Advanced encoding of programs

CS294: Program Synthesis for Everyone

Ras Bodik
Emina Torlak

Division of Computer Science
University of California, Berkeley

today

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

today

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

Lecture 3

ENCODING	SOLVER	TIME (SEC)
QF_AUFLIA	CVC3	>600
	Z3	159
QF_AUFBV	Boolector	409
	Z3	287
	CVC3	119
QF_AUFBV (non extensional)	CVC3	>600
	Boolector	>600
	Z3	25
	STP	11
REL_BV	Rosette	9
REL	Kodkod	5

today

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

Lecture 3

ENCODING	SOLVER	TIME (SEC)
QF_AUFLIA	CVC3	>600
	Z3	159
QF_AUFBV	Boolector	409
	Z3	287
	CVC3	119
QF_AUFBV (non extensional)	CVC3	>600
	Boolector	>600
	Z3	25
	STP	11
REL_BV	Rosette	9
REL	Kodkod	5

today

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

Lecture 3

ENCODING	SOLVER	TIME (SEC)
QF_AUFLIA	CVC3	>600
	Z3	159
QF_AUFBV	Boolector	409
	Z3	287
	CVC3	119
QF_AUFBV (non extensional)	CVC3	>600
	Boolector	>600
	Z3	25
	STP	11
REL_BV	Rosette	9
REL	Kodkod	5

today

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)

Lecture 3

ENCODING	SOLVER	TIME (SEC)
QF_AUFLIA	CVC3	>600
	Z3	159
QF_AUFBV	Boolector	409
	Z3	287
	CVC3	119
QF_AUFBV (non extensional)	CVC3	>600
	Boolector	>600
	Z3	25
	STP	11
REL_BV	Rosette	9
REL	Kodkod	5

alloy.mit.edu/kodkod

about | download | documentation | publications | applications | thanks

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.

news

Kodkod 1.5 is available for download!

It provides native support for the latest SAT solvers, including SAT4J 2.3, MiniSat 2.2, CryptoMiniSat 2, Lingeling and Plingeling.

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$^+$), ExUML (UML)

checking code & memory models

- Forge, Karun, Miniatur, TACO, MemSAT

declarative programming, fault recovery & data structure repair

- Squander, PBnJ, Tarmeem, Cobbler

declarative configuration

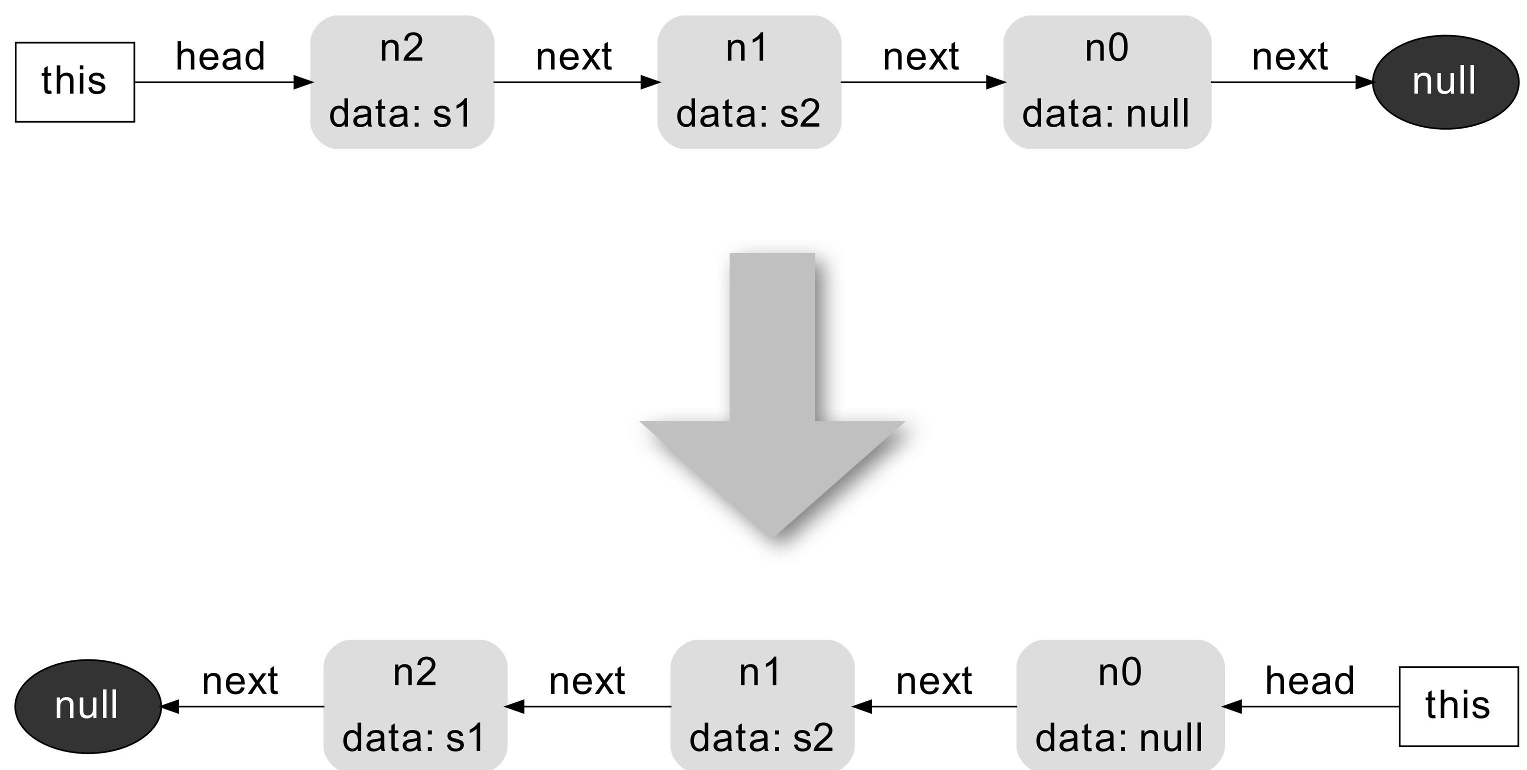
- ConfigAssure (networks), Margrave (policies)

test-case generation

- Kesit, Whispec

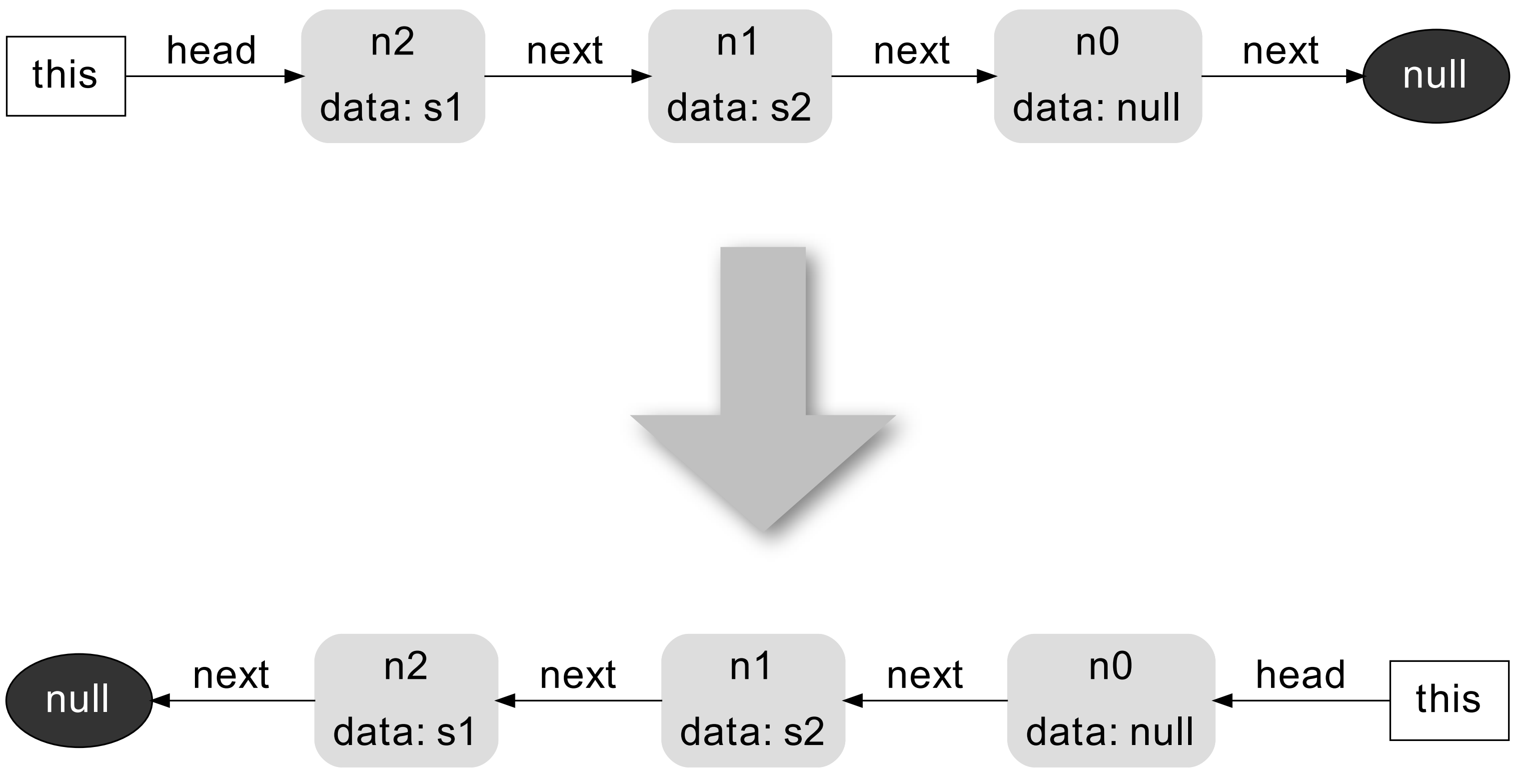
example: reversing a linked list

```
class List {
  Node head;

  void reverse() {





  }
}

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



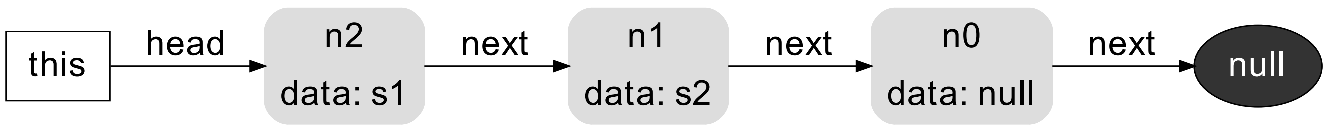
example: reversing a linked list

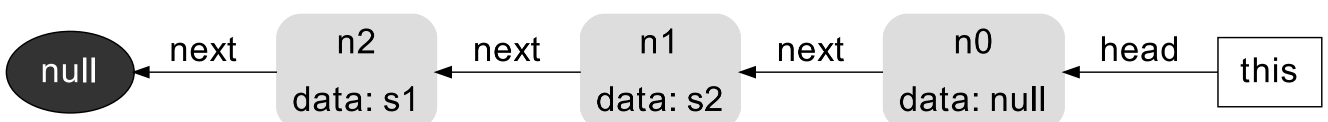
```
class List {
  Node head;

  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;
}
```



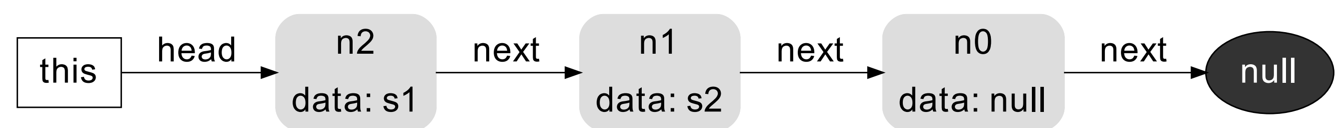


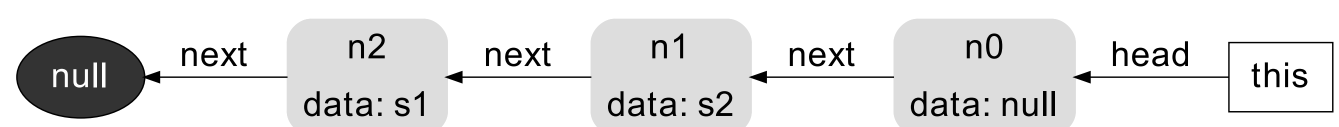
@invariant **no** ^next ∩ **iden**

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;
}
```

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



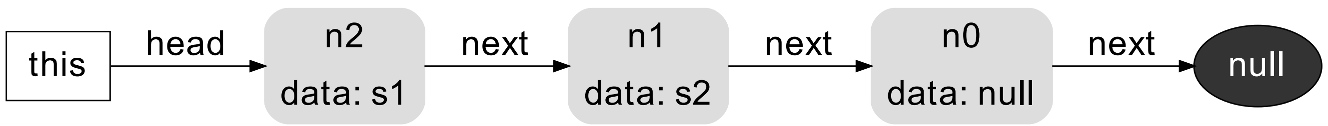


@invariant **no** ^next ∩ **iden**

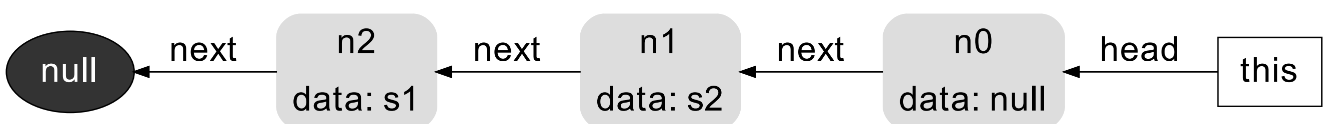
invariants, pre and post conditions

```
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;
}
```

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



@ensures
this.head.*next = this.old(head).*old(next) &&



@invariant **no** ^next ∩ **iden**

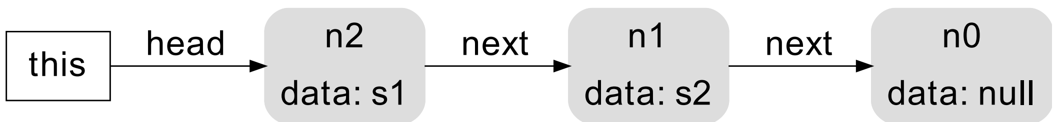
invariants, pre and post conditions

```
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;
}
```

@requires

this.head != null &&

this.head.next != null

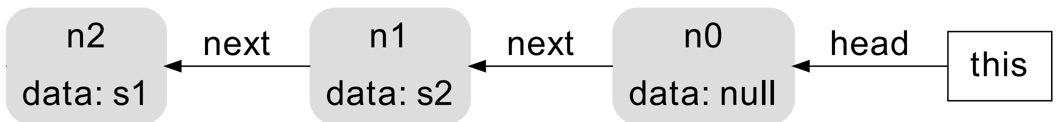


@ensures

this.head.*next = this.old(head).*old(next) &&

let N = (this.head.*next – null), bound = N × N **|**

next ∩ bound = ~old(next) ∩ bound



@invariant **no** ^next ∩ **iden**

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;
}
```

@requires Pre(this, head, next)

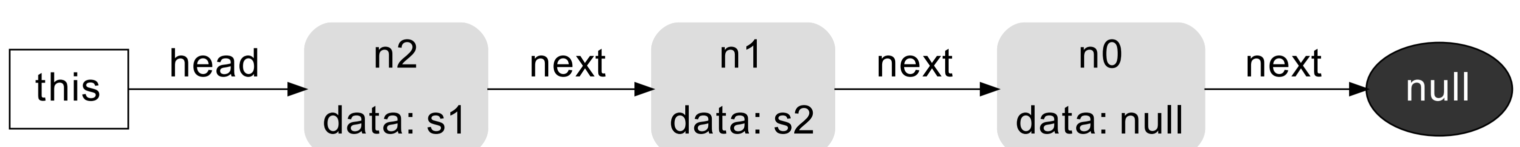
@ensures Post(this, old(head), head, old(next), next)

@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;
}
```



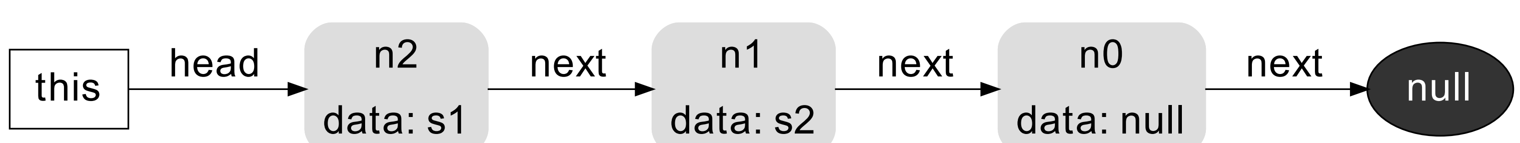
a relational view of the heap

@invariant 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>, … }



a relational view of the heap

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

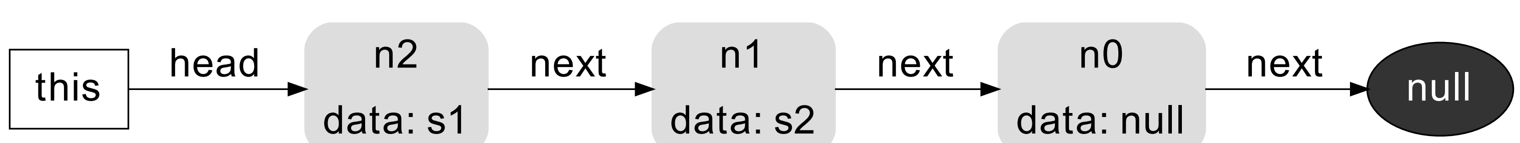
```
void reverse() {
   Node near = head;
   Node mid = near.next;
   Node far = mid.next;

   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> }



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

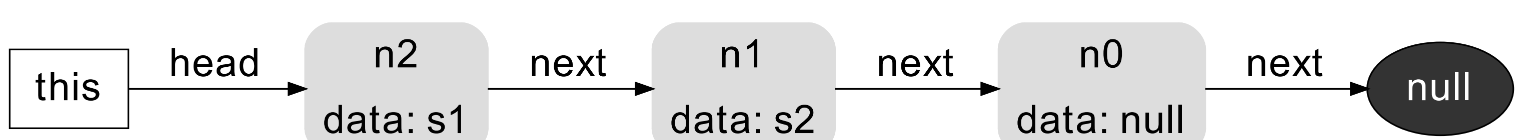
- head ≡ { <this, n2> }, next ≡ { <n2, n1>, … }

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)

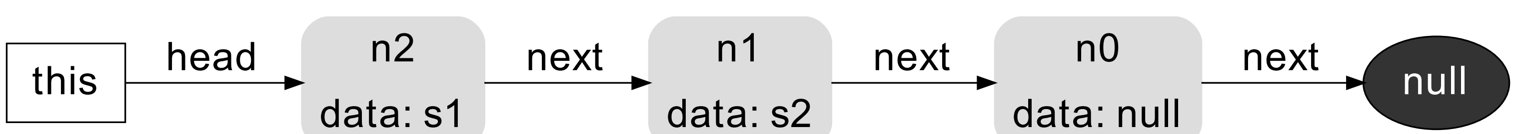
- List ≡ { <this> }, Node ≡ { <n0>, <n1>, <n2> }

objects as scalars (singleton unary relations)

- **this** ≡ { <this> }, **null** ≡ { <null> }

field access as relational join (.)

- **this**.head ≡ { <this> } . { <this, n2> } = { <n2> }



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)

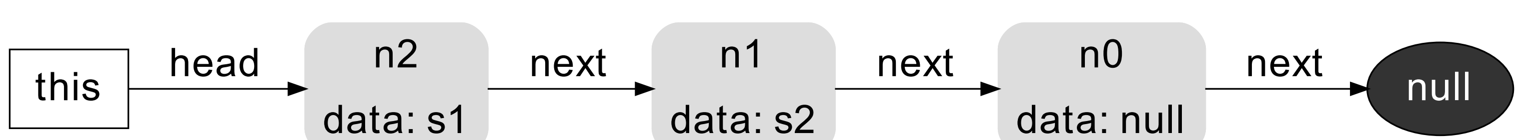
- `this` ≡ { <this> }, `null` ≡ { <null> }

field access as relational join (.)

- `this.head` ≡ { <this> } . { <this, n2> } = { <n2> }

field update as relational override (++)

- `this.head = null` ≡ head ++ (this × null) = { <this, n2> } ++ { <this, null> } = { <this, null> }



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 = 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)

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)

```
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);
}
```

finitize loops

- e.g., unwind once

convert to SSA

- SSA for both locals and fields

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;

  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);
}
```

finitize loops

- e.g., unwind once

convert to SSA

- SSA for both locals and fields

encode program semantics in relational logic

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;

   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);
}
```

finitize loops

- e.g., unwind once

convert to SSA

- SSA for both locals and fields

encode program semantics in relational logic

specify analysis bounds

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;

   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);
}
```

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 near_0 = this.head;
   Node mid_0 = near_0.next;
   Node far_0 = mid_0.next;

   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);
}
```

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);
}
```

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

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);
}
```

$\text{this} \subseteq \text{List} \land \textbf{one}\ \text{this} \land$
$\text{head} \subseteq \text{List} \times (\text{Node} \cup \text{null}) \land (\forall\ l\text{: List} \mid \textbf{one}\ l.\text{head})$
$\text{next} \subseteq \text{Node} \rightarrow (\text{Node} \cup \text{null}) \land$
$\text{data} \subseteq \text{Node} \rightarrow (\text{String} \cup \text{null}) \land$

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);
}
```

$this \subseteq List \wedge$ **one** $this \wedge$
$head \subseteq List \rightarrow (Node \cup null) \wedge$
$next \subseteq Node \rightarrow (Node \cup null) \wedge$
$data \subseteq Node \rightarrow (String \cup null) \wedge$

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

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

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

$far_2 = null \wedge Inv(next) \wedge Pre(this, head, next) \wedge$
$\neg (Inv(next_3) \wedge Post(this, head, head_0, next, next_3))$

specifying analysis bounds

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

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

$\text{next}_0 = \text{next} \mathbin{++} (\text{near}_0 \times \text{far}_0)$,
$\text{guard} = (\text{far}_0 \mathrel{!=} \text{null})$,
$\text{next}_1 = \text{next}_0 \mathbin{++} (\text{mid}_0 \times \text{near}_0)$,
$\text{near}_1 = \text{mid}_0$,
$\text{mid}_1 = \text{far}_0$,
$\text{far}_1 = \text{far}_0.\text{next}_1$,

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

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

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

this ⊆ List ∧ **one** this
head ⊆ List → (Node ∪ null)
next ⊆ Node → (Node ∪ null)
data ⊆ Node → (String ∪ 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 != null),
$next_1$ = $next_0$ ++ ($mid_0 \times near_0$),
$near_1$ = mid_0,
mid_1 = far_0,
far_1 = far_0.$next_1$,

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

far_2 = null ∧ Inv(next) ∧ Pre(this, head, next) ∧
¬ (Inv($next_3$) ∧ Post(this, head, $head_0$, next, $next_3$))

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

$next_0$ = next ++ ($near_0$ × far_0),
guard = (far_0 != null),
$next_1$ = $next_0$ ++ (mid_0 × $near_0$),
$near_1$ = mid_0,
mid_1 = far_0,
far_1 = far_0.$next_1$,

$near_2$ = **if** guard **then** $near_1$ **else** $near_0$,
mid_2 = **if** guard **then** mid_1 **else** mid_0,
far_2 = **if** guard **then** far_1 **else** far_0,
$next_2$ = **if** guard **then** $next_1$ **else** $next_0$,
$next_3$ = $next_2$ ++ (mid_2 × $near_2$)
$head_0$ = head ++ (this × mid_2) |

far_2 = null ∧ Inv(next) ∧ Pre(this, head, next) ∧
¬ (Inv($next_3$) ∧ Post(this, head, $head_0$, next, $next_3$))

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** }

upper bound

specifying analysis bounds

$this \subseteq List \land$ **one** $this$
$head \subseteq List \rightarrow (Node \cup null)$
$next \subseteq Node \rightarrow (Node \cup null)$
$data \subseteq Node \rightarrow (String \cup null)$

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

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

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

$far_2 = null \land Inv(next) \land Pre(this, head, next) \land$
$\neg (Inv(next_3) \land Post(this, head, head_0, next, next_3))$

universe

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

$\{ \langle null \rangle \} \subseteq null \subseteq \{ \langle null \rangle \}$

lower bound

$\{\} \subseteq this \subseteq \{ \langle this \rangle \}$
$\{\} \subseteq List \subseteq \{ \langle this \rangle \}$
$\{\} \subseteq Node \subseteq \{ \langle n0 \rangle, \langle n1 \rangle, \langle n2 \rangle \}$
$\{\} \subseteq String \subseteq \{ \langle s0 \rangle, \langle s1 \rangle, \langle s2 \rangle \}$

$\{\} \subseteq head \subseteq \{ this \} \times \{ n0, n1, n2, null \}$
$\{\} \subseteq next \subseteq \{ n0, n1, n2 \} \times \{ n0, n1, n2, null \}$
$\{\} \subseteq data \subseteq \{ n0, n1, n2 \} \times \{ s0, s1, s2, null \}$

upper bound

specifying analysis bounds

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

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

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

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

$far_2 = null \land Inv(next) \land Pre(this, head, next) \land$
$\neg (Inv(next_3) \land Post(this, head, head_0, next, next_3))$

universe

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

$\text{null} = \{ \langle \text{null} \rangle \}$

lower bound

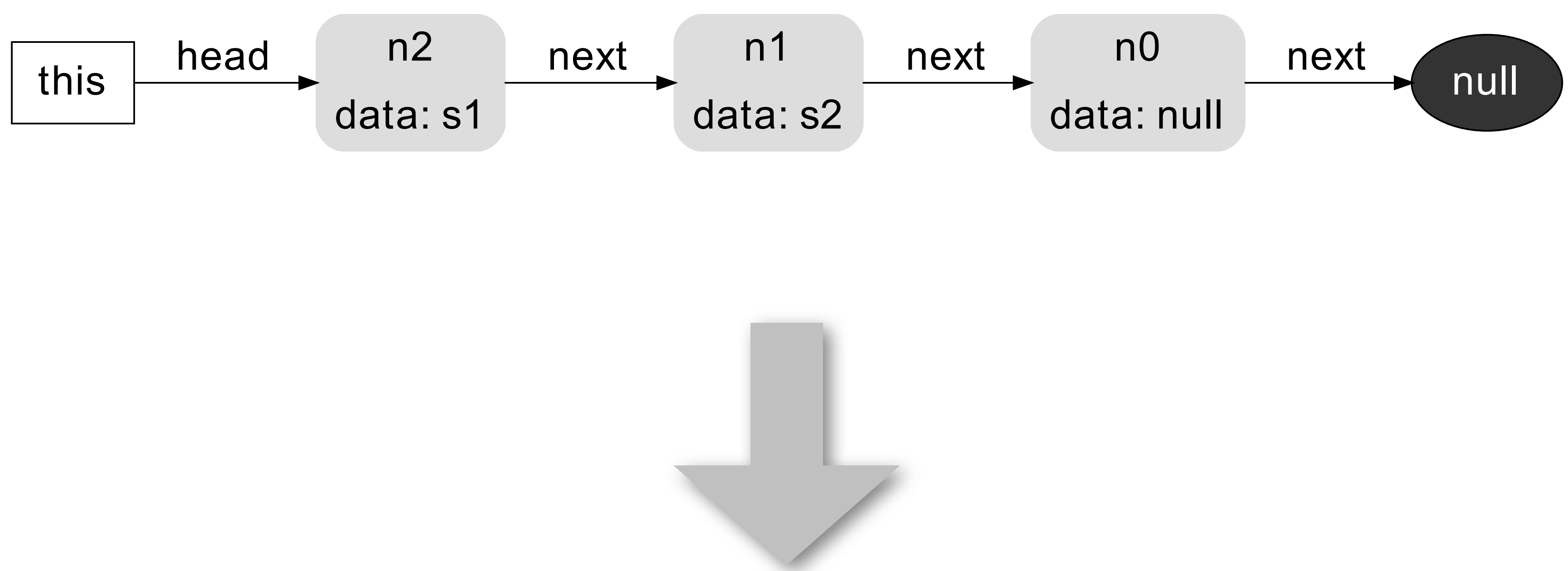
$\{\} \subseteq \text{this} \subseteq \{ \langle \text{this} \rangle \}$
$\{\} \subseteq \text{List} \subseteq \{ \langle \text{this} \rangle \}$
$\{\} \subseteq \text{Node} \subseteq \{ \langle n0 \rangle, \langle n1 \rangle, \langle n2 \rangle \}$
$\{\} \subseteq \text{String} \subseteq \{ \langle s0 \rangle, \langle s1 \rangle, \langle s2 \rangle \}$

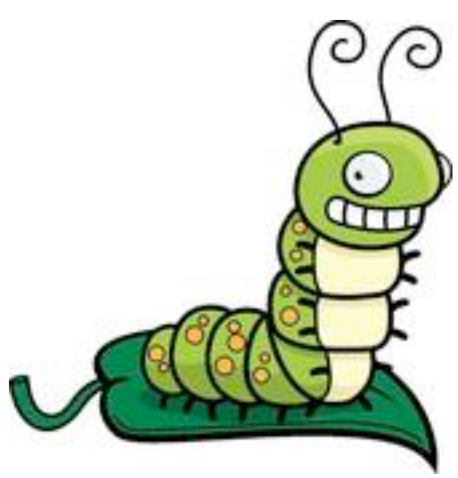
$\{\} \subseteq \text{head} \subseteq \{ \text{this} \} \times \{ n0, n1, n2, \text{null} \}$
$\{\} \subseteq \text{next} \subseteq \{ n0, n1, n2 \} \times \{ n0, n1, n2, \text{null} \}$
$\{\} \subseteq \text{data} \subseteq \{ n0, n1, n2 \} \times \{ s0, s1, s2, \text{null} \}$

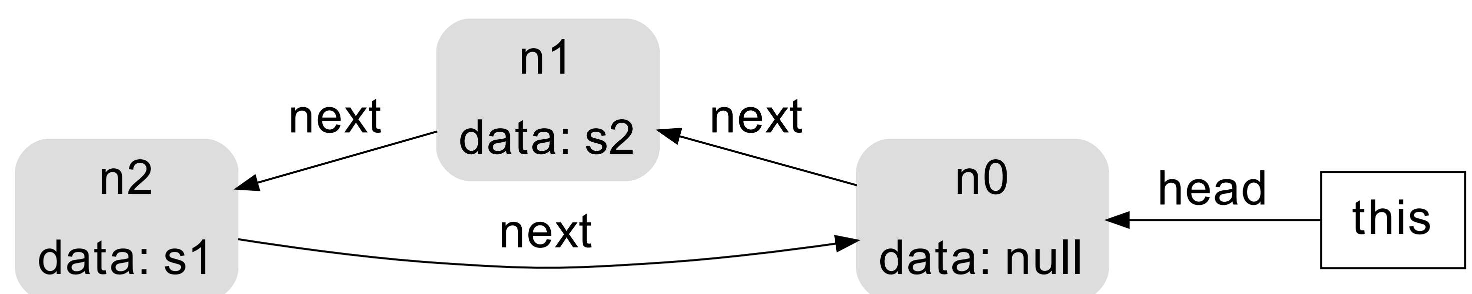
upper bound

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)

```
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)

```
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)

```
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);
}
```

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

$this \subseteq List \wedge \textbf{one}\ this \wedge$
$head \subseteq List \rightarrow (Node \cup null) \wedge$
$next \subseteq Node \rightarrow (Node \cup null) \wedge$
$data \subseteq Node \rightarrow (String \cup null) \wedge$

$head_0 \subseteq List \rightarrow (Node \cup null) \wedge$
$next_3 \subseteq Node \rightarrow (Node \cup null) \wedge$

$Inv(next) \wedge Pre(this, head, next) \wedge$
$Inv(next_3) \wedge Post(this, head, head_0, next, next_3)$

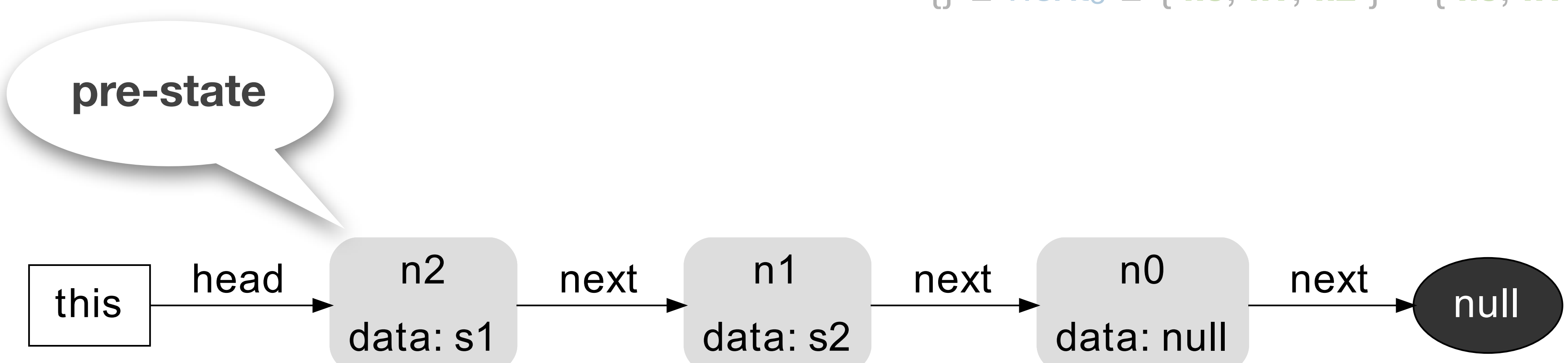
$\{\ this, n0, n1, n2, s0, s1, s2, null\ \}$

$null = \{ \langle null \rangle \}$

$this = \{ \langle this \rangle \}$
$List = \{ \langle this \rangle \}$
$Node = \{ \langle n0 \rangle, \langle n1 \rangle, \langle n2 \rangle \}$
$String = \{ \langle s1 \rangle, \langle s2 \rangle \}$

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

$\{\} \subseteq head_0 \subseteq \{ this \} \times \{ n0, n1, n2, null \}$
$\{\} \subseteq next_3 \subseteq \{ n0, n1, n2 \} \times \{ n0, n1, n2, null \}$



data repair using partial models

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

$\text{head}_0 \subseteq \text{List} \rightarrow (\text{Node} \cup \text{null}) \wedge$
$\text{next}_3 \subseteq \text{Node} \rightarrow (\text{Node} \cup \text{null}) \wedge$

$\text{Inv}(\text{next}) \wedge \text{Pre}(\text{this}, \text{head}, \text{next}) \wedge$
$\text{Inv}(\text{next}_3) \wedge \text{Post}(\text{this}, \text{head}, \text{head}_0, \text{next}, \text{next}_3)$

encoding of the repair

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

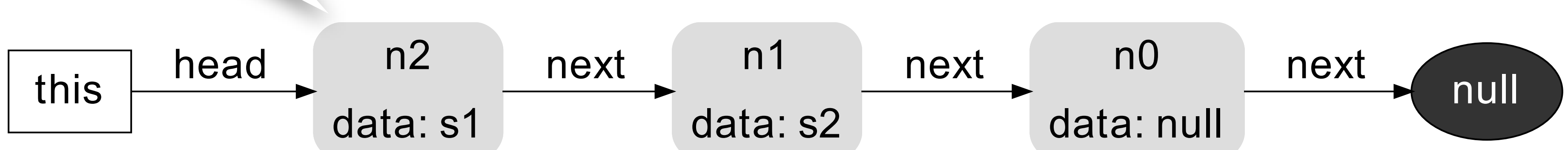
null = { <null> }

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> }

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

pre-state



data repair using partial models

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

$head_0 \subseteq List \rightarrow (Node \cup null) \land$
$next_3 \subseteq Node \rightarrow (Node \cup null) \land$

$Inv(next) \land Pre(this, head, next) \land$
$Inv(next_3) \land Post(this, head, head_0, next, next_3)$

post-state as variables

encoding of the repair

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

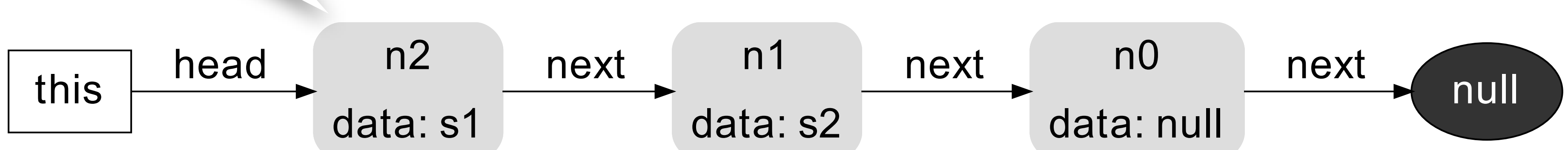
$null = \{ \langle null \rangle \}$

$this = \{ \langle this \rangle \}$
$List = \{ \langle this \rangle \}$
$Node = \{ \langle n0 \rangle, \langle n1 \rangle, \langle n2 \rangle \}$
$String = \{ \langle s1 \rangle, \langle s2 \rangle \}$

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

$\{\} \subseteq head_0 \subseteq \{ this \} \times \{ n0, n1, n2, null \}$
$\{\} \subseteq next_3 \subseteq \{ n0, n1, n2 \} \times \{ n0, n1, n2, null \}$

pre-state



data repair using partial models

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

$head_0 \subseteq List \rightarrow (Node \cup null) \land$
$next_3 \subseteq Node \rightarrow (Node \cup null) \land$

$Inv(next) \land Pre(this, head, next) \land$
$Inv(next_3) \land Post(this, head, head_0, next, next_3)$

post-state as variables

encoding of the repair

pre-state as a partial model; solve for post-state only

pre-state

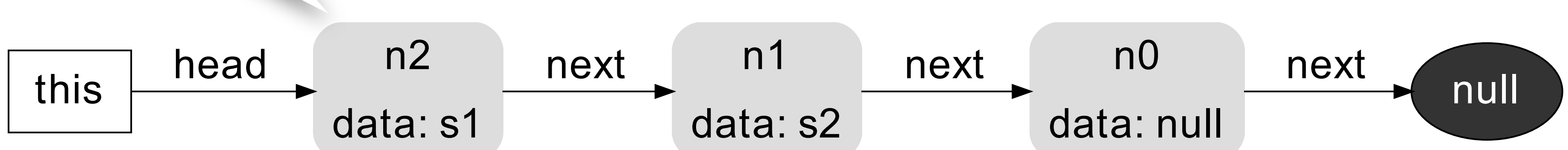
{ this, n0, n1, n2, s0, s1, s2, null }

null = { <null> }

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

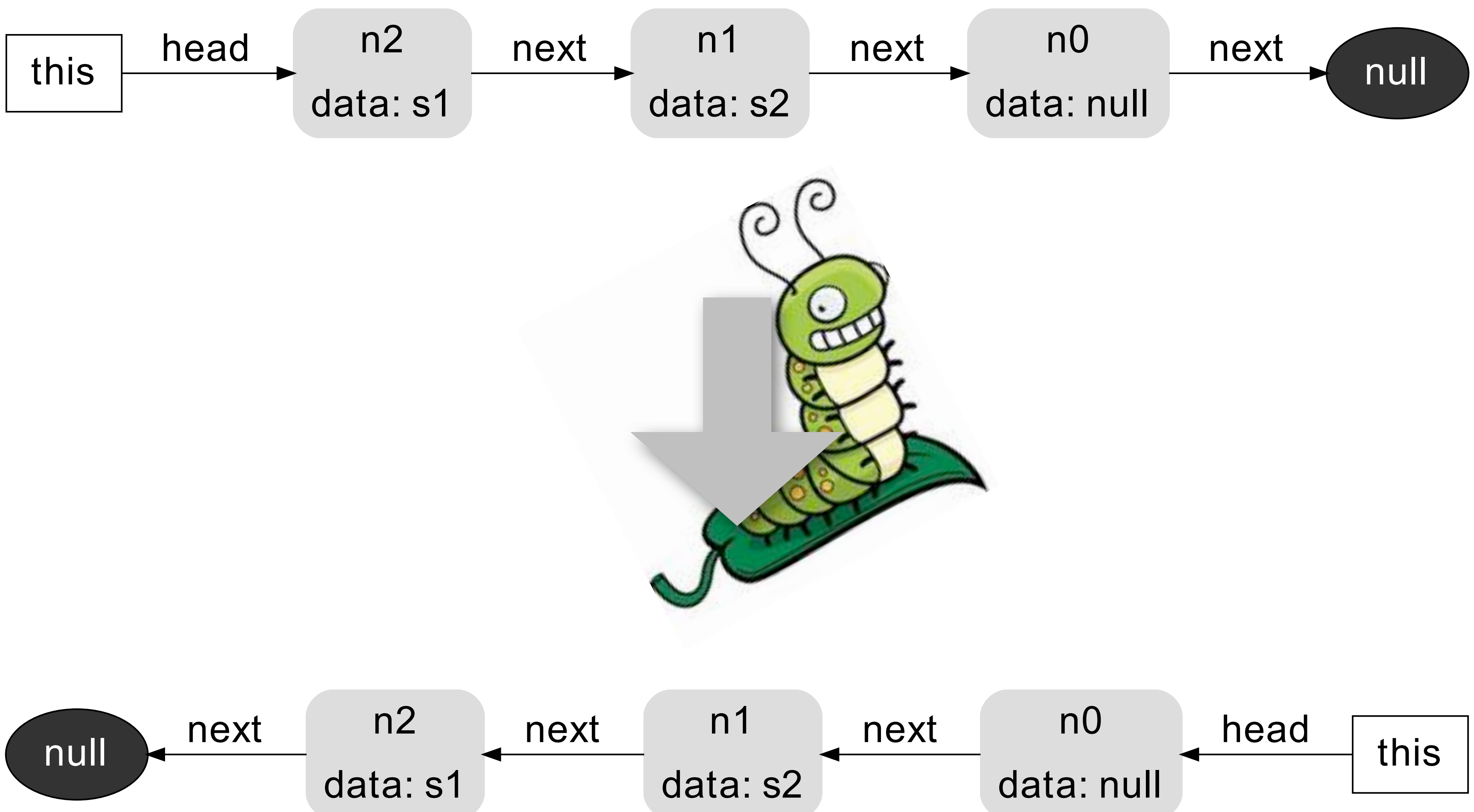
head = { <this, n2> }
next = { <n2, n1>, <n1, n0>, <n0, null> }
data = { <n2, s1>, <n1, s2>, <n0, null> }

$\{\} \subseteq head_0 \subseteq \{ this \} \times \{ n0, n1, n2, null \}$
$\{\} \subseteq next_3 \subseteq \{ n0, n1, n2 \} \times \{ n0, n1, n2, null \}$



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)

```
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, 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);
}
```

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);
}
```

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)

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);
}
```

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

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

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

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

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

  assume far2 == null;

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

given a buggy program, 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 near_0 = this.head;
   Node mid_0 = near_0.next;
   Node far_0 = mid_0.next;

   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);
}
```

this $\subseteq$ List $\wedge$ **one** this $\wedge$
head $\subseteq$ List $\rightarrow$ (Node $\cup$ null) $\wedge$
next $\subseteq$ Node $\rightarrow$ (Node $\cup$ null) $\wedge$
data $\subseteq$ Node $\rightarrow$ (String $\cup$ null) $\wedge$

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 != null),
$next_1 = next_0$ ++ ($mid_0 \times near_0$),
$near_1 = mid_0$,
$mid_1 = far_0$,
$far_1 = far_0.next_1$,

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

far_2 = null $\wedge$ Inv(next) $\wedge$ Pre(this, head, next) $\wedge$
$\neg$ (Inv($next_3$) $\wedge$ Post(this, head, $head_0$, next, $next_3$))

start with the encoding for validation

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);
}
```

$\mathit{this} \subseteq \mathit{List} \wedge \textbf{one}\ \mathit{this} \wedge$
$\mathit{head} \subseteq \mathit{List} \rightarrow (\mathit{Node} \cup \mathit{null}) \wedge$
$\mathit{next} \subseteq \mathit{Node} \rightarrow (\mathit{Node} \cup \mathit{null}) \wedge$
$\mathit{data} \subseteq \mathit{Node} \rightarrow (\mathit{String} \cup \mathit{null}) \wedge$

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

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

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

$far_2 = null \wedge Inv(next) \wedge Pre(this, head, next) \wedge$
$Inv(next_3) \wedge Post(this, head, head_0, next, next_3)$

post-condition must hold

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);
   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);
}
```

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

$\textbf{let}\ \text{near}_0 = \text{this.head},$
$\text{mid}_0 = \text{near}_0.\text{next},$
$\text{far}_0 = \text{mid}_0.\text{next},$

$\text{next}_0 = \text{next} \mathbin{++} (\text{near}_0 \times \text{far}_0),$
$\text{guard} = (\text{far}_0 \mathrel{!=} \text{null}),$
$\text{next}_1 = \text{next}_0 \mathbin{++} (\text{mid}_0 \times \text{near}_0),$
$\text{near}_1 = \text{mid}_0,$
$\text{mid}_1 = \text{far}_0,$
$\text{far}_1 = \text{far}_0.\text{next}_1,$

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

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

introduce free variables for source-level expressions

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);
}
```

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

$\text{near}_0 = \text{this.head} \wedge$
$\text{mid}_0 = \text{near}_0\text{.next} \wedge$
$\text{far}_0 = \text{mid}_0\text{.next} \wedge$

introduce free variables for source-level expressions

$\text{next}_0 = \text{next} \mathbin{++} (\text{near}_0 \times \text{far}_0) \wedge$
$\text{next}_1 = \text{next}_0 \mathbin{++} (\text{mid}_0 \times \text{near}_0) \wedge$
$\text{near}_1 = \text{mid}_0 \wedge$
$\text{mid}_1 = \text{far}_0 \wedge$
$\text{far}_1 = \text{far}_0\text{.next}_1 \wedge$

$\textbf{let}\ \text{guard} = (\text{far}_0 \mathrel{!=} \text{null}),$
$\quad \text{near}_2 = \textbf{if}\ \text{guard}\ \textbf{then}\ \text{near}_1\ \textbf{else}\ \text{near}_0,$
$\quad \text{mid}_2 = \textbf{if}\ \text{guard}\ \textbf{then}\ \text{mid}_1\ \textbf{else}\ \text{mid}_0,$
$\quad \text{far}_2 = \textbf{if}\ \text{guard}\ \textbf{then}\ \text{far}_1\ \textbf{else}\ \text{far}_0,$
$\quad \text{next}_2 = \textbf{if}\ \text{guard}\ \textbf{then}\ \text{next}_1\ \textbf{else}\ \text{next}_0 \mid$

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

fault localization encoding: partial model

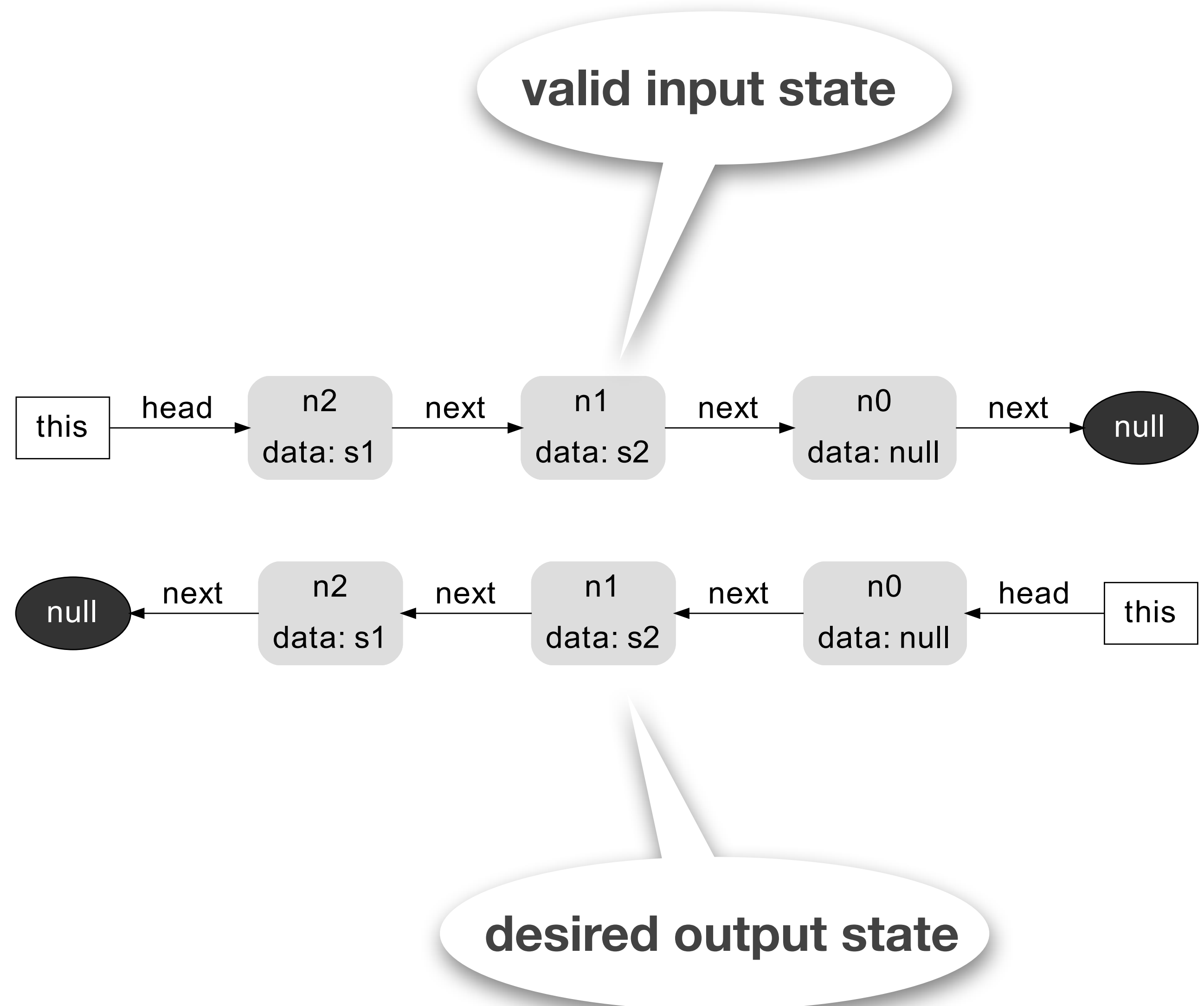
$this \subseteq List \wedge$ **one** $this \wedge$
$head \subseteq List \rightarrow (Node \cup null) \wedge$
$next \subseteq Node \rightarrow (Node \cup null) \wedge$
$data \subseteq Node \rightarrow (String \cup null) \wedge$

$near_0 = this.head \wedge$
$mid_0 = near_0.next \wedge$
$far_0 = mid_0.next \wedge$

$next_0 = next \mathbin{++} (near_0 \times far_0) \wedge$
$next_1 = next_0 \mathbin{++} (mid_0 \times near_0) \wedge$
$near_1 = mid_0 \wedge$
$mid_1 = far_0 \wedge$
$far_1 = far_0.next_1 \wedge$

let $guard = (far_0 \mathrel{!=} null)$,
$near_2 =$ **if** $guard$ **then** $near_1$ **else** $near_0$,
$mid_2 =$ **if** $guard$ **then** mid_1 **else** mid_0,
$far_2 =$ **if** $guard$ **then** far_1 **else** far_0,
$next_2 =$ **if** $guard$ **then** $next_1$ **else** $next_0$ **|**

$next_3 = next_2 \mathbin{++} (mid_2 \times near_2) \wedge$
$head_0 = head \mathbin{++} (this \times mid_2) \wedge$
$far_2 = null \wedge Inv(next) \wedge Pre(this, head, next) \wedge$
$Inv(next_3) \wedge Post(this, head, head_0, next, next_3)$



fault localization encoding: partial model

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

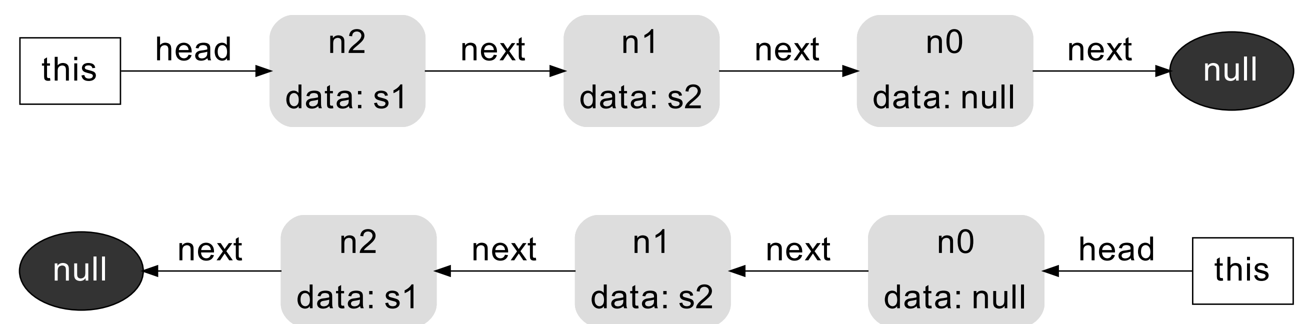
$\text{near}_0 = \text{this.head} \wedge$
$\text{mid}_0 = \text{near}_0.\text{next} \wedge$
$\text{far}_0 = \text{mid}_0.\text{next} \wedge$

$\text{next}_0 = \text{next} \mathbin{++} (\text{near}_0 \times \text{far}_0) \wedge$
$\text{next}_1 = \text{next}_0 \mathbin{++} (\text{mid}_0 \times \text{near}_0) \wedge$
$\text{near}_1 = \text{mid}_0 \wedge$
$\text{mid}_1 = \text{far}_0 \wedge$
$\text{far}_1 = \text{far}_0.\text{next}_1 \wedge$

$\textbf{let}\ \text{guard} = (\text{far}_0 \mathrel{!=} \text{null}),$
$\quad \text{near}_2 = \textbf{if}\ \text{guard}\ \textbf{then}\ \text{near}_1\ \textbf{else}\ \text{near}_0,$
$\quad \text{mid}_2 = \textbf{if}\ \text{guard}\ \textbf{then}\ \text{mid}_1\ \textbf{else}\ \text{mid}_0,$
$\quad \text{far}_2 = \textbf{if}\ \text{guard}\ \textbf{then}\ \text{far}_1\ \textbf{else}\ \text{far}_0,$
$\quad \text{next}_2 = \textbf{if}\ \text{guard}\ \textbf{then}\ \text{next}_1\ \textbf{else}\ \text{next}_0 \mid$

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

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



fault localization encoding: partial model

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

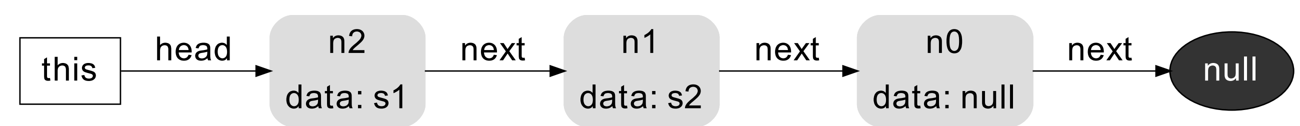
$\text{near}_0 = \text{this.head} \wedge$
$\text{mid}_0 = \text{near}_0\text{.next} \wedge$
$\text{far}_0 = \text{mid}_0\text{.next} \wedge$

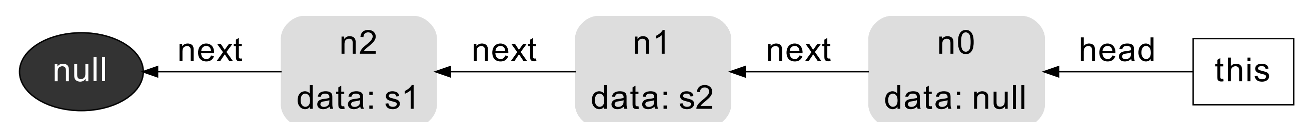
$\text{next}_0 = \text{next} \mathbin{++} (\text{near}_0 \times \text{far}_0) \wedge$
$\text{next}_1 = \text{next}_0 \mathbin{++} (\text{mid}_0 \times \text{near}_0) \wedge$
$\text{near}_1 = \text{mid}_0 \wedge$
$\text{mid}_1 = \text{far}_0 \wedge$
$\text{far}_1 = \text{far}_0\text{.next}_1 \wedge$

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

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

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





$\{\} \subseteq \text{next}_0 \subseteq \{\ \textbf{n0}, \textbf{n1}, \textbf{n2}\ \} \times \{\ \textbf{n0}, \textbf{n1}, \textbf{n2}, \textbf{null}\ \}$
$\{\} \subseteq \text{next}_1 \subseteq \{\ \textbf{n0}, \textbf{n1}, \textbf{n2}\ \} \times \{\ \textbf{n0}, \textbf{n1}, \textbf{n2}, \textbf{null}\ \}$
$\{\} \subseteq \text{near}_0 \subseteq \{\ \textbf{n0}, \textbf{n1}, \textbf{n2}, \textbf{null}\ \}$
$\{\} \subseteq \text{near}_1 \subseteq \{\ \textbf{n0}, \textbf{n1}, \textbf{n2}, \textbf{null}\ \}$
$\{\} \subseteq \text{mid}_0 \subseteq \{\ \textbf{n0}, \textbf{n1}, \textbf{n2}, \textbf{null}\ \}$
$\{\} \subseteq \text{mid}_1 \subseteq \{\ \textbf{n0}, \textbf{n1}, \textbf{n2}, \textbf{null}\ \}$
$\{\} \subseteq \text{far}_0 \subseteq \{\ \textbf{n0}, \textbf{n1}, \textbf{n2}, \textbf{null}\ \}$
$\{\} \subseteq \text{far}_1 \subseteq \{\ \textbf{n0}, \textbf{n1}, \textbf{n2}, \textbf{null}\ \}$

fault localization encoding: partial model

$this \subseteq List \wedge$ **one** $this \wedge$
$head \subseteq List \to (Node \cup null) \wedge$
$next \subseteq Node \to (Node \cup null) \wedge$
$data \subseteq Node \to (String \cup null) \wedge$

$near_0 = this.head \wedge$
$mid_0 = near_0.next \wedge$
$far_0 = mid_0.next \wedge$

$next_0 = next \mathbin{++} (near_0 \times far_0) \wedge$
$next_1 = next_0 \mathbin{++} (mid_0 \times near_0) \wedge$
$near_1 = mid_0 \wedge$
$mid_1 = far_0 \wedge$
$far_1 = far_0.next_1 \wedge$

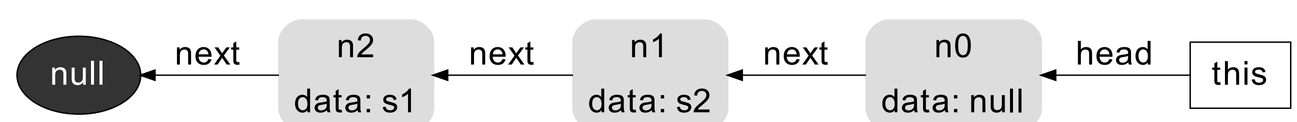
let guard = $(far_0 \mathrel{!=} null)$,
$near_2 =$ **if** guard **then** $near_1$ **else** $near_0$,
$mid_2 =$ **if** guard **then** mid_1 **else** mid_0,
$far_2 =$ **if** guard **then** far_1 **else** far_0,
$next_2 =$ **if** guard **then** $next_1$ **else** $next_0$ |

$next_3 = next_2 \mathbin{++} (mid_2 \times near_2) \wedge$
$head_0 = head \mathbin{++} (this \times mid_2) \wedge$
$far_2 = null \wedge Inv(next) \wedge Pre(this, head, next) \wedge$
$Inv(next_3) \wedge Post(this, head, head_0, next, next_3)$

{ **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**> }



$\{\} \subseteq next_0 \subseteq \{ n0, n1, n2 \} \times \{ n0, n1, n2, null \}$
$\{\} \subseteq next_1 \subseteq \{ n0, n1, n2 \} \times \{ n0, n1, n2, null \}$
$\{\} \subseteq near_0 \subseteq \{ n0, n1, n2, null \}$
$\{\} \subseteq near_1 \subseteq \{ n0, n1, n2, null \}$
$\{\} \subseteq mid_0 \subseteq \{ n0, n1, n2, null \}$
$\{\} \subseteq mid_1 \subseteq \{ n0, n1, n2, null \}$
$\{\} \subseteq far_0 \subseteq \{ n0, n1, n2, null \}$
$\{\} \subseteq far_1 \subseteq \{ n0, n1, n2, null \}$

fault localization encoding: partial model

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

$\text{near}_0 = \text{this.head} \wedge$
$\text{mid}_0 = \text{near}_0\text{.next} \wedge$
$\text{far}_0 = \text{mid}_0\text{.next} \wedge$

$\text{next}_0 = \text{next} \mathbin{++} (\text{near}_0 \times \text{far}_0) \wedge$
$\text{next}_1 = \text{next}_0 \mathbin{++} (\text{mid}_0 \times \text{near}_0) \wedge$
$\text{near}_1 = \text{mid}_0 \wedge$
$\text{mid}_1 = \text{far}_0 \wedge$
$\text{far}_1 = \text{far}_0\text{.next}_1 \wedge$

let guard = (far_0 != null),
$\text{near}_2 =$ **if** guard **then** near_1 **else** near_0,
$\text{mid}_2 =$ **if** guard **then** mid_1 **else** mid_0,
$\text{far}_2 =$ **if** guard **then** far_1 **else** far_0,
$\text{next}_2 =$ **if** guard **then** next_1 **else** next_0 **|**

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

{ **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**> }

head_0 = { <**this**, **n0**> }
next_3 = { <**n0**, **n1**>, <**n1**, **n2**>, <**n2**, **null**> }

$\{\} \subseteq \text{next}_0 \subseteq \{\textbf{n0}, \textbf{n1}, \textbf{n2}\} \times \{\textbf{n0}, \textbf{n1}, \textbf{n2}, \textbf{null}\}$
$\{\} \subseteq \text{next}_1 \subseteq \{\textbf{n0}, \textbf{n1}, \textbf{n2}\} \times \{\textbf{n0}, \textbf{n1}, \textbf{n2}, \textbf{null}\}$
$\{\} \subseteq \text{near}_0 \subseteq \{\textbf{n0}, \textbf{n1}, \textbf{n2}, \textbf{null}\}$
$\{\} \subseteq \text{near}_1 \subseteq \{\textbf{n0}, \textbf{n1}, \textbf{n2}, \textbf{null}\}$
$\{\} \subseteq \text{mid}_0 \subseteq \{\textbf{n0}, \textbf{n1}, \textbf{n2}, \textbf{null}\}$
$\{\} \subseteq \text{mid}_1 \subseteq \{\textbf{n0}, \textbf{n1}, \textbf{n2}, \textbf{null}\}$
$\{\} \subseteq \text{far}_0 \subseteq \{\textbf{n0}, \textbf{n1}, \textbf{n2}, \textbf{null}\}$
$\{\} \subseteq \text{far}_1 \subseteq \{\textbf{n0}, \textbf{n1}, \textbf{n2}, \textbf{null}\}$

fault localization demo

minimal unsatisfiable core

$this \subseteq List \wedge$ **one** $this \wedge$
$head \subseteq List \rightarrow (Node \cup null) \wedge$
$next \subseteq Node \rightarrow (Node \cup null) \wedge$
$data \subseteq Node \rightarrow (String \cup null) \wedge$

$near_0 = this.head \wedge$
$mid_0 = near_0.next \wedge$
$far_0 = mid_0.next \wedge$

$next_0 = next \mathbin{++} (near_0 \times far_0) \wedge$
$next_1 = next_0 \mathbin{++} (mid_0 \times near_0) \wedge$
$near_1 = mid_0 \wedge$
$mid_1 = far_0 \wedge$
$far_1 = far_0.next_1 \wedge$

let guard = (far_0 != null),
$near_2$ = **if** guard **then** $near_1$ **else** $near_0$,
mid_2 = **if** guard **then** mid_1 **else** mid_0,
far_2 = **if** guard **then** far_1 **else** far_0,
$next_2$ = **if** guard **then** $next_1$ **else** $next_0$ |

$next_3 = next_2 \mathbin{++} (mid_2 \times near_2) \wedge$
$head_0 = head \mathbin{++} (this \times mid_2) \wedge$
$far_2 = null \wedge Inv(next) \wedge Pre(this, head, next) \wedge$
$Inv(next_3) \wedge Post(this, head, head_0, next, next_3)$

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> }

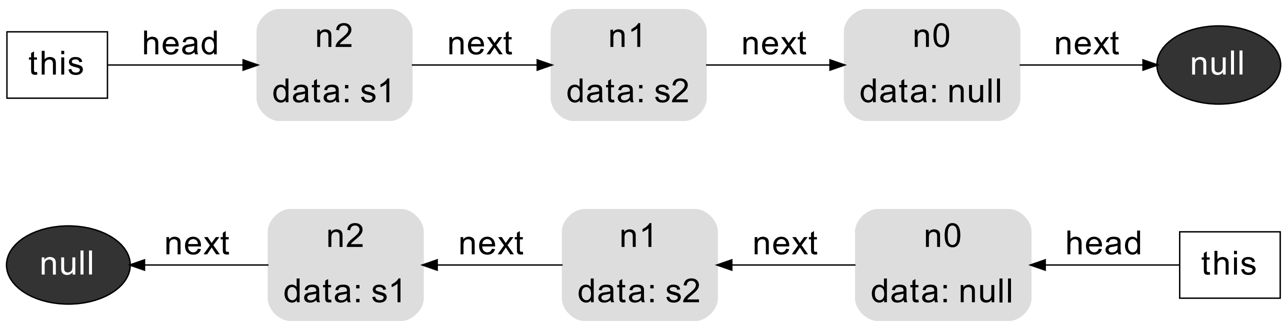
$head_0$ = { <this, n0> }
$next_3$ = { <n0, n1>, <n1, n2>, <n2, null> }

$\{\} \subseteq next_0 \subseteq$ { n0, n1, n2 } × { n0, n1, n2, null }
$\{\} \subseteq next_1 \subseteq$ { n0, n1, n2 } × { n0, n1, n2, null }
$\{\} \subseteq near_0 \subseteq$ { n0, n1, n2, null }
$\{\} \subseteq near_1 \subseteq$ { n0, n1, n2, null }
$\{\} \subseteq mid_0 \subseteq$ { n0, n1, n2, null }
$\{\} \subseteq mid_1 \subseteq$ { n0, n1, n2, null }
$\{\} \subseteq far_0 \subseteq$ { n0, n1, n2, null }
$\{\} \subseteq far_1 \subseteq$ { n0, n1, n2, null }

minimal unsatisfiable core

@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);
}
```



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, 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 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)

```
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)

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);
}
```

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)

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);
}
```

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

$\textbf{let}\ \text{near}_0 = \text{this.head},$
$\text{mid}_0 = \text{near}_0.\text{next},$
$\text{far}_0 = \text{mid}_0.\text{next},$

$\text{next}_0 = \text{next} \mathbin{++} (\text{near}_0 \times \text{far}_0),$
$\text{guard} = (\text{far}_0 \mathrel{!=} \text{null}),$
$\text{next}_1 = \text{next}_0 \mathbin{++} (\text{mid}_0 \times \text{near}_0),$
$\text{near}_1 = \text{mid}_0,$
$\text{mid}_1 = \text{far}_0,$
$\text{far}_1 = \text{far}_0.\text{next}_1,$

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

$\text{far}_2 = \text{null} \wedge \text{Inv}(\text{next}) \wedge \text{Pre}(\text{this}, \text{head}, \text{next}) \wedge$
$\text{Inv}(\text{next}_3) \wedge \text{Post}(\text{this}, \text{head}, \text{head}_0, \text{next}, \text{next}_3)$

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)

```
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);
}
```

$\text{this} \subseteq \text{List} \wedge \textbf{one}\ \text{this} \wedge \textbf{one}\ \text{hole} \wedge$
$\text{head} \subseteq \text{List} \rightarrow (\text{Node} \cup \text{null}) \wedge$
$\text{next} \subseteq \text{Node} \rightarrow (\text{Node} \cup \text{null}) \wedge$
$\text{data} \subseteq \text{Node} \rightarrow (\text{String} \cup \text{null}) \wedge$

encode the hole using syntax relations

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

meaning = $(\text{null} \times \text{null}) \cup (\text{“head”} \times \text{this.head}) \cup (\text{“near}_0\text{”} \times \text{near}_0) \cup (\text{“mid}_0\text{”} \times \text{mid}_0) \cup (\text{“far}_0\text{”} \times \text{far}_0)$

$\text{next}_0 = \text{next} +\!+ (\text{near}_0 \times \text{hole.meaning})$,
$\text{guard} = (\text{far}_0 \mathrel{!=} \text{null})$,
$\text{next}_1 = \text{next}_0 +\!+ (\text{mid}_0 \times \text{near}_0)$,
$\text{near}_1 = \text{mid}_0$,
$\text{mid}_1 = \text{far}_0$,
$\text{far}_1 = \text{far}_0.\text{next}_1$,

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

$\text{far}_2 = \text{null} \wedge \text{Inv}(\text{next}) \wedge \text{Pre}(\text{this}, \text{head}, \text{next}) \wedge \text{Inv}(\text{next}_3) \wedge \text{Post}(\text{this}, \text{head}, \text{head}_0, \text{next}, \text{next}_3)$

synthesis encoding: partial model

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

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

meaning = (null × null) ∪ (“head” × this.head) ∪
(“$near_0$” × $near_0$) ∪ (“mid_0” × mid_0) ∪ (“far_0” × far_0)

$next_0$ = next ++ ($near_0$ × hole.meaning),
guard = (far_0 != null),
$next_1$ = $next_0$ ++ (mid_0 × $near_0$),
$near_1$ = mid_0,
mid_1 = far_0,
far_1 = far_0.$next_1$,

$near_2$ = **if** guard **then** $near_1$ **else** $near_0$,
mid_2 = **if** guard **then** mid_1 **else** mid_0,
far_2 = **if** guard **then** far_1 **else** far_0,
$next_2$ = **if** guard **then** $next_1$ **else** $next_0$,
$next_3$ = $next_2$ ++ (mid_2 × $near_2$)
$head_0$ = head ++ (this × mid_2) |

far_2 = null ∧ Inv(next) ∧ Pre(this, head, next) ∧
Inv($next_3$) ∧ Post(this, head, $head_0$, next, $next_3$)

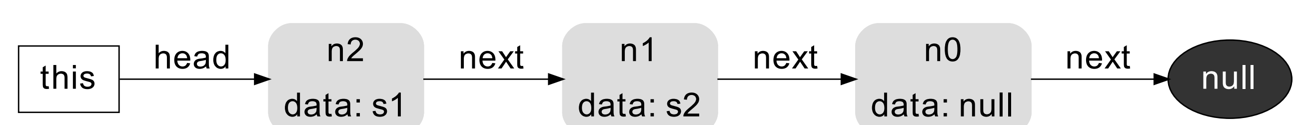
{ **this**, **n0**, **n1**, **n2**, **s0**, **s1**, **s2**, **null**,
“head”, **“$near_0$”**, **“mid_0”**, **“far_0”** }

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”**> }
“$near_0$” = { <**“$near_0$”**> }
“mid_0” = { <**“mid_0”**> }
“far_0” = { <**“far_0”**> }

{} ⊆ hole ⊆ { <**null**>, <**“head”**>, <**“$near_0$”**>,
<**“mid_0”**>, <**“far_0”**> }



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;
}
```

alloy.mit.edu/kodkod

alloy.mit.edu/kodkod