Precise Reasoning with Structured Heaps and Collective Operations à la Map/Reduce

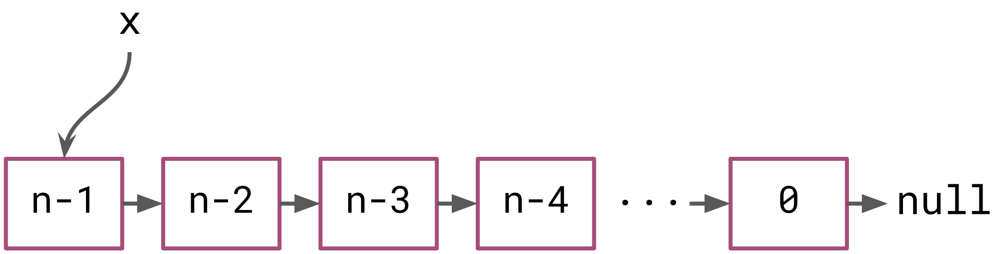
Gregory Essertel, **Guannan Wei**, Tiark Rompf
Department of Computer Science, Purdue University
Jan 2018, PurPL Seminar

Motivation

```
ListNode x = null; int i = 0
while (i < n) {
    ListNode y = new ListNode()
    y.tail = x
    y.head = i
    x = y
    i = i + 1
}

ListNode z = x; int sum = 0
while (z != null) {
    sum = sum + z.head
    z = z.tail
}

assert(sum == n*(n-1)/2)
```

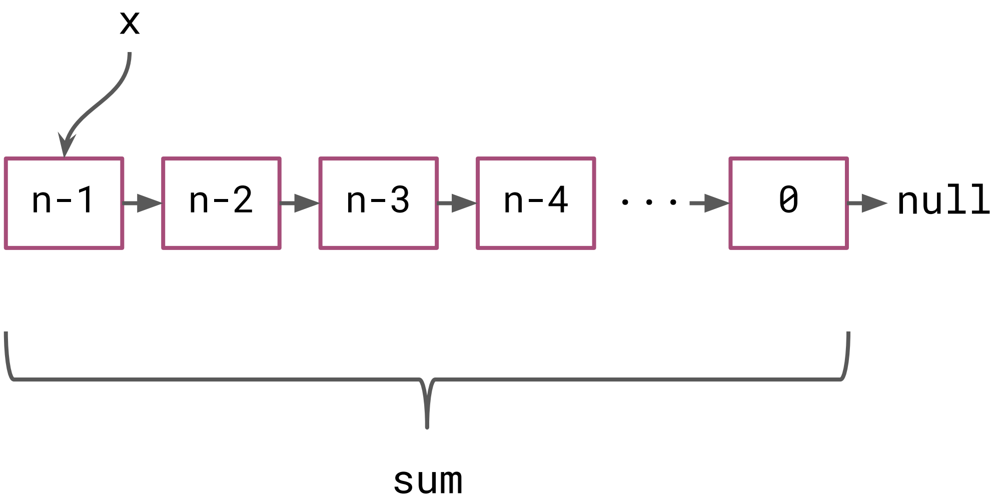
Motivation

```
ListNode x = null; int i = 0
while (i < n) {
    ListNode y = new ListNode()
    y.tail = x
    y.head = i
    x = y
    i = i + 1
}

ListNode z = x; int sum = 0
while (z != null) {
    sum = sum + z.head
    z = z.tail
}

assert(sum == n*(n-1)/2)
```



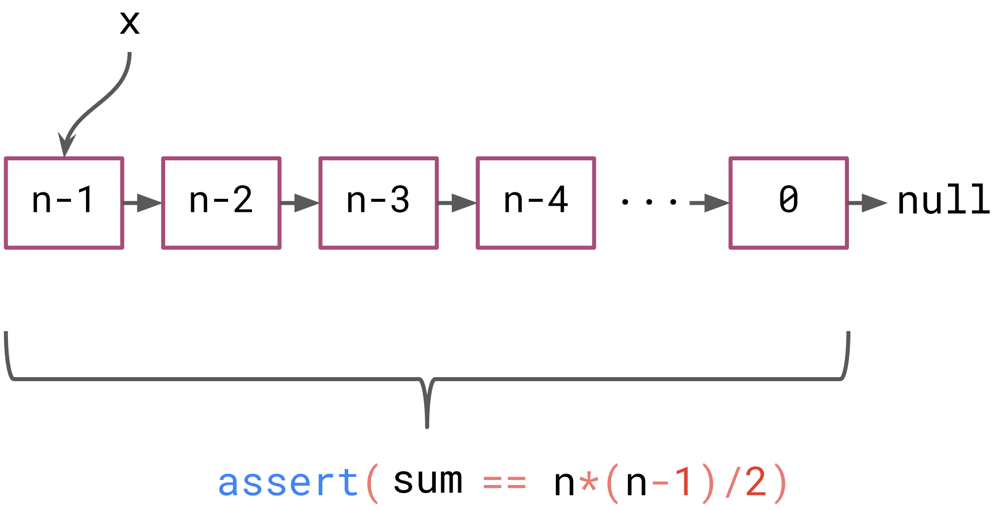
Motivation

```
ListNode x = null; int i = 0
while (i < n) {
    ListNode y = new ListNode()
    y.tail = x
    y.head = i
    x = y
    i = i + 1
}

ListNode z = x; int sum = 0
while (z != null) {
    sum = sum + z.head
    z = z.tail
}

assert(sum == n*(n-1)/2)
```



Motivation

```
ListNode x = null; int i = 0
while (i < n) {
    ListNode y = new ListNode()
    y.tail = x
    y.head = i
    x = y
    i = i + 1
}

ListNode z = x; int sum = 0
while (z != null) {
    sum = sum + z.head
    z = z.tail
}

assert(sum == n*(n-1)/2)
```



Motivation

```
ListNode x = null; int i = 0
while (i < n) {
    ListNode y = new ListNode()
    y.tail = x
    y.head = i
    x = y
    i = i + 1
}

ListNode z = x; int sum = 0
while (z != null) {
    sum = sum + z.head
    z = z.tail
}

assert(sum == n*(n-1)/2)
```

Many techniques failed.

Motivation

```
ListNode x = null; int i = 0
while (i < n) {
    ListNode y = new ListNode()
    y.tail = x
    y.head = i
    x = y
    i = i + 1
}

ListNode z = x; int sum = 0
while (z != null) {
    sum = sum + z.head
    z = z.tail
}

assert(sum == n*(n-1)/2)
```

Many techniques failed.

- program abstractions are usually low-level *scalars*, rather than collections.
  e.g., a linked list contains natural numbers from `0` to `n-1`.

Motivation

```
ListNode x = null; int i = 0
while (i < n) {
    ListNode y = new ListNode()
    y.tail = x
    y.head = i
    x = y
    i = i + 1
}

ListNode z = x; int sum = 0
while (z != null) {
    sum = sum + z.head
    z = z.tail
}

assert(sum == n*(n-1)/2)
```

Many techniques failed.

- program abstractions are usually low-level *scalars*, rather than collections.
  e.g., a linked list contains natural numbers from `0` to `n-1`.

- program abstractions lose the information of *time*.
  e.g., values at different loop iterations (loop context) are not distinguished.

Our Solution

- Borrow ideas from Domain Specific Languages (DSLs)
  - Translate low-level imperative program to high level functional program with preserved semantics
- Introduce first-class collective forms
  - The loop iteration index is an argument rather than a free variable

Our Solution

- Borrow ideas from Domain Specific Languages (DSLs)
  - Translate low-level imperative program to high level functional program with preserved semantics
- Introduce first-class collective forms
  - The loop iteration index is an argument rather than a free variable

```
IMP:

int j = 0, sum = 0
while (j < k) {
    sum = sum + j
    j = j + 1
}
```

Our Solution

- Borrow ideas from Domain Specific Languages (DSLs)
  - Translate low-level imperative program to high level functional program with preserved semantics
- Introduce first-class collective forms
  - The loop iteration index is an argument rather than a free variable

IMP:

```
int j = 0, sum = 0
while (j < k) {
    sum = sum + j
    j = j + 1
}
```

→

FUN:

```
let j = λ(i).if (i > 0)
             then j(i-1)+1
             else 0
let sum = λ(i).if (i > 0)
               then sum(i-1)+j(i-1)
               else 0
let n = idx(i).!(j(i) <= k)
```

Our Solution

- Borrow ideas from Domain Specific Languages (DSLs)
  - Translate low-level imperative program to high level functional program with preserved semantics
- Introduce first-class collective forms
  - The loop iteration index is an argument rather than a free variable

IMP:

```
int j = 0, sum = 0
while (j < k) {
    sum = sum + j        →
    j = j + 1
}
```

FUN:

```
let j = λ(i).if (i > 0)
              then j(i-1)+1
              else 0
let sum = λ(i).if (i > 0)
                then sum(i-1)+j(i-1)
                else 0
let n = idx(i).!(j(i) <= k)
```

STORE:

```
j→if (n > 0) then j(n-1)
                else 0
sum→if (n > 0) then sum(n-1)
                  else 0
```

Our Solution

- Borrow ideas from Domain Specific Languages (DSLs)
    - Translate low-level imperative program to high level functional program with preserved semantics
- Introduce first-class collective forms
    - The loop iteration index is an argument rather than a free variable

IMP:

```
int j = 0, sum = 0
while (j < k) {
    sum = sum + j
    j = j + 1
}
```

→

FUN:

```
let j = λ(i).if (i > 0)
             then j(i-1)+1
             else 0
let sum = λ(i).if (i > 0)
               then sum(i-1)+j(i-1)
               else 0
let n = idx(i).!(j(i) <= k)
```

STORE:

```
j→if (n > 0) then j(n-1)
              else 0
sum→if (n > 0) then sum(n-1)
                else 0
```

Our Solution

- Borrow ideas from Domain Specific Languages (DSLs)
  - Translate low-level imperative program to high level functional program with preserved semantics
- Introduce first-class collective forms
  - The loop iteration index is an argument rather than a free variable

IMP:

```
int j = 0, sum = 0
while (j < k) {
    sum = sum + j
    j = j + 1
}
```

→

FUN:

```
let j = λ(i).Σ(i2 < i + 1){ 1 }


let sum = λ(i).if (i > 0)
               then sum(i-1)+j(i-1)
               else 0
let n = idx(i).!(j(i) <= k)
```

STORE:

```
j→if (n > 0) then j(n-1)
              else 0
sum→if (n > 0) then sum(n-1)
                else 0
```

Our Solution

- Borrow ideas from Domain Specific Languages (DSLs)
  - Translate low-level imperative program to high level functional program with preserved semantics
- Introduce first-class collective forms
  - The loop iteration index is an argument rather than a free variable

IMP:

```
int j = 0, sum = 0
while (j < k) {
    sum = sum + j
    j = j + 1
}
```

FUN:

a fresh var of Σ

```
let j = λ(i).Σ(i2 < i + 1){ 1 }

let sum = λ(i).if (i > 0)
               then sum(i-1)+j(i-1)
               else 0
let n = idx(i).!(j(i) <= k)
```

STORE:

```
j→if (n > 0) then j(n-1)
              else 0
sum→if (n > 0) then sum(n-1)
                else 0
```

Our Solution

- Borrow ideas from Domain Specific Languages (DSLs)
  - Translate low-level imperative program to high level functional program with preserved semantics
- Introduce first-class collective forms
  - The loop iteration index is an argument rather than a free variable

IMP:

```
int j = 0, sum = 0
while (j < k) {
    sum = sum + j
    j = j + 1
}
```

→

FUN:

```
let j = λ(i). Σ_0^{i+1} 1


let sum = λ(i).if (i > 0)
                then sum(i-1)+j(i-1)
                else 0
let n = idx(i).!(j(i) <= k)
```

STORE:

```
j→if (n > 0) then j(n-1)
                else 0
sum→if (n > 0) then sum(n-1)
                  else 0
```

Our Solution

- Borrow ideas from Domain Specific Languages (DSLs)
  - Translate low-level imperative program to high level functional program with preserved semantics
- Introduce first-class collective forms
  - The loop iteration index is an argument rather than a free variable

IMP:

```
int j = 0, sum = 0
while (j < k) {
    sum = sum + j
    j = j + 1
}
```

→

FUN:

```
let j = λ(i).i + 1


let sum = λ(i).if (i > 0)
                then sum(i-1)+j(i-1)
                else 0
let n = idx(i).!(j(i) <= k)
```

STORE:

```
j→if (n > 0) then j(n-1)
              else 0
sum→if (n > 0) then sum(n-1)
                else 0
```

Our Solution

- Borrow ideas from Domain Specific Languages (DSLs)
  - Translate low-level imperative program to high level functional program with preserved semantics
- Introduce first-class collective forms
  - The loop iteration index is an argument rather than a free variable

IMP:

```
int j = 0, sum = 0
while (j < k) {
    sum = sum + j
    j = j + 1
}
```

→

FUN:

```
let j = λ(i).i + 1


let sum = λ(i).if (i > 0)
                then sum(i-1)+j(i-1)
                else 0
let n = idx(i).!(j(i) <= k)
```

STORE:

```
j→if (n > 0) then j(n-1)
                 else 0
sum→if (n > 0) then sum(n-1)
                   else 0
```

Our Solution

- Borrow ideas from Domain Specific Languages (DSLs)
  - Translate low-level imperative program to high level functional program with preserved semantics
- Introduce first-class collective forms
  - The loop iteration index is an argument rather than a free variable

IMP:

```
int j = 0, sum = 0
while (j < k) {
    sum = sum + j        →
    j = j + 1
}
```

FUN:

```
let j = λ(i).i + 1


let sum = λ(i).if (i > 0)
                then sum(i-1)+i
                else 0
let n = idx(i).!(j(i) <= k)
```

STORE:

```
j→if (n > 0) then j(n-1)
                else 0
sum→if (n > 0) then sum(n-1)
                    else 0
```

Our Solution

- Borrow ideas from Domain Specific Languages (DSLs)
  - Translate low-level imperative program to high level functional program with preserved semantics
- Introduce first-class collective forms
  - The loop iteration index is an argument rather than a free variable

IMP:

```
int j = 0, sum = 0
while (j < k) {
    sum = sum + j
    j = j + 1
}
```

→

FUN:

```
let j = λ(i).i + 1

let sum = λ(i).if (i > 0)
               then sum(i-1)+i
               else 0

let n = idx(i).!(j(i) <= k)
```

STORE:

```
j→if (n > 0) then j(n-1)
                else 0
sum→if (n > 0) then sum(n-1)
                  else 0
```

Our Solution

- Borrow ideas from Domain Specific Languages (DSLs)
  - Translate low-level imperative program to high level functional program with preserved semantics
- Introduce first-class collective forms
  - The loop iteration index is an argument rather than a free variable

IMP:

```
int j = 0, sum = 0
while (j < k) {
    sum = sum + j
    j = j + 1
}
```

→

FUN:

```
let j = λ(i).i + 1

let sum = λ(i).Σ(i2 < i+1){ i2 }

let n = idx(i).!(j(i) <= k)
```

STORE:

```
j→if (n > 0) then j(n-1)
              else 0
sum→if (n > 0) then sum(n-1)
                else 0
```

Our Solution

- Borrow ideas from Domain Specific Languages (DSLs)
  - Translate low-level imperative program to high level functional program with preserved semantics
- Introduce first-class collective forms
  - The loop iteration index is an argument rather than a free variable

IMP:

```
int j = 0, sum = 0
while (j < k) {
    sum = sum + j      ⟶
    j = j + 1
}
```

FUN:

```
let j = λ(i).i + 1

let sum = λ(i).(i + 1) * i / 2

let n = idx(i).!(j(i) <= k)
```

STORE:

```
j→if (n > 0) then j(n-1)
                else 0
sum→if (n > 0) then sum(n-1)
                  else 0
```

Our Solution

- Borrow ideas from Domain Specific Languages (DSLs)
  - Translate low-level imperative program to high level functional program with preserved semantics
- Introduce first-class collective forms
  - The loop iteration index is an argument rather than a free variable

IMP:

```
int j = 0, sum = 0
while (j < k) {
    sum = sum + j
    j = j + 1
}
```

→

FUN:

```
let j = λ(i).i + 1

let sum = λ(i).(i + 1) * i / 2

let n = idx(i).!(j(i) <= k)
```

STORE:

```
j→if (n > 0) then j(n-1)
              else 0
sum→if (n > 0) then sum(n-1)
                else 0
```

Our Solution

- Borrow ideas from Domain Specific Languages (DSLs)
  - Translate low-level imperative program to high level functional program with preserved semantics
- Introduce first-class collective forms
  - The loop iteration index is an argument rather than a free variable

IMP:

```
int j = 0, sum = 0
while (j < k) {
    sum = sum + j
    j = j + 1
}
```

→

FUN:

```
let j = λ(i).i + 1

let sum = λ(i).(i + 1) * i / 2

let n = k
```

STORE:

```
j→if (n > 0) then j(n-1)
                  else 0
sum→if (n > 0) then sum(n-1)
                    else 0
```

Our Solution

- Borrow ideas from Domain Specific Languages (DSLs)
  - Translate low-level imperative program to high level functional program with preserved semantics
- Introduce first-class collective forms
  - The loop iteration index is an argument rather than a free variable

IMP:

```
int j = 0, sum = 0
while (j < k) {
    sum = sum + j
    j = j + 1
}
```

→

FUN:

```
let j = λ(i).i + 1

let sum = λ(i).(i + 1) * i / 2

let n = k
```

STORE:

```
j→if (n > 0) then j(n-1)
                  else 0
sum→if (n > 0) then sum(n-1)
                    else 0
```

Our Solution

- Borrow ideas from Domain Specific Languages (DSLs)
  - Translate low-level imperative program to high level functional program with preserved semantics
- Introduce first-class collective forms
  - The loop iteration index is an argument rather than a free variable

IMP:

```
int j = 0, sum = 0
while (j < k) {
    sum = sum + j
    j = j + 1
}
```

→

FUN:

```
let j = λ(i).i + 1

let sum = λ(i).(i + 1) * i / 2

let n = k
```

STORE:

```
j→k

sum→if (n > 0) then sum(n-1)
               else 0
```

Our Solution

- Borrow ideas from Domain Specific Languages (DSLs)
  - Translate low-level imperative program to high level functional program with preserved semantics
- Introduce first-class collective forms
  - The loop iteration index is an argument rather than a free variable

IMP:

```
int j = 0, sum = 0
while (j < k) {
    sum = sum + j
    j = j + 1
}
```

→

FUN:

```
let j = λ(i).i + 1

let sum = λ(i).(i + 1) * i / 2

let n = k
```

STORE:

```
j→k

sum→if (n > 0) then sum(n-1)
                else 0
```

Our Solution

- Borrow ideas from Domain Specific Languages (DSLs)
  - Translate low-level imperative program to high level functional program with preserved semantics
- Introduce first-class collective forms
  - The loop iteration index is an argument rather than a free variable

IMP:

```
int j = 0, sum = 0
while (j < k) {
    sum = sum + j
    j = j + 1
}
```

→

FUN:

```
let j = λ(i).i + 1

let sum = λ(i).(i + 1) * i / 2

let n = k
```

STORE:

```
j→k

sum→(k - 1) * k / 2
```

Collective Operations for Linked List

IMP:

```
ListNode x = null; int i = 0
while (i < n) {
    ListNode y = new ListNode()
    y.tail = x
    y.head = i
    x = y
    i = i + 1
}

ListNode z = x; int sum = 0
while (z != null) {
    sum = sum + z.head
    z = z.tail
}

assert(sum == n*(n-1)/2)
```

Collective Operations for Linked List

IMP:

```
ListNode x = null; int i = 0
while (i < n) {
    ListNode y = new ListNode()
    y.tail = x
    y.head = i
    x = y
    i = i + 1
}

ListNode z = x; int sum = 0
while (z != null) {
    sum = sum + z.head
    z = z.tail
}

assert(sum == n*(n-1)/2)
```

Collective Operations for Linked List

IMP:

```
ListNode x = null; int i = 0
while (i < n) {
    ListNode y = new ListNode()
    y.tail = x
    y.head = i
    x = y
    i = i + 1
}

ListNode z = x; int sum = 0
while (z != null) {
    sum = sum + z.head
    z = z.tail
}

assert(sum == n*(n-1)/2)
```

FUN:

```
let y = λ(i).if (i>0) then &new:ctx[i]
                      else &new:ctx[0]
let x = λ(i).if (i>0) then &new:ctx[i]
                      else &new:ctx[0]
let ctx = λ(i).
  newarray(i2 < i).
    [head -> i2,
     tail -> if (i2>0) then &new:ctx[i2-1]
                       else null]
```

Collective Operations for Linked List

IMP:

```
ListNode x = null; int i = 0
while (i < n) {
    ListNode y = new ListNode()
    y.tail = x
    y.head = i
    x = y
    i = i + 1
}

ListNode z = x; int sum = 0
while (z != null) {
    sum = sum + z.head
    z = z.tail
}

assert(sum == n*(n-1)/2)
```

→

FUN:

a location at context ctx
ctx = root.snd.snd.while[i].fst

```
let y = λ(i).if (i>0) then &new:ctx[i]
                      else &new:ctx[0]
let x = λ(i).if (i>0) then &new:ctx[i]
                      else &new:ctx[0]
let ctx = λ(i).
  newarray(i2 < i).
    [head -> i2,
     tail -> if (i2>0) then &new:ctx[i2-1]
                       else null]
```

Collective Operations for Linked List

IMP:

```
ListNode x = null; int i = 0
while (i < n) {
    ListNode y = new ListNode()
    y.tail = x
    y.head = i
    x = y
    i = i + 1
}

ListNode z = x; int sum = 0
while (z != null) {
    sum = sum + z.head
    z = z.tail
}

assert(sum == n*(n-1)/2)
```

FUN:

```
let y = λ(i).if (i>0) then &new:ctx[i]
                      else &new:ctx[0]
let x = λ(i).if (i>0) then &new:ctx[i]
                      else &new:ctx[0]
let ctx = λ(i).
  newarray(i2 < i).
    [head -> i2,
     tail -> if (i2>0) then &new:ctx[i2-1]
                       else null]
```

create an new array for 0 to i-1

Collective Operations for Linked List

IMP:

```
ListNode x = null; int i = 0
while (i < n) {
    ListNode y = new ListNode()
    y.tail = x
    y.head = i
    x = y
    i = i + 1
}

ListNode z = x; int sum = 0
while (z != null) {
    sum = sum + z.head
    z = z.tail
}

assert(sum == n*(n-1)/2)
```

→

FUN:

```
let y = λ(i).if (i>0) then &new:ctx[i]
                      else &new:ctx[0]
let x = λ(i).if (i>0) then &new:ctx[i]
                      else &new:ctx[0]
let ctx = λ(i).
  newarray(i2 < i).
    [head -> i2,
     tail -> if (i2>0) then &new:ctx[i2-1]
                       else null]
```

a record with two fields: head and tail

Collective Operations for Linked List

IMP:

```
ListNode x = null; int i = 0
while (i < n) {
    ListNode y = new ListNode()
    y.tail = x
    y.head = i
    x = y
    i = i + 1
}

ListNode z = x; int sum = 0
while (z != null) {
    sum = sum + z.head
    z = z.tail
}

assert(sum == n*(n-1)/2)
```

FUN:

```
let y = λ(i).if (i>0) then &new:ctx[i]
                       else &new:ctx[0]
let x = λ(i).if (i>0) then &new:ctx[i]
                       else &new:ctx[0]
let ctx = λ(i).
  newarray(i2 < i).
    [head -> i2,
     tail -> if (i2>0) then &new:ctx[i2-1]
                        else null]
```

Collective Operations for Linked List

IMP:

```
ListNode x = null; int i = 0
while (i < n) {
    ListNode y = new ListNode()
    y.tail = x
    y.head = i
    x = y
    i = i + 1
}

ListNode z = x; int sum = 0
while (z != null) {
    sum = sum + z.head
    z = z.tail
}

assert(sum == n*(n-1)/2)
```

FUN:

```
let y = λ(i).if (i>0) then &new:ctx[i]
                      else &new:ctx[0]
let x = λ(i).if (i>0) then &new:ctx[i]
                      else &new:ctx[0]
let ctx = λ(i).
  newarray(i2 < i).
    [head -> i2,
     tail -> if (i2>0) then &new:ctx[i2-1]
                       else null]

let sum = λ(i).if (i>0) then
                 sum(i-1) + σ[z(i-1)][head]
                 else σ[x(n-1)][head]
let z = λ(i).if (i>0) then σ[z(i-1)][tail]
                      else σ[x(n-1)][tail]
```

Collective Operations for Linked List

IMP:

```
ListNode x = null; int i = 0
while (i < n) {
    ListNode y = new ListNode()
    y.tail = x
    y.head = i
    x = y
    i = i + 1
}

ListNode z = x; int sum = 0
while (z != null) {
    sum = sum + z.head
    z = z.tail
}

assert(sum == n*(n-1)/2)
```

FUN:

```
let y = λ(i).if (i>0) then &new:ctx[i]
                      else &new:ctx[0]
let x = λ(i).if (i>0) then &new:ctx[i]
                      else &new:ctx[0]
let ctx = λ(i).
  newarray(i2 < i).
    [head -> i2,
     tail -> if (i2>0) then &new:ctx[i2-1]
                       else null]

let sum = λ(i).if (i>0) then
               sum(i-1) + σ[z(i-1)][head]
               else σ[x(n-1)][head]
let z = λ(i).if (i>0) then σ[z(i-1)][tail]
                      else σ[x(n-1)][tail]
```

Collective Operations for Linked List

IMP:

```
ListNode x = null; int i = 0
while (i < n) {
    ListNode y = new ListNode()
    y.tail = x
    y.head = i
    x = y
    i = i + 1
}

ListNode z = x; int sum = 0
while (z != null) {
    sum = sum + z.head
    z = z.tail
}

assert(sum == n*(n-1)/2)
```

FUN:

```
let y = λ(i).if (i>0) then &new:ctx[i]
                      else &new:ctx[0]
let x = λ(i).if (i>0) then &new:ctx[i]
                      else &new:ctx[0]
let ctx = λ(i).
  newarray(i2 < i).
    [head -> i2,
     tail -> if (i2>0) then &new:ctx[i2-1]
                       else null]

let sum = λ(i).(i+1) * i / 2

let z = λ(i).if (i>0) then &new:ctx[n-i-2]
                      else &new:ctx[n-2]
```

Formal Model: `IMP`

Statement Evaluation $\boxed{[\![\, s \,]\!](\sigma, c) = \sigma'}$

$$
\begin{array}{rcl}
[\![\, . \,]\!] & : & \mathrm{Stm} \to \mathrm{Sto} \times \mathrm{Ctx} \rightharpoonup \mathrm{Option\ Sto} \\
[\![\, x := \mathrm{new} \,]\!](\sigma, c) & = & \sigma[\&\mathrm{new}{:}c \mapsto [\,], \\
 & & \qquad \&x \mapsto [0 \mapsto \&\mathrm{new}{:}c]] \\
[\![\, e_1[e_2] := e_3 \,]\!](\sigma, c) & = & l \leftarrow [\![\, e_1 \,]\!](\sigma) \gg\!= \mathrm{toLoc} \\
 & & n \leftarrow [\![\, e_2 \,]\!](\sigma) \\
 & & v \leftarrow [\![\, e_3 \,]\!](\sigma) \\
 & & o \leftarrow \sigma[l] \\
 & & \sigma[l \mapsto o[n \mapsto v]] \\
[\![\, \mathrm{if}\ (e)\ s_1\ \mathrm{else}\ s_2 \,]\!](\sigma, c) & = & b \leftarrow [\![\, e \,]\!](\sigma) \gg\!= \mathrm{toBool} \\
 & & \mathrm{if}\ b\ \mathrm{then}\ [\![\, s_1 \,]\!](\sigma, c.\mathrm{then}) \\
 & & \mathrm{else}\ [\![\, s_2 \,]\!](\sigma, c.\mathrm{else}) \\
[\![\, \mathrm{while}\ e\ \mathrm{do}\ s \,]\!](\sigma, c) & = & [\![\, e\ s \,]\!](\sigma, c)(n)\ \mathrm{where} \\
 & & n = \#(\lambda i.(\sigma' \leftarrow [\![\, e\ s \,]\!](\sigma, c)(i) \\
 & & \qquad\quad b \ \ \leftarrow [\![\, e \,]\!](\sigma') \gg\!= \mathrm{toBool} \\
 & & \qquad\quad \mathrm{Some}\ \neg b)\ \mathrm{getOrElse\ true}) \\
[\![\, s_1;\ s_2 \,]\!](\sigma, c) & = & \sigma' \leftarrow [\![\, s_1 \,]\!](\sigma, c.\mathrm{fst}) \\
 & & [\![\, s_2 \,]\!](\sigma', c.\mathrm{snd}) \\
[\![\, \mathrm{skip} \,]\!](\sigma, c) & = & \mathrm{Some}\ \sigma \\
[\![\, \mathrm{abort} \,]\!](\sigma, c) & = & \mathrm{None}
\end{array}
$$

- Functional semantics in monadic style defines its meaning
- [[.]] : Stm -> Sto x Ctx -> Opt Sto

Formal Model: FUN

Expressions $\boxed{g \in \text{Fxp}}$

$g ::=$	
$n \mid b \mid l \mid []$	Constant (nat, bool, ctx, obj)
x	Variable
$e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2$	Arithmetic
$e_1 < e_2 \mid e_1 = e_2 \mid e_1 \wedge e_2 \mid \neg e$	Boolean
$e_1 \ni e_2$	Field exists?
$e_1[e_2]$	Field read
$e_1[e_2 \mapsto e_3]$	Field update
$\text{if } e \text{ then } s_1 \text{ else } s_2$	Conditional
$\text{letrec } x_1 = g_1, \ \ldots \ \text{in } g_n$	Recursive let
$g_1(g_2)$	Function application
$\lambda(x).\ g$	Function
$\#(x).\ g$	First index
$\Sigma(x < g_1).\ g_2$	Sum
$\Pi(x < g_1).\ g_2$	Product
$\forall\ (x < g_1).\ g_2$	Conjunction
$\langle . \rangle(x < g_1).\ g_2$	Sequence
$w ::=$	Value
$n \mid b \mid l$	Constant
$[n_0 \mapsto w_0, \ \ldots]$	Object

- FUN: λ calculus + records + collective operations
- store is represented by a record

Formal Model: IMP→FUN

Statement Evaluation

$$\boxed{[\![\ s\ ]\!](\sigma, c) = \sigma'}$$

$$
\begin{array}{rcl}
[\![\ .\ ]\!] & : & \text{Stm} \to \text{Sto} \times \text{Ctx} \rightharpoonup \text{Option Sto} \\
[\![\ x := \text{new}\ ]\!](\sigma, c) & = & \sigma[\&\text{new}{:}c \mapsto [\,], \\
 & & \quad \&x \mapsto [0 \mapsto \&\text{new}{:}c]] \\
[\![\ e_1[e_2] := e_3\ ]\!](\sigma, c) & = & l \leftarrow [\![\ e_1\ ]\!](\sigma) \gg= \text{toLoc} \\
 & & n \leftarrow [\![\ e_2\ ]\!](\sigma) \\
 & & v \leftarrow [\![\ e_3\ ]\!](\sigma) \\
 & & o \leftarrow \sigma[l] \\
 & & \sigma[l \mapsto o[n \mapsto v]] \\
[\![\ \text{if}\ (e)\ s_1\ \text{else}\ s_2\ ]\!](\sigma, c) & = & b \leftarrow [\![\ e\ ]\!](\sigma) \gg= \text{toBool} \\
 & & \text{if}\ b\ \text{then}\ [\![\ s_1\ ]\!](\sigma, c.\text{then}) \\
 & & \text{else}\ [\![\ s_2\ ]\!](\sigma, c.\text{else}) \\
[\![\ \text{while}\ e\ \text{do}\ s\ ]\!](\sigma, c) & = & [\![\ e\ s\ ]\!](\sigma, c)(n)\ \text{where} \\
 & & n = \#(\lambda i.(\sigma' \leftarrow [\![\ e\ s\ ]\!](\sigma, c)(i) \\
 & & \quad b \leftarrow [\![\ e\ ]\!](\sigma') \gg= \text{toBool} \\
 & & \quad \text{Some}\ \neg b)\ \text{getOrElse true}) \\
[\![\ s_1;\ s_2\ ]\!](\sigma, c) & = & \sigma' \leftarrow [\![\ s_1\ ]\!](\sigma, c.\text{fst}) \\
 & & [\![\ s_2\ ]\!](\sigma', c.\text{snd}) \\
[\![\ \text{skip}\ ]\!](\sigma, c) & = & \text{Some}\ \sigma \\
[\![\ \text{abort}\ ]\!](\sigma, c) & = & \text{None}
\end{array}
$$

Translation

$$
\begin{array}{rcl}
\text{None} & = & [\text{valid} \mapsto \text{false}] \\
\text{Some}\ g & = & [\text{valid} \mapsto \text{true}, \text{data} \mapsto g] \\
g \gg= f & = & \text{if}\ g.\text{valid then}\ f(g.\text{data})\ \text{else None} \\
g_1\ \text{getOrElse}\ g_2 & = & \text{if}\ g_1.\text{valid then}\ g_1.\text{data else}\ g_2 \\
\text{Val}\ n & = & [\text{tpe} \mapsto \text{nat}, \text{val} \mapsto n] \\
\text{Val}\ b & = & [\text{tpe} \mapsto \text{bool}, \text{val} \mapsto b] \\
\text{Val}\ l & = & [\text{tpe} \mapsto \text{loc}, \text{val} \mapsto l] \\
\text{toNat}\ g & = & \text{if}\ g.\text{tpe} = \textit{nat}\ \text{then Some}\ g.\text{val else None} \\
\text{toBool}\ g & = & \text{if}\ g.\text{tpe} = \textit{bool}\ \text{then Some}\ g.\text{val else None} \\
\text{toLoc}\ g & = & \text{if}\ g.\text{tpe} = \textit{loc}\ \text{then Some}\ g.\text{val else None} \\
o[n] & = & \text{if}\ o \ni n\ \text{then}\ [\text{valid} \mapsto \text{true}, \text{data} \mapsto o[n]] \\
 & & \text{else}\ [\text{valid} \mapsto \text{false}] \\
\sigma[l] & = & \text{if}\ \sigma \ni l\ \text{then}\ [\text{valid} \mapsto \text{true}, \text{data} \mapsto \sigma[l]] \\
 & & \text{else}\ [\text{valid} \mapsto \text{false}]
\end{array}
$$

Formal Model: Simplification on FUN

- Rule-based rewriting to obtain collective forms

Collective Forms

$$\begin{aligned}
&\text{letrec } f = (\lambda(i).\ \text{if } 0 < i \text{ then} \\
&f(i-1)[i \mapsto g_i] \text{ else } [\,]) \text{ in } f(a) \quad \equiv \quad \langle . \rangle(i < a + 1).\ g_i
\end{aligned}$$

$$\begin{aligned}
&\text{letrec } f = (\lambda(i).\ \text{if } 0 < i \text{ then} \\
&f(i-1) + g_i \text{ else } 0) \text{ in } f(a) \quad \equiv \quad \Sigma(i < a + 1).\ g_i
\end{aligned}$$

$$\Sigma(i < n).\ i \quad \equiv \quad n/2 * (n-1)$$

$$\#(i).\ \neg(i < a) \quad \equiv \quad a$$

$$\ldots$$

Formal Model: Simplification on FUN

- Rule-based rewriting to obtain collective forms

Collective Forms

$$\text{letrec } f = (\lambda(i).\ \text{if } 0 < i \text{ then } f(i-1)[i \mapsto g_i] \text{ else } []) \text{ in } f(a) \equiv \langle . \rangle (i < a+1).\ g_i$$

$$\text{letrec } f = (\lambda(i).\ \text{if } 0 < i \text{ then } f(i-1) + g_i \text{ else } 0) \text{ in } f(a) \equiv \Sigma(i < a+1).\ g_i$$

```
let j = λ(i).if (i > 0) then j(i-1)+1 else 0

let j = λ(i).Σ(i2 < i + 1){ 1 }
```

$$\Sigma(i < n).\ i \equiv n/2 * (n-1)$$

$$\#(i).\ \neg(i < a) \equiv a$$

$$\ldots$$

Evaluation

- We implement a prototypical tool `SIGMA` for a subset of C
- Evaluate against with CPAChecker and SeaHorn on SV-COMP benchmarks with loops

Evaluation

- We implement a prototypical tool `SIGMA` for a subset of C
- Evaluate against with CPAChecker and SeaHorn on SV-COMP benchmarks with loops

Name	CPAchecker	SeaHorn	SIGMA
simple_built_from_end_true-unreach-call.i	TIMEOUT	250	273
list_addnat_false-unreach-call.i	2890	190	215
list_addnat_true-unreach-call.i	305560	170	215
loop_addnat_false-unreach-call.i	2830	190	285
loop_addnat_true-unreach-call.i	TIMEOUT	200	285
loop_addsubnat_false-unreach-call.i	3140	210	364
loop_addsubnat_true-unreach-call.i	TIMEOUT	230	364
nestedloop_mul1_true-unreach-call.i	OUT OF MEMORY	7280	405
nestedloop_mul2_true-unreach-call.i	TIMEOUT	240	365

Evaluation

- We implement a prototypical tool SIGMA for a subset of C
- Evaluate against with CPAChecker and SeaHorn on SV-COMP benchmarks with loops

cells in red are incorrect results

Name	CPAchecker	SeaHorn	SIGMA
simple_built_from_end_true-unreach-call.i	TIMEOUT	250	273
list_addnat_false-unreach-call.i	2890	190	215
list_addnat_true-unreach-call.i	305560	170	215
loop_addnat_false-unreach-call.i	2830	190	285
loop_addnat_true-unreach-call.i	TIMEOUT	200	285
loop_addsubnat_false-unreach-call.i	3140	210	364
loop_addsubnat_true-unreach-call.i	TIMEOUT	230	364
nestedloop_mul1_true-unreach-call.i	OUT OF MEMORY	7280	405
nestedloop_mul2_true-unreach-call.i	TIMEOUT	240	365

Summary

- To verify program with loops, we translate low-level code to high-level DSL with collective forms
- The semantics and errors are preserved during translation
- Analyses are done through rewriting and simplification
- Heap abstraction also uses collective forms to reflect program structure
- We have scaled up our approach to a subset of C and use it to successfully verify programs from SV-COMP benchmarks

Thanks!

Questions/comments?