Garter
Garbage Collection
Garter / GC

Example 1
let x = (1, 2), y = let tmp = (10, 20)
    in tmp[0] + tmp[1], p0 = x[0] + y, p1 = x[1] + y
in
(p0, p1)
let x = (1, 2), y = let tmp = (10, 20) in tmp[0] + tmp[1], p0 = x[0] + y, p1 = x[1] + y in (p0, p1)
let x = (1, 2)
, y = let tmp = (10, 20)
in tmp[0] + tmp[1]
, p0 = x[0] + y
, p1 = x[1] + y
in (p0, p1)
let x = (1, 2), y = let tmp = (10, 20) in tmp[0] + tmp[1], p0 = x[0] + y, p1 = x[1] + y in (p0, p1)
let x = (1, 2)
, y = let tmp = (10, 20)
   in tmp[0] + tmp[1]
, p0 = x[0] + y
, p1 = x[1] + y
in
(p0, p1)
let x = (1, 2), y = let tmp = (10, 20)
in tmp[0] + tmp[1], p0 = x[0] + y, p1 = x[1] + y
in (p0, p1)
let x = (1, 2), y = let tmp = (10, 20) in tmp[0] + tmp[1], p0 = x[0] + y, p1 = x[1] + y in (p0, p1)

Result (rax) = 0x21
let \( x = (1, 2) \)
, \( y = \) let \( \text{tmp} = (10, 20) \)
in \( \text{tmp}[0] + \text{tmp}[1] \)
, \( p0 = x[0] + y \)
, \( p1 = x[1] + y \)in
\((p0, p1)\)

Suppose we had a smaller, 4-word heap
let x = (1, 2), y = let tmp = (10, 20) in tmp[0] + tmp[1], p0 = x[0] + y, p1 = x[1] + y in (p0, p1)
let x = (1, 2), y = let tmp = (10, 20) in tmp[0] + tmp[1], p0 = x[0] + y, p1 = x[1] + y in (p0, p1)

Out of memory!
Can't allocate (p0, p1)

1 2 10 20
0x00 0x08 0x10 0x18 0x20

rbp
x
y
p0
p1
0x01
r15
let x = (1, 2), y = let tmp = (10, 20) in tmp[0] + tmp[1], p0 = x[0] + y, p1 = x[1] + y in (p0, p1)

(10, 20) is “garbage”

Q: How to determine if cell is garbage?
let x = (1, 2)
, y = let tmp = (10, 20)
    in tmp[0] + tmp[1]
, p0 = x[0] + y
, p1 = x[1] + y
in (p0, p1)

(10, 20) is “garbage”
let x = (1, 2), y = let tmp = (10, 20) in tmp[0] + tmp[1], p0 = x[0] + y, p1 = x[1] + y in (p0, p1)
let \( x = (1, 2) \), \( y = \) let \( \text{tmp} = (10, 20) \) in \( \text{tmp}[0] + \text{tmp}[1] \), \( p0 = x[0] + y \), \( p1 = x[1] + y \) in \( (p0, p1) \)

**Result** (rax) = 0x11
Garter / GC
Example 2
let y = let tmp = (10, 20) in tmp[0] + tmp[1], x = (1, 2), p0 = x[0] + y, p1 = x[1] + y in (p0, p1)

Start with a 4-word heap
let y = let tmp = (10, 20)
in tmp[0] + tmp[1], x = (1, 2), p0 = x[0] + y, p1 = x[1] + y
in (p0, p1)
let y = let tmp = (10, 20)
    in tmp[0] + tmp[1]
, x = (1, 2)
, p0 = x[0] + y
, p1 = x[1] + y
in
(p0, p1)
let y = let tmp = (10, 20)
in tmp[0] + tmp[1]
x = (1, 2)
, p0 = x[0] + y
, p1 = x[1] + y
in
(p0, p1)
let y = let tmp = (10, 20)
in tmp[0] + tmp[1]
, x = (1, 2)
, p0 = x[0] + y
, p1 = x[1] + y
in (p0, p1)
let y = let tmp = (10, 20) in tmp[0] + tmp[1], x = (1, 2), p0 = x[0] + y, p1 = x[1] + y in (p0, p1)
let y = let tmp = (10, 20) in tmp[0] + tmp[1], x = (1, 2), p0 = x[0] + y, p1 = x[1] + y in (p0, p1)

Out of memory!
Can’t allocate (p0, p1)
let y = let tmp = (10, 20)
in tmp[0] + tmp[1]
, x = (1, 2)
, p0 = x[0] + y
, p1 = x[1] + y
in (p0, p1)

ex2: garbage in the middle

Lets reclaim & recycle garbage!

```
go
let y = (10, 20)
in y[0] + y[1]
, x = (1, 2)
, p0 = x[0] + y
, p1 = x[1] + y
in (p0, p1)
```
ex2: garbage in the middle

```
let y = let tmp = (10, 20)
  in tmp[0] + tmp[1]
, x = (1, 2)
, p0 = x[0] + y
, p1 = x[1] + y
in (p0, p1)
```

 Lets reclaim & recycle garbage!

QUIZ: Which cells are garbage?

(A) 0x00, 0x08  (B) 0x08, 0x10  (C) 0x18, 0x20  (D) None  (E) All
let y = let tmp = (10, 20)
    in tmp[0] + tmp[1]
, x = (1, 2)
, p0 = x[0] + y
, p1 = x[1] + y
in (p0, p1)

ex2: garbage in the middle

Let's reclaim & recycle garbage!

QUIZ: Which cells are garbage?
Those that are not reachable from stack
let y = let tmp = (10, 20) in tmp[0] + tmp[1]
\[ y = 30 \]

, x = (1, 2)
\[ x = \begin{bmatrix} 31 \\ 1 \end{bmatrix} \]

, p0 = x[0] + y
\[ p0 = 32 \]

, p1 = x[1] + y
\[ p1 = 32 \]

in (p0, p1)

ex2: garbage in the middle

Let's reclaim & recycle garbage!

QUIZ: Which cells are garbage?

Those that are not reachable from stack
ex2: garbage in the middle

```
let y = let tmp = (10, 20)
    in tmp[0] + tmp[1]
, x = (1, 2)
, p0 = x[0] + y
, p1 = x[1] + y
in (p0, p1)
```

Q: How to reclaim space?
Why is it not enough to rewind r15?
let y = let tmp = (10, 20)
    in tmp[0] + tmp[1]
, x = (1, 2)
, p0 = x[0] + y
, p1 = x[1] + y
in (p0, p1)

ex2: garbage in the middle

Lets reclaim & recycle garbage!

Why is it not enough to rewind r15?
Want free space to be contiguous (i.e. go to end of heap)
ex2: garbage in the middle

```
let y = let tmp = (10, 20) in tmp[0] + tmp[1], x = (1, 2), p0 = x[0] + y, p1 = x[1] + y in (p0, p1)
```

Solution: Compaction

Copy “live” cells into “garbage” ...
ex2: garbage in the middle

```
let y = let tmp = (10, 20)
    in tmp[0] + tmp[1]
  , x = (1, 2)
  , p0 = x[0] + y
  , p1 = x[1] + y
in (p0, p1)
```

Solution: Compaction

Copy “live” cells into “garbage” ...
ex2: garbage in the middle

```plaintext
let y = let tmp = (10, 20)
    in tmp[0] + tmp[1]
, x = (1, 2)
, p0 = x[0] + y
, p1 = x[1] + y
in (p0, p1)
```

Let's reclaim & recycle garbage!

**Solution: Compaction**

Copy “live” cells into “garbage” ...
let y = let tmp = (10, 20) in tmp[0] + tmp[1], x = (1, 2), p0 = x[0] + y, p1 = x[1] + y in (p0, p1)

Solution: Compaction

Copy “live” cells into “garbage” …
let y = let tmp = (10, 20)
    in tmp[0] + tmp[1]
, x = (1, 2)
, p0 = x[0] + y
, p1 = x[1] + y
in (p0, p1)

let y = let tmp = (10, 20)
    in tmp[0] + tmp[1]
, x = (1, 2)
, p0 = x[0] + y
, p1 = x[1] + y
in (p0, p1)

Let's reclaim & recycle garbage!

Solution: Compaction

Copy "live" cells into "garbage" … and then … rewind r15!
let y = let tmp = (10, 20)
    in tmp[0] + tmp[1]
, x = (1, 2)
, p0 = x[0] + y
, p1 = x[1] + y
in (p0, p1)

Yay! Have space for (p0, p1)
let y = let tmp = (10, 20) in tmp[0] + tmp[1], x = (1, 2), p0 = x[0] + y, p1 = x[1] + y in (p0, p1)

Yay! Have space for \((p0, p1)\)
let y = let tmp = (10, 20)
  in tmp[0] + tmp[1]
, x = (1, 2)
, p0 = x[0] + y
, p1 = x[1] + y
in
(p0, p1)

Result (rax) = 0x09
Garter / GC
Example 3
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)
in
x[0] + y + z
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
    , x = (y, y + 1)
    , z = foo(100, 200)

in
    x[0] + y + z
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)

in
    x[0] + y + z
def foo(p, q):
   let tmp = (p, q)
   in tmp[0] + tmp[1]

let y = foo(10, 20)
   , x = (y, y + 1)
   , z = foo(100, 200)
in
   x[0] + y + z
def foo(p, q):
    let tmp = (p, q)
in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)
in x[0] + y + z
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y  = foo(10, 20)
, x  = (y, y + 1)
, z  = foo(100, 200)
in
    x[0] + y + z
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)
in
  x[0] + y + z

Return (rax) = 30
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)
in
x[0] + z

Return (rax) = 30
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y  = foo(10, 20)
 , x  = (y, y + 1)
 , z  = foo(100, 200)
 in
 x[0] + z
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
    , x = (y, y + 1)
    , z = foo(100, 200)
in
    x[0] + z
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
    , x = (y, y + 1)
    , z = foo(100, 200)
in
    x[0] + z
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)

in
    x[0] + z
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)
in
x[0] + z
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)
in
x[0] + z
```python
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)
in
x[0] + z
```
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)

in
  x[0] + z

Let's reclaim & recycle garbage!
ex3: garbage in the middle (with stack)

```python
def foo(p, q):
    let tmp = (p, q)
in tmp[0] + tmp[1]

let y = foo(10, 20), x = (y, y + 1), z = foo(100, 200)
in x[0] + z
```

Lets reclaim & recycle garbage!

QUIZ: Which cells are garbage?

(A) 0x00, 0x08  (B) 0x08, 0x10  (C) 0x10, 0x18  (D) None  (E) All
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)
in
    x[0] + z

Let's reclaim & recycle garbage!

QUIZ: Which cells are garbage?
Those that are not reachable from any stack frame
ex3: garbage in the middle (with stack)

```python
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)
in x[0] + z
```

Let's reclaim & recycle garbage!

**QUIZ: Which cells are garbage?**
Those that are not reachable from any stack frame
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)
in
x[0] + z

Lets reclaim & recycle garbage!

Which cells are garbage?
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)
in
x[0] + z

Compact the live cells
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)

in
x[0] + z

Compact the live cells
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)

in
x[0] + z

Compact the live cells
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)

in x[0] + z

Compact the live cells … then rewind r15
def foo(p, q):
    let tmp = (p, q)
in tmp[0] + tmp[1]

let y = foo(10, 20), x = (y, y + 1), z = foo(100, 200)
in x[0] + z

Compact the live cells ... then rewind r15
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)
in
x[0] + z

Problem???
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)

in
    x[0] + z

Problem! Have to REDIRECT existing pointers
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)

in
    x[0] + z
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)

in
    x[0] + z

1. Compute FORWARD addrs
   (i.e. new compacted addrs)
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)
in
    x[0] + z

1. Compute \text{FORWARD} \text{ addr}s
e.g. 0x11 \rightarrow 0x01
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)
in
    x[0] + z

1. Compute **FORWARD** addrs
e.g. 0x11 —> 0x01

2. **REDIRECT** addrs on stack
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)
in
x[0] + z

1. Compute FORWARD addrs
e.g. 0x11 —> 0x01

2. REDIRECT addrs on stack

3. COMPACT cells on heap
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)
in
    x[0] + z

1. Compute FORWARD addrs
e.g. 0x11 —> 0x01

2. REDIRECT addrs on stack

3. COMPACT cells on heap
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)

in
    x[0] + z

Yay! Have space for (p, q)
ex3: garbage in the middle (with stack)

```python
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)
in
x[0] + z

Yay! Have space for (p, q)
```

```plaintext
30 31 100 200
0x00 0x08 0x10 0x18 0x20
r15
```
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)

in
x[0] + z

Return (rax) = 300
ex3: garbage in the middle (with stack)

```python
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
    , x = (y, y + 1)
    , z = foo(100, 200)

in x[0] + z
```

Return \( \text{rax} = 300 \)
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
, x = (y, y + 1)
, z = foo(100, 200)
in
x[0] + z
def foo(p, q):
    let tmp = (p, q)
    in tmp[0] + tmp[1]

let y = foo(10, 20)
    , x = (y, y + 1)
    , z = foo(100, 200)

in
    x[0] + z

Return (rax) = 30+300 = 330
Garter / GC
Example 4
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in
(1000, l)
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l  = range(t1, t1 + 3)
in
(1000, l)

call range(0, 3)
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in
(1000, l)

QUIZ: What is heap when range(0, 3) returns?

(A)  

(B)
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in (1000, l)
```python
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in (1000, l)
```

Result \(\text{sum}(0x11) = 3\)
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in
(1000, l)

ex4: recursive data

r15

<table>
<thead>
<tr>
<th>2</th>
<th>false</th>
<th>1</th>
<th>0x01</th>
<th>0</th>
<th>0x11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x08</td>
<td>0x10</td>
<td>0x18</td>
<td>0x20</td>
<td>0x28</td>
</tr>
</tbody>
</table>
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in
(1000, l)

call range(3,6)
ex4: recursive data

```python
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in (1000, l)
```

**QUIZ: What is the value of l?**

(A) 0x30 (B) 0x31 (C) 0x50 (D) 0x51 (E) 0x60
def range(i, j):
    if (j <= i): false else: (i,range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in
(1000, l)

Yikes! Out of Memory!
QUIZ: Which cells are “live” on the heap?

(A) 0x00
(B) 0x10
(C) 0x20
(D) 0x30
(E) 0x40
(F) 0x50
def range(i, j):
    if (j <= i): false else: (i,range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in
(1000, l)

1. **MARK** live addrs
2. Compute **FORWARD** addrs
3. **REDIRECT** addrs on stack
4. **COMPACT** cells on heap
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in (1000, l)

1. MARK live addr
reachable from stack
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l  = range(t1, t1 + 3)
in (1000, l)

1. **MARK** live addr

reachable from stack
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == False: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in (1000, l)
def range(i, j):
    if (j <= i): false else: (i,range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l  = range(t1, t1 + 3)
in (1000, l)

ex4: recursive data

1. MARK live addrs
   reachable from stack
def range(i, j):
    if (j <= i): false else: (i,range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l  = range(t1, t1 + 3)
in
(1000, l)

1. MARK live addr
reachable from stack
def range(i, j):
    if (j <= i): false else: (i,range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l  = range(t1, t1 + 3)
in (1000, l)
ex4: recursive data

def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in
(1000, l)

Done!
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)

in (1000, l)

2. Compute FORWARD addr

ex4: recursive data
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in (1000, l)

2. Compute FORWARD addr
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in (1000, l)

2. Compute FORWARD addrs
def range(i, j):
    if (j <= i): false else: (i,range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in
(1000, l)
ex4: recursive data

```python
def range(i, j):
    if (j <= i): false else: (i,range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in
(1000, l)
```

2. Compute FORWARD addr
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in
(1000, l)
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in (1000, l)

2. Compute FORWARD addr
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in (1000, l)

ex4: recursive data

2. Compute FORWARD addr
2. Compute **FORWARD** addrs
def range(i, j):
    if (j <= i): false else: (i,range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l  = range(t1, t1 + 3)
in (1000, l)

2. Compute FORwARd addrs
```python
def range(i, j):
    if (j <= i):
        False
    else:
        (i, range(i+1, j))

def sum(l):
    if l == False:
        0
    else:
        l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in
    (1000, l)
```

2. Compute **FORWARD** addrs
Where should we store the forward addrs?
def range(i, j):
    if (j <= i): false else: (i,range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in
(1000, l)

3. REDIRECT addr on stack
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in (1000, l)

3. REDIRECT addr on stack
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in
(1000, l)

3. REDIRECT addr on stack
ex4: recursive data

```python
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in
(1000, l)
```

3. REDIRECT addrs on stack and heap!
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in (1000, l)

3. REDIRECT addrs on stack and heap!
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
        in sum(l1)
    , l = range(t1, t1 + 3)
in (1000, l)

3. REDIRECT addrs on stack and heap!
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in (1000, l)

3. REDIRECT addrs on stack and heap!
def range(i, j):
    if (j <= i): false else: (i,range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in (1000, l)

3. REDIRECT addrs on stack and heap!
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in
(1000, l)

4. **COMPACT** cells on heap
Copy cell to forward addr!
def range(i, j):
    if (j <= i): false else: (i,range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)

(1000, l)

4. **COMPACT** cells on heap
Copy cell to forward addr!
def range(i, j):
    if (j <= i): false else: (i,range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in
(1000, l)

4. COMPACT cells on heap
Copy cell to forward addr!
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l  = range(t1, t1 + 3)
in (1000, l)

4. **COMPACT** cells on heap

Copy cell to forward addr!
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in
(1000, l)

4. COMPACT cells on heap
Copy cell to forward addr!
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in
(1000, l)

4. COMPACT cells on heap
Copy cell to forward addr!

ex4: recursive data
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)

(1000, l)
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
    in sum(l1)
, l = range(t1, t1 + 3)
in
(1000, l)

GC Complete!
Have space for (1000, l)
def range(i, j):
    if (j <= i): false else: (i, range(i+1, j))

def sum(l):
    if l == false: 0 else: l[0] + sum(l[1])

let t1 =
    let l1 = range(0, 3)
      in sum(l1)
    , l = range(t1, t1 + 3)
  in (1000, l)

GC Complete!
Have space for (1000, l)
ex4: recursive data

QUIZ: What should `print(0x21)` show?

(A) `(0, (1, (2, false)))`
(B) `(3, (4, (5, false)))`
(C) `(0, (1, (2, (3, (4, (5, false)))))))`  
(D) `(3, (4, (5, (0, (1, (2, false)))))))`
(E) `(2, (1, (0, (3, (4, (5, false))))) )`

```
2 0x51 1 0x01 0 0x11 5 false 4 0x31 3 0x41
0x00 0x08 0x10 0x18 0x20 0x28 0x30 0x38 0x40 0x48 0x50 0x58 0x60
```
QUIZ: Which cells are “live” on the heap?

(A) 0x00
(B) 0x10
(C) 0x20
(D) 0x30
(E) 0x40
(F) 0x50
or

or

or

or

or

or
Cons

Cons 1

Cons 2

Cons 3

Nil
saltDarkChoco :: Cake

vs.

howToSaltChoco :: Recipe Cake

INGREDIENTS

FOR THE CAKE:

2 ½ cups/310 grams self-rising flour, sifted (see note)
¾ cup/45 grams cocoa powder, sifted
1 ½ cups/295 grams sugar
4 large eggs, lightly beaten
1 ¼ cups/360 milliliters whole milk
1 cup plus 2 tablespoons/255 grams unsalted butter, melted and slightly cooled
7 ounces/200 grams dark chocolate, melted and slightly cooled
2 teaspoons vanilla extract
1 teaspoon flaky sea salt, white or black

FOR THE GANACHE:

1 cup/240 milliliters sour cream
14 ounces/400 grams milk

PREPARATION

Step 1
Heat oven to 350 degrees. Line 2 8-inch round cake tins with parchment paper. Place the flour, cocoa, sugar, eggs, milk, butter, dark chocolate and vanilla in a large bowl and whisk until smooth. (You may need to use a spatula to start, but use a whisk once the ingredients begin to combine.) Divide the mixture evenly between the tins and bake for 35 to 40 minutes or until a wooden skewer inserted into the center comes out clean. Allow to cool in the tins for 10 minutes before turning out onto wire racks to cool completely.

Step 2
Make the ganache: Place the sour cream and melted chocolate in a large bowl. Whisk to combine and refrigerate for 10 to 15 minutes or until firm. Place 1 of the cakes on a cake stand or plate. Spread with half the ganache. Top with the remaining cake and ganache. Sprinkle with the salt to serve.

Tip
To make your own self-rising flour, combine 2 1/2 cups/320 grams all-purpose flour; 1 tablespoon plus 3/4 teaspoon baking powder; and 1/2 teaspoon plus 1/8 teaspoon fine salt. Use the entire amount in place of the self-rising flour listed in the ingredients.
Node

Node

Node

Leaf 3

Leaf

Leaf 4

Leaf 5

Leaf

Leaf 1

Leaf 2
PW-1: Async Time
PW-2: Async Space
PW-3: Sound & Complete
PW-4: Invariant Synthesis
PW-5: Modular Effects
PW-6: Coord. Service
PW-7: Microservice Flows
let rec wwhile (f, b) = 
  let (b', c') = f b in 
  if c' = true then wwhile (f b') 
  else b'

RITE:  (f, b')

SEMINAL: ((f b'); [[...]])

let rec clone x n = 
  if n <= 0 then [] else 
  x :: clone (n-1)

RITE:  clone (n-1) n

SEMINAL: clone [[...]] (n-1)

let sqsum xs = 
  let f a x = a + (x ** 2) in 
  let base = 0 in 
  List.fold_left f base xs

RITE:  (x * x)

SEMINAL: (x + 2)
let rec clone x n =
  if n <= 0 then [] else
  x :: clone (n-1)

RITE: clone (n-1) n

SEMINAL: clone [[...]] (n-1)
Model
{Policy}

Controller

2.query
3.data
1.request

View

4.response
Model
@ Policy

Controller

View

1. request

2. query

3. data

4. response
The question then is, is it possible to use (overapproximate) static analyses to precisely report that the target location is reachable, *without* actually finding a feasible path to it? Intuitively, the code through the for-loop is irrelevant to the reachability of the error location. In other words, if we can reason that *there exists some path* from the start to the end of the loop, *i.e.*, from location 3: to 5:, and along such a path, the variables $x$, $a$ are not modified, then we are guaranteed that the location ERR: can be reached.
INil

ICons

ICons

ICons

1

2

3

INil
ea >>= (\va -> eb)
next :: ST0 String
next = ST0C (\s -> (s+1, show s))

wtf :: ST0 [String]
wtf = next >>= (\v -> return [v])

quiz = evalState wtf 1
next :: ST0 String
next = ST0C (\s -> (s+1, show s))

wtf :: ST0 [String]
wtf = next >>= (\v1 -> next >>= (\v2 -> return [v1, v2]))

quiz = evalState wtf 1
calc0

int  intOp  calc0

10  -  5  -  5