Programming with constraint solvers

toward a shared infrastructure for code checking, angelic execution, debugging, and synthesis

emina torlak

BOOGIE 2012 · July 08, 2012
programming ...

specification

P(x) {
  ...
}

assume pre(x)
assert post(P(x))
programming ...

formula, input/output pairs, traces, another program, ...

P(x) {
  
}

programming ...

assume pre(x)
assert post(P(x))
programming with a solver

```plaintext
assume pre(x)
P(x) {
    ...
}
assert post(P(x))
```

translate(…)

SAT/SMT solver
programming with a solver: code checking

Is there a valid input $x$ for which $P(x)$ violates the spec?

```plaintext
assume pre(x)
P(x) {
  ...
}
assert post(P(x))
```

$\exists x . \text{pre}(x) \land \neg \text{post}(P(x))$

SAT/SMT solver

CBMC [Oxford], Dafny [MSR], Jahob [EPFL], Miniatur / MemSAT [IBM], etc.
Is there a valid input $x$ for which $P(x)$ violates the spec?

**Assume** $\text{pre}(x)$

$\text{P}(x) \{ ... \}$

**Assert** $\text{post}(\text{P}(x))$

$$\exists x . \text{pre}(x) \land \neg \text{post}(\text{P}(x))$$

**Counterexample** $x = 42$

**SAT/SMT solver**

CBMC [Oxford], Dafny [MSR], Jahob [EPFL], Miniatur / MemSAT [IBM], etc.
programming with a solver: debugging

Given x and x′, what subset of P is responsible for P(x) ≠ x′?

```
assume pre(x)
P(x) {
  v = x + 2
  ...
}
assert post(P(x))
```

```
 pre(x) ∧ post(x′) ∧ x′ = P(x)
```

SAT/SMT solver
Given $x$ and $x'$, what subset of $P$ is responsible for $P(x) \neq x'$?

```latex
\text{assume pre}(x)\\
\text{P}(x) \{ v = x + 2 \} \\
\text{assert post}(P(x))
```

\[ \text{pre}(x) \wedge \text{post}(x') \wedge x' = P(x) \]

SAT/SMT solver

MAXSAT/ MIN CORE

repair candidates
programming with a solver: angelic execution

Given x, choose v at runtime so that P(x, v) satisfies the spec.

\[ \exists v . \text{pre}(x) \land \text{post}(P(x, v)) \]

assume \( \text{pre}(x) \)
\[
P(x) \{
  v = \text{choose}()
  ...
\}
assert \( \text{post}(P(x)) \)

SAT/SMT solver

Kaplan [EPFL], PBnJ [UCLA], Skalch [Berkeley], Squander [MIT], etc.
programming with a solver: angelic execution

Given $x$, choose $v$ at runtime so that $P(x, v)$ satisfies the spec.

\[
\exists v . \ pre(x) \land post(P(x, v))
\]

\[
\begin{align*}
\text{assume} & \quad pre(x) \\
P(x) \{ \\
\quad v = \text{choose()} \\
\quad \ldots \} \\
\text{assert} & \quad post(P(x))
\end{align*}
\]

\[
v = 0, \ldots
\]

trace

model

SAT/SMT solver

Kaplan [EPFL], PBnJ [UCLA], Skalch [Berkeley], Squander [MIT], etc.
programming with a solver: synthesis

Replace ?? with expression e so that $P_e(x)$ satisfies the spec on all valid inputs.

```
assume pre(x)
P(x) {
v = ??
... }
assert post(P(x))
```

$\exists e . \forall x . pre(x) \Rightarrow post(P_e(x))$

SAT/SMT solver

Comfusy [EPFL], Sketch [Berkeley / MIT]
Replace ?? with expression e so that $P_e(x)$ satisfies the spec on all valid inputs.

\[ \forall x . \text{pre}(x) \rightarrow \text{post}(P_e(x)) \]

SAT/SMT solver

Comfusy [EPFL], Sketch [Berkeley / MIT]
Replace ?? with expression e so that $P_e(x)$ satisfies the spec on all valid inputs.

```plaintext
assume pre(x)
P(x) {
  v = x - 2
  ...
}
assert post(P(x))
```

$\exists e . \forall x . pre(x) \land post(P_e(x))$

SAT/SMT solver
programming with kodkod

```
assume pre(x)
P(x) {
  ...
}
assert post(P(x))
```

translate(…)

?
KODKOD
a constraint solver for relational logic

about kdkod
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 (ema at alum.mit.edu)
some applications of kodkod

checking theorems & designs

› Alloy4 (Alloy), Nitpick (Isabelle/HOL), ProB (B, Event-B, Z and TLA⁺), 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**

```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;
    }
}
```

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

```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;
}
```
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;
}
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() {
        @requires Pre(this, head, next)
        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 {
    @invariant Inv(next)
    Node next;
    String data;
}
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;
}

![Diagram of node traversal](image.png)
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 \(\equiv\) \{ <this, n2> \}, next \(\equiv\) \{ <n2, n1>, ... \}
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 $\equiv \{ <this, n2> \}$, next $\equiv \{ <n2, n1>, \ldots \}$

types as sets (unary relations)
- List $\equiv \{ <this> \}$, Node $\equiv \{ <n0>, <n1>, <n2> \}$
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;
}

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)

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> }

\[
\begin{array}{cccccc}
\text{this} & \text{head} & n2 & \text{data: s1} & \text{next} & n1 & \text{data: s2} & \text{next} & n0 & \text{data: null} & \text{next} & \text{null}
\end{array}
\]
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> \}
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;
}
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;
}


code checking with kodkod

@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;
    if (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

next\_0 = update(next, near\_0, far\_0);
boolean guard = (far\_0 != null);
next\_1 = update(next\_0, mid\_0, near\_0);
near\_1 = mid\_0;
mid\_1 = far\_0;
far\_1 = far\_0.next\_1;

near\_2 = phi(guard, near\_1, near\_0);
mid\_2 = phi(guard, mid\_1, mid\_0);
far\_2 = phi(guard, far\_1, far\_0);
next\_2 = phi(guard, next\_1, next\_0);

assume far\_2 == null;

next\_3 = update(next\_2, mid\_2, near\_2);
head\_0 = update(head, this, mid\_2);
}
code checking with kodkod

@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;
    near₀.next = far₀;
    if (far₀ != null) {
        mid₀.next = near₀;
        near = mid₀;
        mid = far₀;
        far = far₀.next;
    }
    assume far₂ == null;
    next₃ = update(next₂, mid₂, near₂);
    head₀ = update(head, this, mid₂);
}
code checking with kodkod

@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;
    near₀.next = far₀;
    while (far₀ != null) {
        mid₀.next = near₀;
        near₀ = mid₀;
        mid₀ = far₀;
        far₀ = far₀.next;
    }
    mid₀.next = near₀;
    head₀ = mid₀;
}

finitize loops
▷ e.g., unwind once
convert to SSA
▷ SSA for both locals and fields
encode program semantics in relational logic
specify analysis bounds
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;
}
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);
}
```

introduce a unary relation for each reference and type, and a binary relation for each field

- constrain reference relations to be singletons
- constrain field relations to be functions

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

@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) ∧

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

@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

@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

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

let near_0 = this.head,
    mid_0 = near_0.next,
    far_0 = mid_0.next,

    next_0 = next ++ (near_0 \times far_0),
guard = (far_0 != \text{null}),
next_1 = next_0 ++ (mid_0 \times near_0),
near_1 = mid_0,
    mid_1 = far_0,
    far_1 = far_0.next_1,

    near_2 = \text{if guard then near_1 else near_0},
    mid_2 = \text{if guard then mid_1 else mid_0},
    far_2 = \text{if guard then far_1 else far_0},
next_2 = \text{if guard then next_1 else next_0},
next_3 = next_2 ++ (mid_2 \times near_2)
head_0 = head ++ (this \times mid_2) |

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, next, next_3)})

finite universe of uninterpreted elements
\begin{itemize}
    \item e.g., 1 List object, 3 of everything else
\end{itemize}

upper bound on each relation
\begin{itemize}
    \item set of tuples, drawn from the universe, that the relation \text{may} contain
\end{itemize}

lower bound on each relation
\begin{itemize}
    \item set of tuples, drawn from the universe, that the relation \text{must} contain
    \item lower bounds collectively form a \text{partial model}
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,
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))

universe

{ this, n₀, n₁, n₂, s₀, s₁, s₂, null }
{ <null> } ⊆ null ⊆ { <null> }

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

{} ⊆ head ⊆ { this } × { n₀, n₁, n₂, null }
{} ⊆ next ⊆ { n₀, n₁, n₂ } × { n₀, n₁, n₂, null }
{} ⊆ data ⊆ { n₀, n₁, n₂ } × { s₀, s₁, s₂, null }
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,
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))

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

{ <null> } ⊆ null ⊆ { <null> }

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

{} ⊆ head ⊆ { this } × { n0, n1, n2, null }
{} ⊆ next ⊆ { n0, n1, n2 } × { n0, n1, n2, null }
{} ⊆ data ⊆ { n0, n1, n2 } × { s0, s1, s2, null }

upper bound
specifying analysis bounds

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

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

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

\text{let } \text{near}_0 = \text{this.\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} \ ++ \ (\text{near}_0 \times \text{far}_0),
\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 \text{near}_2 = \text{if } \text{guard} \ \text{then } \text{near}_1 \ \text{else } \text{near}_0,
\quad \text{mid}_2 = \text{if } \text{guard} \ \text{then } \text{mid}_1 \ \text{else } \text{mid}_0,
\quad \text{far}_2 = \text{if } \text{guard} \ \text{then } \text{far}_1 \ \text{else } \text{far}_0,
\quad \text{next}_2 = \text{if } \text{guard} \ \text{then } \text{next}_1 \ \text{else } \text{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)
\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))}

\{ \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 \ 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 } \}

{ <\text{null}> } \subseteq \text{null} \subseteq { <\text{null}> }
code checking demo
a bug! what to do about it?
data repair with angelic execution

@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 with angelic execution

@invariant \texttt{Inv(next)}
@requires \texttt{Pre(this, head, next)}
@ensures \texttt{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 != \texttt{null});
    next1 = update(next0, mid0, near0);
    near1 = mid0;
    mid1 = far0;
    far1 = far0.next1;

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

    assume far2 == \texttt{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 with angelic execution

@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 with angelic execution

@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

for details see

- Cobbler [Zaeem et al., 2012]
- Squander [Milicevic et al., 2011]
- PBnJ [Samimi et al., 2010]
- Tarmeem [Zaeem et al., 2010]
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)

{ 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 }

pre-state
data repair using partial models

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

\[ \text{head}_0 \subseteq \text{List} \rightarrow (\text{Node} \cup \text{null}) \land \]
\[ \text{next}_3 \subseteq \text{Node} \rightarrow (\text{Node} \cup \text{null}) \land \]

\[ \text{Inv} (\text{next}) \land \text{Pre} (\text{this}, \text{head}, \text{next}) \land \]
\[ \text{Inv} (\text{next}_3) \land \text{Post} (\text{this}, \text{head}, \text{head}_0, \text{next}, \text{next}_3) \]

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

\[
\{ \} \subseteq \text{head}_0 \subseteq \{ \text{this} \} \times \{ \text{n0, n1, n2, null} \}
\]
\[
\{ \} \subseteq \text{next}_3 \subseteq \{ \text{n0, n1, n2} \} \times \{ \text{n0, n1, n2, null} \}
\]
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)

{ 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 }

encoding of the repair

pre-state

post-state as variables


data repair using partial models

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

\[ \text{head}_0 \subseteq \text{List} \rightarrow (\text{Node} \cup \text{null}) \land \]
\[ \text{next}_3 \subseteq \text{Node} \rightarrow (\text{Node} \cup \text{null}) \land \]

\[ \text{Inv}(\text{next}) \land \text{Pre}(\text{this}, \text{head}, \text{next}) \land \]
\[ \text{Inv}(\text{next}_3) \land \text{Post}(\text{this}, \text{head}, \text{head}_0, \text{next}, \text{next}_3) \]

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

\[ \text{null} = \{ <\text{null}> \} \]

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

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

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

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

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
fault localization encoding

@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);
}

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);
}
```

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);
}
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

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

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

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

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

\[ \text{let} \quad \text{guard} = (\text{far}_0 \neq \text{null}), \]

\[ \quad \text{near}_2 = \text{if} \; \text{guard} \; \text{then} \; \text{near}_1 \; \text{else} \; \text{near}_0, \quad \text{mid}_2 = \text{if} \; \text{guard} \; \text{then} \; \text{mid}_1 \; \text{else} \; \text{mid}_0, \quad \text{far}_2 = \text{if} \; \text{guard} \; \text{then} \; \text{far}_1 \; \text{else} \; \text{far}_0, \quad \text{next}_2 = \text{if} \; \text{guard} \; \text{then} \; \text{next}_1 \; \text{else} \; \text{next}_0 | \]

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

\[ \{ \text{this, n0, n1, n2, s0, s1, s2, null} \} \]
fault localization encoding: partial model

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

\[
\begin{align*}
\{ \text{this, n0, n1, n2, s0, s1, s2, null} \}
\end{align*}
\]
fault localization encoding: partial model

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)

{ 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> }

null ⊆ next₀ ⊆ { n₀, n₁, n₂ } × { n₀, n₁, n₂, null }
null ⊆ next₁ ⊆ { n₀, n₁, n₂ } × { n₀, n₁, n₂, null }
null ⊆ near₀ ⊆ { n₀, n₁, n₂, null }
null ⊆ near₁ ⊆ { n₀, n₁, n₂, null }
null ⊆ mid₀ ⊆ { n₀, n₁, n₂, null }
null ⊆ mid₁ ⊆ { n₀, n₁, n₂, null }
null ⊆ far₀ ⊆ { n₀, n₁, n₂, null }
null ⊆ far₁ ⊆ { n₀, n₁, n₂, null }

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

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

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

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

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

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

\{ \text{this, } n_0, n_1, n_2, s_0, s_1, s_2, \text{null} \}

\{ \text{null} \}
\{ \text{this} \}
\{ \text{List} \}
\{ \text{Node} \}
\{ \text{String} \}

\{ \text{head} \}
\{ \text{next} \}
\{ \text{data} \}

\{ \text{head}_0 \}
\{ \text{next}_3 \}
\{ \text{far}_2 \}

{} \subseteq \text{next}_0 \subseteq \{ n_0, n_1, n_2 \} \times \{ n_0, n_1, n_2, \text{null} \}
{} \subseteq \text{next}_1 \subseteq \{ n_0, n_1, n_2 \} \times \{ n_0, n_1, n_2, \text{null} \}
{} \subseteq \text{near}_0 \subseteq \{ n_0, n_1, n_2, \text{null} \}
{} \subseteq \text{near}_1 \subseteq \{ n_0, n_1, n_2, \text{null} \}
{} \subseteq \text{mid}_0 \subseteq \{ n_0, n_1, n_2, \text{null} \}
{} \subseteq \text{mid}_1 \subseteq \{ n_0, n_1, n_2, \text{null} \}
{} \subseteq \text{far}_0 \subseteq \{ n_0, n_1, n_2, \text{null} \}
{} \subseteq \text{far}_1 \subseteq \{ n_0, n_1, n_2, \text{null} \}
fault localization demo
minimal unsatisfiable core

this ⊆ List ∧ one this ∧
head ⊆ List → (Node ∪ null) ∧
ext ⊆ 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(next) ∧ 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 }
minimal unsatisfiable core

@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);
    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);
}
```
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)

void reverse() {
    Node near_0 = this.head;
    Node mid_0 = near_0.next;
    Node far_0 = mid_0.next;

    next_0 = update(next, near_0, ??);
    next_1 = update(next_0, mid_0, near_0);
    near_1 = mid_0;
    mid_1 = far_0;
    far_1 = far_0.next_1;

    boolean guard = (far_0 != null);
    near_2 = phi(guard, near_1, near_0);
    mid_2 = phi(guard, mid_1, mid_0);
    far_2 = phi(guard, far_1, far_0);
    next_2 = phi(guard, next_1, next_0);

    assume far_2 == null;

    next_3 = update(next_2, mid_2, near_2);
    head_0 = update(head, this, mid_2);
}
synthesizing a sketch-like fix

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 ]

```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, ??);
    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)

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);
}
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.next;

    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

- wrap into a CEGIS loop (not done here)
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)
synthesis encoding

@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);
}

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

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)

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> }

“head” = { “<head>” }
“near₀” = { “<near₀>” }
“mid₀” = { “<mid₀>” }
“far₀” = { “<far₀>” }

{} ⊆ hole ⊆ { <null>, “<head>”, “<near₀>”,
“<mid₀>”, “<far₀>” }

this head n2 next n1 next n0 next null
synthesis demo
patched list reversal

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

    mid.next = near;
    head = mid;
}
state of the art: fundamental challenges

Decidability, tractability and scalability

- work in a small finite universe
- rich properties, incomplete reasoning
- work with (mostly) decidable logics
- complete reasoning, limited properties

assume $\text{pre}(x)$
$P(x) \{$
  ...
$\}$
assert $\text{post}(P(x))$

translate($\ldots$)

SAT/SMT solver
state of the art: fundamental challenges

decidability, tractability and scalability
- work in a **small finite universe**
- rich properties, incomplete reasoning
- work with (mostly) **decidable logics**
- complete reasoning, limited properties

```
assume pre(x)
P(x) {
  ...
}
assert post(P(x))
```
state of the art: fundamental challenges

```
assume pre(x)
P(x) {
  ...
}
assert post(P(x))
```

decidability, tractability and scalability

- work in a **small finite universe**
- rich properties, incomplete reasoning
- work with (mostly) **decidable logics**
- complete reasoning, limited properties

translate(…)

SAT/SMT solver
state of the art: engineering challenges

building a new tool ≈ Ph.D.

- pre-requisites
  - program analysis, formal methods, logic
  - deep knowledge of the solver
  - fragmented infrastructure
  - hard to reuse encodings, analyses

```
assume pre(x)
P(x) {
  ...
}
assert post(P(x))
```

translate(…)

SAT/SMT solver
state of the art: engineering challenges

building a new tool ≈ Ph.D.

- pre-requisites
  - program analysis, formal methods, logic
  - deep knowledge of the solver
- fragmented infrastructure
  - hard to reuse encodings, analyses

\begin{verbatim}
assume pre(x)
P(x) {
  ...
}
assert post(P(x))
\end{verbatim}

\texttt{translate(...)}

\texttt{SAT/SMT solver}
state of the art: engineering challenges

building a new tool ≈ Ph.D.

- pre-requisites
- program analysis, formal methods, logic
- deep knowledge of the solver
- fragmented infrastructure
- hard to reuse encodings, analyses

```
assume pre(x)
P(x) {
  ...
}
assert post(P(x))
```

translate(…)

SAT/SMT solver
state of the art: design & usability challenges

using an existing tool ≈ ABD

- it’s hard to keep the system in the loop
- need abstractions for communicating domain knowledge to the solver
- it’s hard to keep the user in the loop
- need abstractions for communicating failures to the user

```plaintext
assume pre(x)
P(x) {
  ...
}
assert post(P(x))
```

translate(…)

SAT/SMT solver
state of the art: design & usability challenges

using an existing tool $\approx$ ABD

- it’s hard to keep the **system in the loop**
  - need abstractions for communicating domain knowledge to the solver
- it’s hard to keep the **user in the loop**
  - need abstractions for communicating failures to the user

assume $\text{pre}(x)$
$\text{P}(x) \{$
  ...
$\}$
assert $\text{post}(\text{P}(x))$

translate(...)
state of the art: design & usability challenges

using an existing tool $\approx$ ABD

- it’s hard to keep the **system in the loop**
  - need abstractions for communicating domain knowledge to the solver
- it’s hard to keep the **user in the loop**
  - need abstractions for communicating failures to the user

```plaintext
assume pre(x)
P(x) {
  ...
}
assert post(P(x))
```

```plaintext
translate(…)
```
vision: programming with solvers in ?? years

building a tool $\approx$ embedding a DSL into Scala or Racket

using a tool $\approx$ using a DSL

\[
\begin{align*}
\text{assume } & \text{pre}(x) \\
\text{P}(x) & \{ \\
\text{...} & \\
\} \\
\text{assert } & \text{post}(\text{P}(x))
\end{align*}
\]
our new tool: rosette

assume pre(x)
P(x) {
  ...
}
assert post(P(x))

code checking
angelic execution
debugging
synthesis
for high performance computing
our new tool: rosette

if you need to prototype a solver-based tool for your programming model or DSL

› define the language by writing an interpreter
› use angelic execution to simplify this task
› define what “holes” look like in the language
› Rosette will do the rest
› automatic translation to formulas
› thus enabling checking, angelic execution, debugging and synthesis for your language

if you need to a more scalable tool

› Rosette will help you develop an efficient translator to formulas
› debug and synthesize an efficient encoding
› by ensuring it is semantically equivalent to the one obtained from the interpreter
thank you!
thank you!

questions?