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

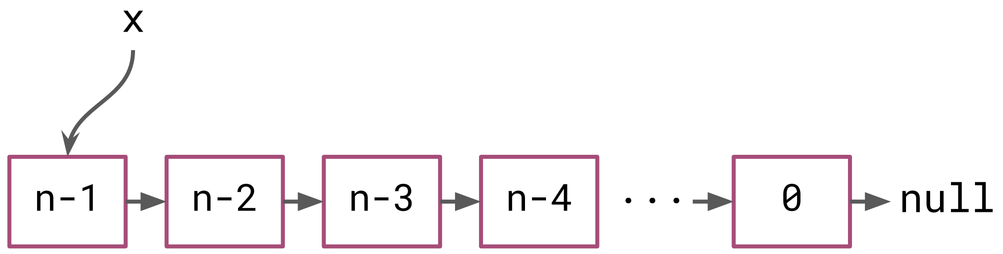
Gregory Essertel, **Guannan Wei**, Tiark Rompf
Department of Computer Science, Purdue University
December 1, 2017, MWPLS

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)
```

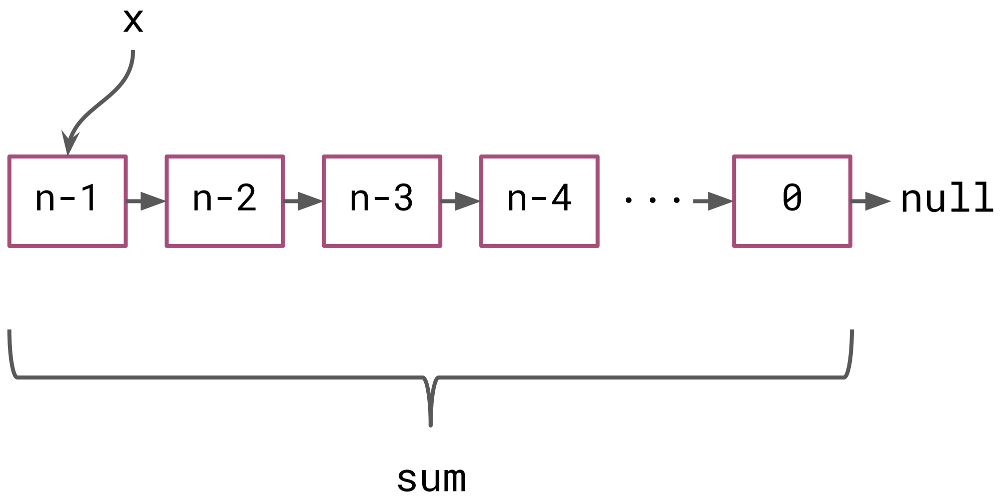
Motivation

```
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)
```



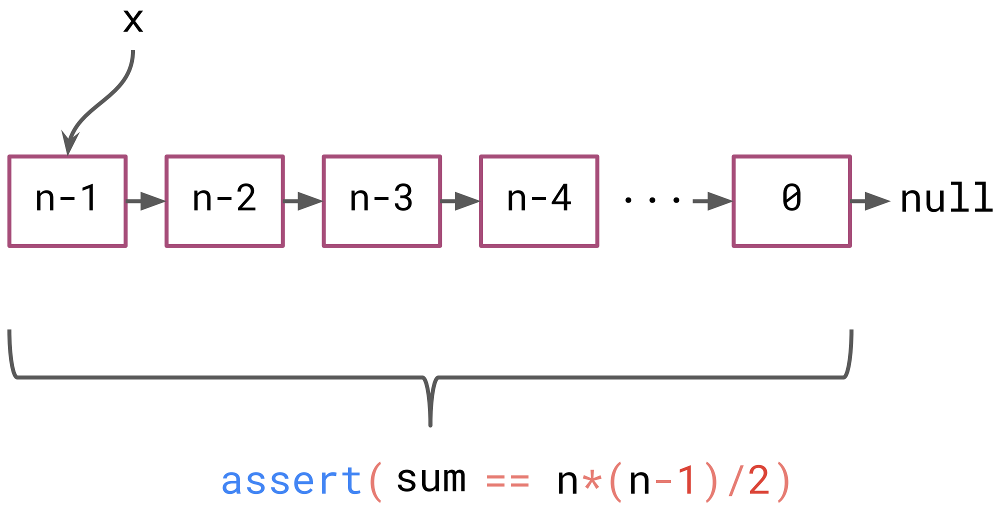
Motivation

```
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 are not distinguished.

Our Solution

- Borrow ideas from Domain Specific Languages (DSL)
  - Translate low-level imperative program to high level functional program with semantics preserved
- First-class collective forms
  - The loop iteration index is not a free variable

Our Solution

- Borrow ideas from Domain Specific Languages (DSL)
  - Translate low-level imperative program to high level functional program with semantics preserved
- First-class collective forms
  - The loop iteration index is not 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 (DSL)
  - Translate low-level imperative program to high level functional program with semantics preserved
- First-class collective forms
  - The loop iteration index is not a free variable

```
IMP:                          FUN:

int j = 0, sum = 0            let j = λ(i).if (i > 0)
while (j < k) {                           then j(i-1)+1
    sum = sum + j   ──→                   else 1
    j = j + 1                 let sum = λ(i).if (i > 0)
}                                           then sum(i-1)+j(i-1)
                                            else 0
                              let n = #(i).!(j(i) <= k)
```

Our Solution

- Borrow ideas from Domain Specific Languages (DSL)
  - Translate low-level imperative program to high level functional program with semantics preserved
- First-class collective forms
  - The loop iteration index is not 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 1
let sum = λ(i).if (i > 0)
               then sum(i-1)+j(i-1)
               else 0
let n = #(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 (DSL)
  - Translate low-level imperative program to high level functional program with semantics preserved
- First-class collective forms
  - The loop iteration index is not 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 1
let sum = λ(i).if (i > 0)
               then sum(i-1)+j(i-1)
               else 0
let n = #(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 (DSL)
  - Translate low-level imperative program to high level functional program with semantics preserved
- First-class collective forms
  - The loop iteration index is not 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 = #(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 (DSL)
    - Translate low-level imperative program to high level functional program with semantics preserved
- First-class collective forms
    - The loop iteration index is not 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 = #(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 (DSL)
  - Translate low-level imperative program to high level functional program with semantics preserved
- First-class collective forms
  - The loop iteration index is not a free variable

IMP:

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

$\longrightarrow$

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 = #(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 (DSL)
  - Translate low-level imperative program to high level functional program with semantics preserved
- First-class collective forms
  - The loop iteration index is not 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 = #(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 (DSL)
  - Translate low-level imperative program to high level functional program with semantics preserved
- First-class collective forms
  - The loop iteration index is not a free variable

IMP:

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

$\longrightarrow$

FUN:

```
let j = λ(i).i + 1

let sum = λ(i).if (i > 0)
               then sum(i-1)+i
               else 0
let n = #(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 (DSL)
  - Translate low-level imperative program to high level functional program with semantics preserved
- First-class collective forms
  - The loop iteration index is not 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 = #(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 (DSL)
  - Translate low-level imperative program to high level functional program with semantics preserved
- First-class collective forms
  - The loop iteration index is not 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 = #(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 (DSL)
    - Translate low-level imperative program to high level functional program with semantics preserved
- First-class collective forms
    - The loop iteration index is not a free variable

```
IMP:

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

$\longrightarrow$

```
FUN:

let j = λ(i).i + 1

let sum = λ(i).(i + 1) * i / 2

let n = #(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 (DSL)
  - Translate low-level imperative program to high level functional program with semantics preserved
- First-class collective forms
  - The loop iteration index is not a free variable

IMP:

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

$\longrightarrow$

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 (DSL)
    - Translate low-level imperative program to high level functional program with semantics preserved
- First-class collective forms
    - The loop iteration index is not a free variable

IMP:

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

$\longrightarrow$

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 (DSL)
  - Translate low-level imperative program to high level functional program with semantics preserved
- First-class collective forms
  - The loop iteration index is not 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

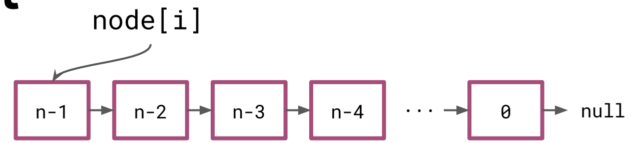
```
IMP:

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)
```

FUN:



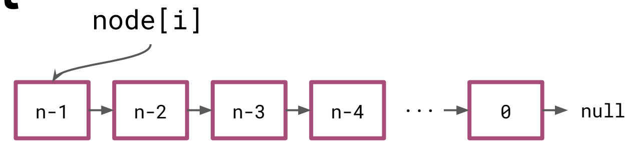
```
let y = λ(i).if (i>0) then &new:node[i]
                      else &new:node[0]
let x = λ(i).if (i>0) then &new:node[i]
                      else &new:node[0]
let node = λ(i).
  newArray(i2<i).
    [tail -> if (i2>0) then &new:node[i2-1]
                       else null,
     head -> i2]
```

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:



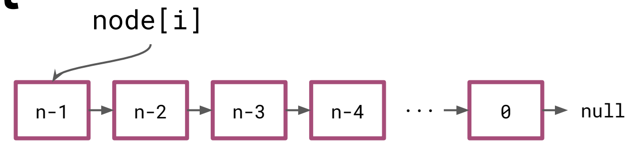
```
let y = λ(i).if (i>0) then &new:node[i]
                      else &new:node[0]
let x = λ(i).if (i>0) then &new:node[i]
                      else &new:node[0]
let node = λ(i).
  newArray(i2<i).
    [tail -> if (i2>0) then &new:node[i2-1]
                       else null,
     head -> i2]

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

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:



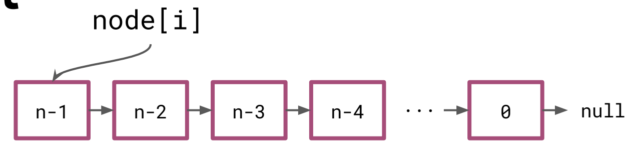
```
let y = λ(i).if (i>0) then &new:node[i]
                      else &new:node[0]
let x = λ(i).if (i>0) then &new:node[i]
                      else &new:node[0]
let node = λ(i).
  newArray(i2<i).
    [tail -> if (i2>0) then &new:node[i2-1]
                       else null,
     head -> i2]

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

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

let z = λ(i).if (i>0) then &new:node[n-i-2]
                      else &new:node[n-2]
let sum = λ(i).(i+1) * i / 2
```

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
- Heap abstraction also use 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!