text{List} \land \text{one this} \\
\text{head} &\subseteq \text{List} \rightarrow (\text{Node} \cup \text{null}) \\
\text{next} &\subseteq \text{Node} \rightarrow (\text{Node} \cup \text{null}) \\
\text{data} &\subseteq \text{Node} \rightarrow (\text{String} \cup \text{null}) \\
\end{align*}
\]

let \(\text{near}_0 = \text{this}.\text{head},\)
\(\text{mid}_0 = \text{near}_0.\text{next},\)
\(\text{far}_0 = \text{mid}_0.\text{next},\)

\[
\begin{align*}
\text{next}_0 &= \text{next} \quad (\text{near}_0 \times \text{far}_0), \\
\text{guard} &= (\text{far}_0 \neq \text{null}), \\
\text{next}_1 &= \text{next}_0 \quad (\text{mid}_0 \times \text{near}_0), \\
\text{near}_1 &= \text{mid}_0, \\
\text{mid}_1 &= \text{far}_0, \\
\text{far}_1 &= \text{far}_0.\text{next}_1, \\
\text{near}_2 &= \text{if guard then near}_1 \text{ else near}_0, \\
\text{mid}_2 &= \text{if guard then mid}_1 \text{ else mid}_0, \\
\text{far}_2 &= \text{if guard then far}_1 \text{ else far}_0, \\
\text{next}_2 &= \text{if guard then next}_1 \text{ else next}_0, \\
\text{next}_3 &= \text{next}_2 \quad (\text{mid}_2 \times \text{near}_2) \\
\text{head}_0 &= \text{head} \quad (\text{this} \times \text{mid}_2), \\
\text{far}_2 &= \text{null} \land \text{Inv(next)} \land \text{Pre(this, head, next)} \land \\
&\quad \lnot (\text{Inv(next}_3) \land \text{Post(this, head, head}_0, \text{next}, \text{next}_3))
\end{align*}
\]

\[
\begin{align*}
\text{universe} &= \{ \text{this, n0, n1, n2, s0, s1, s2, null} \} \\
\text{null} &= \{ <\text{null}> \} \\
\{ \} &\subseteq \text{this} \subseteq \{ <\text{this}> \} \\
\{ \} &\subseteq \text{List} \subseteq \{ <\text{this}> \} \\
\{ \} &\subseteq \text{Node} \subseteq \{ <\text{n0}, <\text{n1}, <\text{n2}> \} \\
\{ \} &\subseteq \text{String} \subseteq \{ <\text{s0}, <\text{s1}, <\text{s2}> \} \\
\{ \} &\subseteq \text{head} \subseteq \{ \text{this} \} \times \{ \text{n0, n1, n2, null} \} \\
\{ \} &\subseteq \text{next} \subseteq \{ \text{n0, n1, n2} \} \times \{ \text{n0, n1, n2, null} \} \\
\{ \} &\subseteq \text{data} \subseteq \{ \text{n0, n1, n2} \} \times \{ \text{s0, s1, s2, null} \}
\end{align*}
\]
code checking demo
a bug! what to do about it?
data repair: fallback to the specification

@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);
}
```
data repair: fallback to the specification

@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);
}
```

given a (valid) pre-state and a bad post-state at runtime, solve for a post-state that satisfies the specification and continue executing
data repair: fallback to the specification

@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);
}
```

given a (valid) pre-state and a bad post-state at runtime, solve for a post-state that satisfies the specification and continue executing
don’t solve for the pre-state; express it as a partial model
data repair: fallback to the specification

@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);
}
data repair using partial models

\[ \text{this} \subseteq \text{List} \land \text{one this} \land \]
\[ \text{head} \subseteq \text{List} \rightarrow (\text{Node} \cup \text{null}) \land \]
\[ \text{next} \subseteq \text{Node} \rightarrow (\text{Node} \cup \text{null}) \land \]
\[ \text{data} \subseteq \text{Node} \rightarrow (\text{String} \cup \text{null}) \land \]
\[ \text{head}_0 \subseteq \text{List} \rightarrow (\text{Node} \cup \text{null}) \land \]
\[ \text{next}_3 \subseteq \text{Node} \rightarrow (\text{Node} \cup \text{null}) \land \]
\[ \text{Inv}(\text{next}) \land \text{Pre}((\text{this}, \text{head}, \text{next}) \land \]
\[ \text{Inv}(\text{next}_3) \land \text{Post}((\text{this}, \text{head}, \text{head}_0, \text{next}, \text{next}_3) \]

\[ \{ \text{this}, \text{n0}, \text{n1}, \text{n2}, \text{s0}, \text{s1}, \text{s2}, \text{null} \} \]
\[ \text{null} = \{ \langle \text{null} \rangle \} \]
\[ \text{this} = \{ \langle \text{this} \rangle \} \]
\[ \text{List} = \{ \langle \text{this} \rangle \} \]
\[ \text{Node} = \{ \langle \text{n0}, \text{n1}, \text{n2} \rangle \} \]
\[ \text{String} = \{ \langle \text{s1}, \text{s2} \rangle \} \]
\[ \text{head} = \{ \langle \text{this}, \text{n2} \rangle \} \]
\[ \text{next} = \{ \langle \text{n2}, \text{n1} \rangle, \langle \text{n1}, \text{n0} \rangle, \langle \text{n0}, \text{null} \rangle \} \]
\[ \text{data} = \{ \langle \text{n2}, \text{s1} \rangle, \langle \text{n1}, \text{s2} \rangle, \langle \text{n0}, \text{null} \rangle \} \]

\[ \{ \} \subseteq \text{head}_0 \subseteq \{ \text{this} \} \times \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \} \]
\[ \{ \} \subseteq \text{next}_3 \subseteq \{ \text{n0}, \text{n1}, \text{n2} \} \times \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \} \]
data repair using partial models

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

head₀ ⊆ List → (Node ∪ null) ∧
next₃ ⊆ Node → (Node ∪ null) ∧

Inv(next) ∧ Pre(this, head, next) ∧
Inv(next₃) ∧ Post(this, head, head₀, next, next₃)

encoding of the repair

pre-state

\[
\begin{align*}
\text{this} & \quad \text{head} & \quad \text{n2} & \quad \text{next} & \quad \text{n1} & \quad \text{next} & \quad \text{n0} & \quad \text{next} & \quad \text{null} \\
\text{data: s1} & & & & \text{data: s2} & & \text{data: null} & &
\end{align*}
\]

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

null = \{ <null> \}

this = \{ <this> \}
List = \{ <this> \}
Node = \{ <n₀>, <n₁>, <n₂> \}
String = \{ <s₁>, <s₂> \}

head = \{ <this, n₂> \}
next = \{ <n₂, n₁>, <n₁, n₀>, <n₀, null> \}
data = \{ <n₂, s₁>, <n₁, s₂>, <n₀, null> \}

\{ \} ⊆ head₀ ⊆ \{ this \} × \{ n₀, n₁, n₂, null \}
\{ \} ⊆ next₃ ⊆ \{ n₀, n₁, n₂ \} × \{ n₀, n₁, n₂, null \}
data repair using partial models

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

head₀ ⊆ List → (Node ∪ null) ∧
next₃ ⊆ Node → (Node ∪ null) ∧

Inv(next) ∧ Pre(this, head, next) ∧
Inv(next₃) ∧ Post(this, head, head₀, next, next₃)

post-state as variables

null = { <null> }
this = { <this> }
List = { <this> }
Node = { <n₀>, <n₁>, <n₂> }
String = { <s₁>, <s₂> }

head = { <this, n₂> }
next = { <n₂, n₁>, <n₁, n₀>, <n₀, null> }
data = { <n₂, s₁>, <n₁, s₂>, <n₀, null> }

{} ⊆ head₀ ⊆ { this } × { n₀, n₁, n₂, null }
{} ⊆ next₃ ⊆ { n₀, n₁, n₂ } × { n₀, n₁, n₂, null }

encoding of the repair

pre-state

{ this, n₀, n₁, n₂, s₀, s₁, s₂, null }
data repair using partial models

\[
\begin{align*}
\text{this} & \subseteq \text{List} \land \text{one this} \land \\
\text{head} & \subseteq \text{List} \rightarrow (\text{Node} \cup \text{null}) \land \\
\text{next} & \subseteq \text{Node} \rightarrow (\text{Node} \cup \text{null}) \land \\
\text{data} & \subseteq \text{Node} \rightarrow (\text{String} \cup \text{null}) \land \\
\text{head}_0 & \subseteq \text{List} \rightarrow (\text{Node} \cup \text{null}) \land \\
\text{next}_3 & \subseteq \text{Node} \rightarrow (\text{Node} \cup \text{null}) \land \\
\text{Inv}(\text{next}) & \land \text{Pre(}\text{this}, \text{head}, \text{next}) \land \\
\text{Inv}(\text{next}_3) & \land \text{Post(}\text{this}, \text{head}, \text{head}_0, \text{next}, \text{next}_3)
\end{align*}
\]

\[
\begin{align*}
\{ \text{this}, \text{n0}, \text{n1}, \text{n2}, \text{s0}, \text{s1}, \text{s2}, \text{null} \} \\
\text{null} = \{<\text{null}>\}
\end{align*}
\]

\[
\begin{align*}
\text{this} = \{<\text{this}>\}
\end{align*}
\]

\[
\begin{align*}
\text{List} = \{<\text{this}>\}
\end{align*}
\]

\[
\begin{align*}
\text{Node} = \{<\text{n0}>, <\text{n1}>, <\text{n2}>\}
\end{align*}
\]

\[
\begin{align*}
\text{String} = \{<\text{s1}>, <\text{s2}>\}
\end{align*}
\]

\[
\begin{align*}
\text{head} = \{<\text{this}, \text{n2}>\}
\end{align*}
\]

\[
\begin{align*}
\text{next} = \{<\text{n2}, \text{n1}>, <\text{n1}, \text{n0}>, <\text{n0}, \text{null}>\}
\end{align*}
\]

\[
\begin{align*}
\text{data} = \{<\text{n2}, \text{s1}>, <\text{n1}, \text{s2}>, <\text{n0}, \text{null}>\}
\end{align*}
\]

\[
\begin{align*}
\{\} \subseteq \text{head}_0 \subseteq \{ \text{this} \} \times \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \}
\end{align*}
\]

\[
\begin{align*}
\{\} \subseteq \text{next}_3 \subseteq \{ \text{n0}, \text{n1}, \text{n2} \} \times \{ \text{n0}, \text{n1}, \text{n2}, \text{null} \}
\end{align*}
\]
data repair demo
but the bug is still lurking in the code ...
fault localization with minimal unsat cores

@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);
}
```
fault localization with minimal unsat cores

@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);
}
```

given a buggy program, a valid input and the expected output, find a minimal subset of program statements that prevents the execution on the given input from reaching the desired output state.
fault localization with minimal unsat cores

@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);
}
```

given a buggy program, a valid input and the expected output, find a minimal subset of program statements that prevents the execution on the given input from reaching the desired output state.

introduce additional “indicator” relations into the encoding.
fault localization with minimal unsat cores

@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);
}
```
given a buggy program, a valid input and the expected output, find a minimal subset of program statements that prevents the execution on the given input from reaching the desired output state.

introduce additional “indicator” relations into the encoding.

the resulting formula, together with the input/output partial model, will be unsatisfiable.
fault localization with minimal unsat cores

given a buggy program, a valid input and the expected output, find a minimal subset of program statements that prevents the execution on the given input from reaching the desired output state

introduce additional “indicator” relations into the encoding

the resulting formula, together with the input/output partial model, will be unsatisfiable

a minimal unsatisfiable core of this formula represents an irreducible cause of the program’s failure to meet the specification

› there may be (and usually are) more than one such core

› a fully fleshed out approach would take advantage of additional cores and cores from multiple failing input/output pairs

@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);
}
```
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);
}

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

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

@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);
}
```

post-condition must hold

- 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)
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);
    }

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: indicator relations

@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);
    next1 = update(next0, mid0, near0);
    near1 = mid0;
    mid1 = far0;
    far1 = far0.next1;

    boolean guard = (far0 != null);
    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) ∧

    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 encoding: partial model

\[
\begin{align*}
this & \subseteq \text{List} \land \text{one this} \land \\
\text{head} & \subseteq \text{List} \rightarrow (\text{Node} \cup \text{null}) \land \\
\text{next} & \subseteq \text{Node} \rightarrow (\text{Node} \cup \text{null}) \land \\
\text{data} & \subseteq \text{Node} \rightarrow (\text{String} \cup \text{null}) \land \\
near_0 & = \text{this}.\text{head} \land \\
\text{mid}_0 & = \text{near}_0.\text{next} \land \\
\text{far}_0 & = \text{mid}_0.\text{next} \land \\
\text{next}_0 & = \text{next} ++ (\text{near}_0 \times \text{far}_0) \land \\
\text{next}_1 & = \text{next}_0 ++ (\text{mid}_0 \times \text{near}_0) \land \\
\text{near}_1 & = \text{mid}_0 \land \\
\text{mid}_1 & = \text{far}_0 \land \\
\text{far}_1 & = \text{far}_0.\text{next}_1 \land
\end{align*}
\]

\textbf{let} guard = (\text{far}_0 \neq \text{null}),
\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) \land \\
\text{head}_0 = \text{head} ++ (\text{this} \times \text{mid}_2) \land \\
\text{far}_2 = \text{null} \land \text{Inv}(\text{next}) \land \text{Pre}(\text{this, head, next}) \land \\
\text{Inv}(\text{next}_3) \land \text{Post}(\text{this, head, head}_0, \text{next, next}_3)
fault localization encoding: partial model

\[
\begin{align*}
\text{this} & \subseteq \text{List} \land \text{one this} \land \\
\text{head} & \subseteq \text{List} \to (\text{Node} \cup \text{null}) \land \\
\text{next} & \subseteq \text{Node} \to (\text{Node} \cup \text{null}) \land \\
\text{data} & \subseteq \text{Node} \to (\text{String} \cup \text{null}) \land \\
\text{near}_0 & = \text{this.head} \land \\
\text{mid}_0 & = \text{near}_0.\text{next} \land \\
\text{far}_0 & = \text{mid}_0.\text{next} \land \\
\text{next}_0 & = \text{next}++(\text{near}_0 \times \text{far}_0) \land \\
\text{next}_1 & = \text{next}_0++(\text{mid}_0 \times \text{near}_0) \land \\
\text{near}_1 & = \text{mid}_0 \land \\
\text{mid}_1 & = \text{far}_0 \land \\
\text{far}_1 & = \text{far}_0.\text{next}_1 \land \\
\text{let} & \text{guard} = (\text{far}_0 \neq \text{null}) \\
\text{near}_2 & = \text{if} \text{ guard then near}_1 \text{ else near}_0, \\
\text{mid}_2 & = \text{if} \text{ guard then mid}_1 \text{ else mid}_0, \\
\text{far}_2 & = \text{if} \text{ guard then far}_1 \text{ else far}_0, \\
\text{next}_2 & = \text{if} \text{ guard then next}_1 \text{ else next}_0 | \\
\text{next}_3 & = \text{next}_2++(\text{mid}_2 \times \text{near}_2) \land \\
\text{head}_0 & = \text{head}++(\text{this} \times \text{mid}_2) \land \\
\text{far}_2 & = \text{null} \land \text{Inv}(\text{next}) \land \text{Pre}(\text{this, head, next}) \land \\
\text{Inv}(\text{next}_3) \land \text{Post}(\text{this, head, head}_0, \text{next, next}_3)
\end{align*}
\]
fault localization encoding: partial model

\[
\begin{align*}
\text{this} & \subseteq \text{List} \land \text{one this} \land \\
\text{head} & \subseteq \text{List} \rightarrow (\text{Node} \cup \text{null}) \land \\
\text{next} & \subseteq \text{Node} \rightarrow (\text{Node} \cup \text{null}) \land \\
\text{data} & \subseteq \text{Node} \rightarrow (\text{String} \cup \text{null}) \\
\text{near}_0 & = \text{this.head} \\
\text{mid}_0 & = \text{near}_0.\text{next} \\
\text{far}_0 & = \text{mid}_0.\text{next} \\
\text{next}_0 & = \text{next} \text{++} (\text{near}_0 \times \text{far}_0) \\
\text{next}_1 & = \text{next}_0 \text{++} (\text{mid}_0 \times \text{near}_0) \\
\text{near}_1 & = \text{mid}_0 \\
\text{mid}_1 & = \text{far}_0 \\
\text{far}_1 & = \text{far}_0.\text{next}_1 \\
\text{let guard} & = (\text{far}_0 \neq \text{null}), \\
\text{near}_2 & = \text{if guard then near}_1 \text{ else near}_0, \\
\text{mid}_2 & = \text{if guard then mid}_1 \text{ else mid}_0, \\
\text{far}_2 & = \text{if guard then far}_1 \text{ else far}_0, \\
\text{next}_2 & = \text{if guard then next}_1 \text{ else next}_0 \\
\text{next}_3 & = \text{next}_2 \text{++} (\text{mid}_2 \times \text{near}_2) \\
\text{head}_0 & = \text{head} \text{++} (\text{this} \times \text{mid}_2) \\
\text{far}_2 & = \text{null} \land \text{Inv(next)} \land \text{Pre(this, head, next)} \land \\
\text{Inv(next)} \land \text{Post(this, head, head}_0, \text{next, next}_3) & \\
\{ \text{this, n0, n1, n2, s0, s1, s2, null} \}
\end{align*}
\]
fault localization encoding: partial model

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

\text{near}_0 = \text{this.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} ++ (\text{near}_0 \times \text{far}_0) \land
\text{next}_1 = \text{next}_0 ++ (\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

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

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

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

null = \{<\text{null}>\}
\text{this} = \{<\text{this}>\}
\text{List} = \{<\text{this}>\}
\text{Node} = \{<\text{n0>, <n1>, <n2>>\}
\text{String} = \{<\text{s1>, <s2}>\}

\text{head} = \{<\text{this, n2}>\}
\text{next} = \{<\text{n2, n1>, <n1, n0>, <n0, null}>\}
\text{data} = \{<\text{n2, s1>, <n1, s2>, <n0, null}>\}
**fault localization encoding: partial model**

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

\[
\begin{align*}
\{ \text{this, n0, n1, n2, s0, s1, s2, null } \} \\
\text{null} & = \{ <\text{null}> \} \\
\text{this} & = \{ <\text{this}> \} \\
\text{List} & = \{ <\text{this}> \} \\
\text{Node} & = \{ <\text{n0}, <\text{n1}, <\text{n2} > \} \\
\text{String} & = \{ <\text{s1}, <\text{s2} > \} \\
\text{head} & = \{ <\text{this, n2} > \} \\
\text{next} & = \{ <\text{n2, n1}>, <\text{n1, n0}>, <\text{n0, null} > \} \\
\text{data} & = \{ <\text{n2, s1}>, <\text{n1, s2}>, <\text{n0, null} > \} \\
\text{head}_0 & = \{ <\text{this, n0} > \} \\
\text{next}_3 & = \{ <\text{n0, n1}, <\text{n1, n2}, <\text{n2, null} > \} \\
\{ } & \subseteq \text{next}_0 \subseteq \{ \text{n0, n1, n2} \} \times \{ \text{n0, n1, n2, null} \} \\
\{ } & \subseteq \text{next}_1 \subseteq \{ \text{n0, n1, n2} \} \times \{ \text{n0, n1, n2, null} \} \\
\{ } & \subseteq \text{near}_0 \subseteq \{ \text{n0, n1, n2, null} \} \\
\{ } & \subseteq \text{near}_1 \subseteq \{ \text{n0, n1, n2, null} \} \\
\{ } & \subseteq \text{mid}_0 \subseteq \{ \text{n0, n1, n2, null} \} \\
\{ } & \subseteq \text{mid}_1 \subseteq \{ \text{n0, n1, n2, null} \} \\
\{ } & \subseteq \text{far}_0 \subseteq \{ \text{n0, n1, n2, null} \} \\
\{ } & \subseteq \text{far}_1 \subseteq \{ \text{n0, n1, n2, null} \}
\end{align*}
\]
fault localization demo
minimal unsatisfiable core

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(next₁) ∧ Post(this, head, head₀, next, next₃)

constraints that are UNSAT (with respect to bounds) but become SAT if any member is removed

{ this, n₀, n₁, n₂, s₀, s₁, s₂, null }
null = { <null> }
this = { <this> }
List = { <this> }
Node = { <n₀>, <n₁>, <n₂> }
String = { <s₁>, <s₂> }

head = { <this, n₂> }
next = { <n₂, n₁>, <n₁, n₀>, <n₀, null> }
data = { <n₂, s₁>, <n₁, s₂>, <n₀, null> }

head₀ = { <this, n₀> }
next₃ = { <n₀, n₁>, <n₁, n₂>, <n₂, null> }

{} ⊆ next₀ ⊆ { n₀, n₁, n₂ } × { n₀, n₁, n₂, null }
{} ⊆ next₁ ⊆ { n₀, n₁, n₂ } × { n₀, n₁, n₂, null }
{} ⊆ near₀ ⊆ { n₀, n₁, n₂, null }
{} ⊆ near₁ ⊆ { n₀, n₁, n₂, null }
{} ⊆ mid₀ ⊆ { n₀, n₁, n₂, null }
{} ⊆ mid₁ ⊆ { n₀, n₁, n₂, null }
{} ⊆ far₀ ⊆ { n₀, n₁, n₂, null }
{} ⊆ far₁ ⊆ { n₀, n₁, n₂, null }
void reverse() {
    Node near0 = this.head;
    Node mid0 = near0.next;
    Node far0 = mid0.next;

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

    boolean guard = (far0 != null);
    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);
}
synthesizing a sketch-like fix

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

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

    boolean guard = (far0 != null);
    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);
}
synthesizing a sketch-like fix

@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, ??);
    next1 = update(next0, mid0, near0);
    near1 = mid0;
    mid1 = far0;
    far1 = far0.next1;

    boolean guard = (far0 != null);
    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);
}
```

drill a hole in one of the localized statements
  • e.g., at the earliest opportunity for a fix
synthesizing a sketch-like fix

@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, ??);
    next1 = update(next0, mid0, near0);
    near1 = mid0;
    mid1 = far0;
    far1 = far0.next1;

    boolean guard = (far0 != null);
    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);
}
```

drill a hole in one of the localized statements

- e.g., at the earliest opportunity for a fix

we want to replace the hole with an expression from a (small) grammar so that the program satisfies its spec on all inputs

- e.g., [ variable | null ]
synthesizing a sketch-like fix

@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, ??);
    next1 = update(next0, mid0, near0);
    near1 = mid0;
    mid1 = far0;
    far1 = far0.next1;

    boolean guard = (far0 != null);
    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);
}
```

drill a hole in one of the localized statements

- e.g., at the earliest opportunity for a fix

we want to replace the hole with an expression from a (small) grammar so that the program satisfies its spec on all inputs

- e.g., [ variable | null ]

encode the synthesis problem (for one input) using relations that represent syntax, together with a “meaning” expression connecting syntax to semantics
synthesizing a sketch-like fix

@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, ??);
    next₁ = update(next₀, mid₀, near₀);
    near₁ = mid₀;
    mid₁ = far₀;
    far₁ = far₀.next₁;

    boolean guard = (far₀ != null);
    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);
}
synthesis encoding

```plaintext
@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, ??);
    next1 = update(next0, mid0, near0);
    near1 = mid0;
    mid1 = far0;
    far1 = far0.next1;

    boolean guard = (far0 != null);
    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);
}

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)
```

start with the first step of the repair encoding
synthesis encoding

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

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

    next0 = update(next, near0, ??);
    next1 = update(next0, mid0, near0);
    near1 = mid0;
    mid1 = far0;
    far1 = far0.next1;

    boolean guard = (far0 != null);
    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 ∧ one hole ∧
head ⊆ List → (Node ∪ null) ∧
next ⊆ Node → (Node ∪ null) ∧
data ⊆ Node → (String ∪ null) ∧

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

    meaning = (null × null) ∪ (“head” × this.head) ∪
    (“near0” × near0) ∪ (“mid0” × mid0) ∪ (“far0” × far0)

    next0 = next ++ (near0 × hole.meaning),
    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)
**synthesis encoding: partial model**

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

\text{let near}_0 = \text{this.head}, \\
\quad mid_0 = \text{near}_0.\text{next}, \\
\quad far_0 = \text{mid}_0.\text{next}, \\
\quad \\
\quad \text{meaning} = (\text{null} \times \text{null}) \cup ("\text{head}" \times \text{this.head}) \cup \\
\quad ("\text{near}_0" \times \text{near}_0) \cup ("\text{mid}_0" \times \text{mid}_0) \cup ("\text{far}_0" \times \text{far}_0) \\
\quad \\
\quad \text{next}_0 = \text{next} ++ (\text{near}_0 \times \text{hole.}\text{meaning}), \\
\quad \text{guard} = (\text{far}_0 \neq \text{null}), \\
\quad \text{next}_1 = \text{next}_0 ++ (\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 \\
\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 ++ (\text{mid}_2 \times \text{near}_2) \\
\quad \text{head}_0 = \text{head} ++ (\text{this} \times \text{mid}_2) \mid \\
\quad \text{far}_2 = \text{null} \land \text{Inv(next)} \land \text{Pre(this, head, next)} \land \\
\quad \text{Inv(next}_3) \land \text{Post(this, head, head}_0, \text{next, next}_3) \\
\]

\{ \text{this, n0}, \text{n1}, \text{n2}, \text{s0}, \text{s1}, \text{s2}, \text{null}, \\
\quad "\text{head}"", "\text{near}_0"", "\text{mid}_0"", "\text{far}_0" \}

\quad \text{null} = \{ \langle\text{null}\rangle\} \\
\quad \text{this} = \{ \langle\text{this}\rangle\} \\
\quad \text{List} = \{ \langle\text{this}\rangle\} \\
\quad \text{Node} = \{ <\text{n0}>, <\text{n1}>, <\text{n2}> \} \\
\quad \text{String} = \{ <\text{s1}>, <\text{s2}> \} \\
\quad \text{head} = \{ <\text{this}, \text{n2}>, \} \\
\quad \text{next} = \{ <\text{n2}, \text{n1}>, <\text{n1}, \text{n0}>, <\text{n0}, \text{null}> \} \\
\quad \text{data} = \{ <\text{n2}, \text{s1}>, <\text{n1}, \text{s2}>, <\text{n0}, \text{null}> \} \\
\quad "\text{head}" = \{ \langle"\text{head}"\rangle\} \\
\quad "\text{near}_0" = \{ \langle"\text{near}_0"\rangle\} \\
\quad "\text{mid}_0" = \{ \langle"\text{mid}_0"\rangle\} \\
\quad "\text{far}_0" = \{ \langle"\text{far}_0"\rangle\} \\
\quad \}

\{ \} \subseteq \text{hole} \subseteq \{ \langle\text{null}\rangle, <"\text{head}>", <"\text{near}_0">, <"\text{mid}_0">, <"\text{far}_0"> \}
synthesis demo
patched list reversal

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

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