Imperial College London

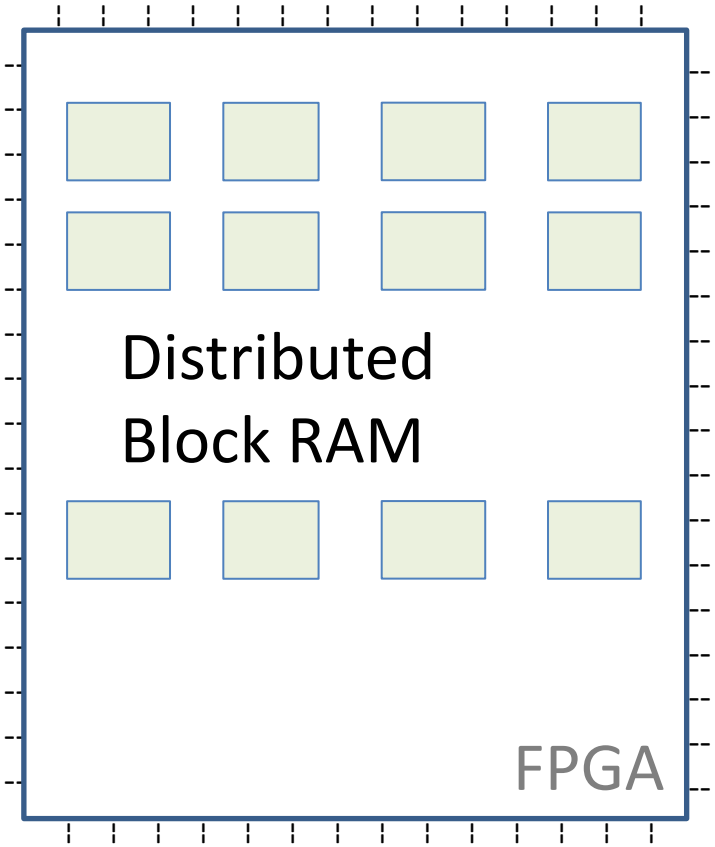
Separation Logic-Assisted Code Transformations for Efficient High-Level Synthesis

Felix Winterstein, Samuel Bayliss,
George Constantinides

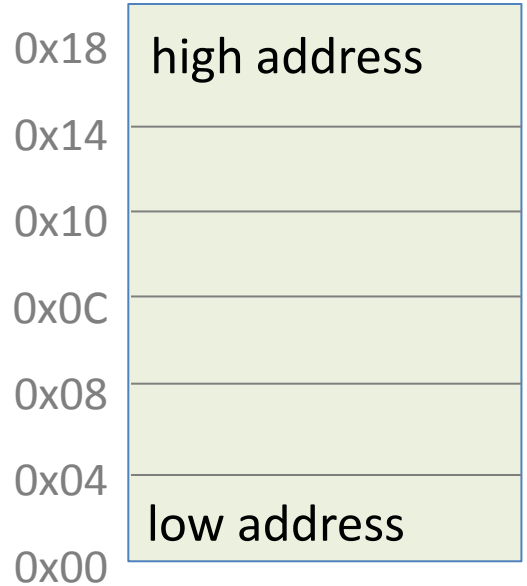
12 May 2014

esa

Background

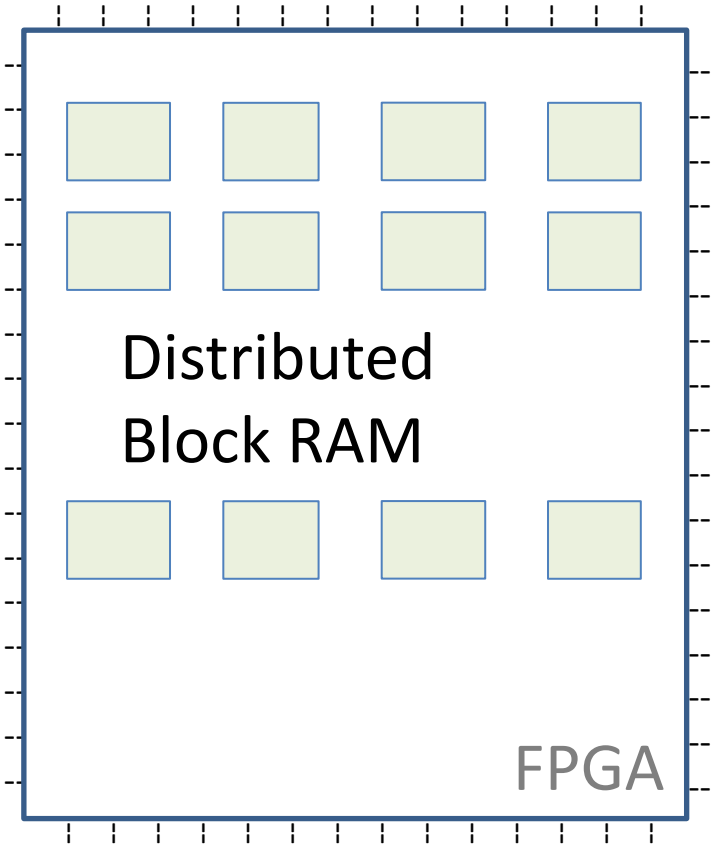


SW memory model



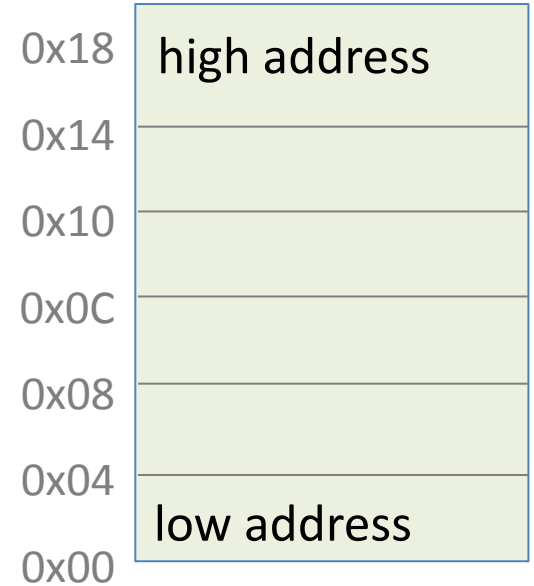

```
int main() {
  x = A[i];
  p = new int;
  *p = 3;
  ...
}
```

Background



HLS

SW memory model



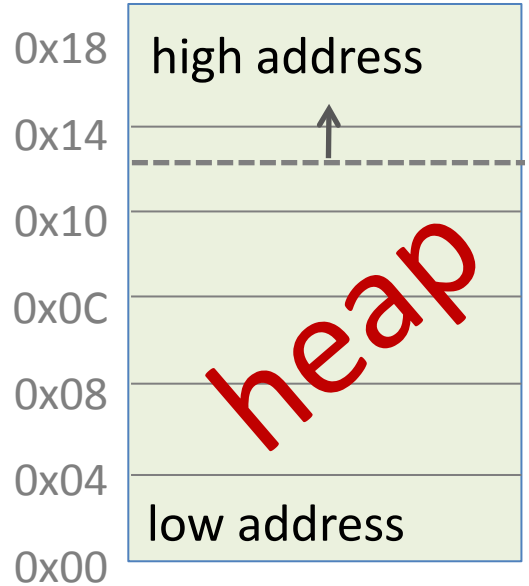

```
int main() {
  x = A[i];
  p = new int;
  *p = 3;
  ...
}
```

Background

Lack of automated optimizations …

- … for programs using pointers
- … because pointers are difficult to analyze
- … and memory is allocated, disposed, and reused at run-time
- Yet widely used in SW

SW memory model




```
int main() {
  x = A[i];
  p = new int;
  *p = 3;
  …
}
```

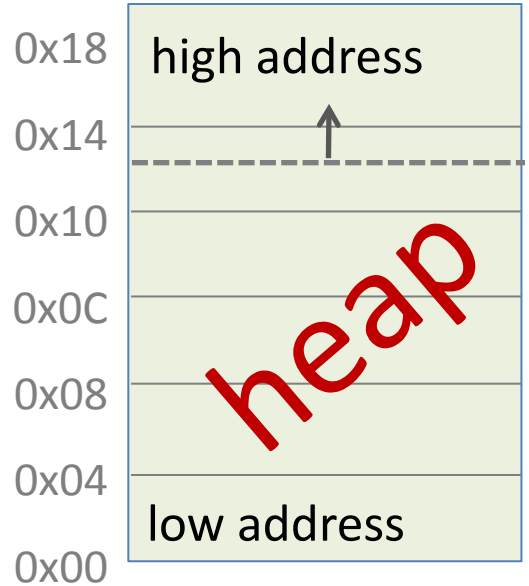
Background

Lack of automated optimizations …

- … for programs using pointers
- … because pointers are difficult to analyze
- … and memory is allocated, disposed, and reused at run-time
- Yet widely used in SW

This work takes a step towards closing this gap

SW memory model

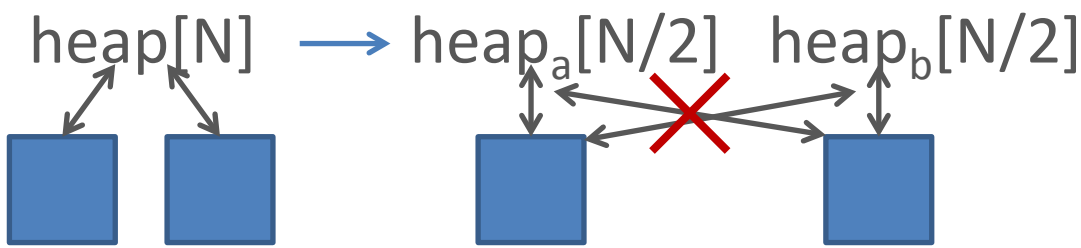



```
int main() {
  x = A[i];
  p = new int;
  *p = 3;
  ...
}
```

Our contribution

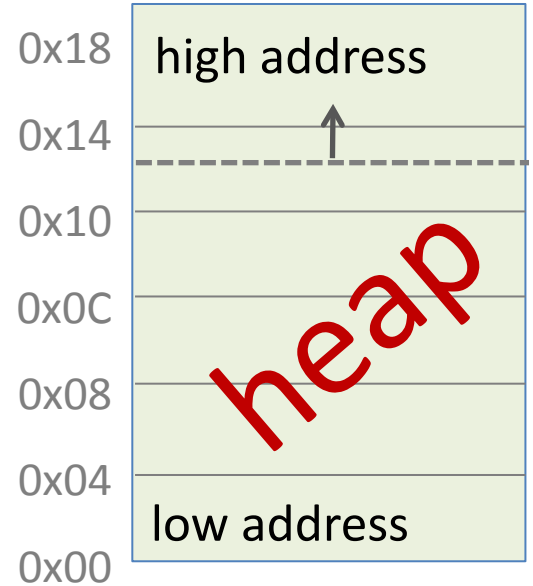
Our goal

- Partition heap-allocated data structures ('heaplets')
- Synthesize parallel memory accesses



- Ensure that heap partitions are 'private'

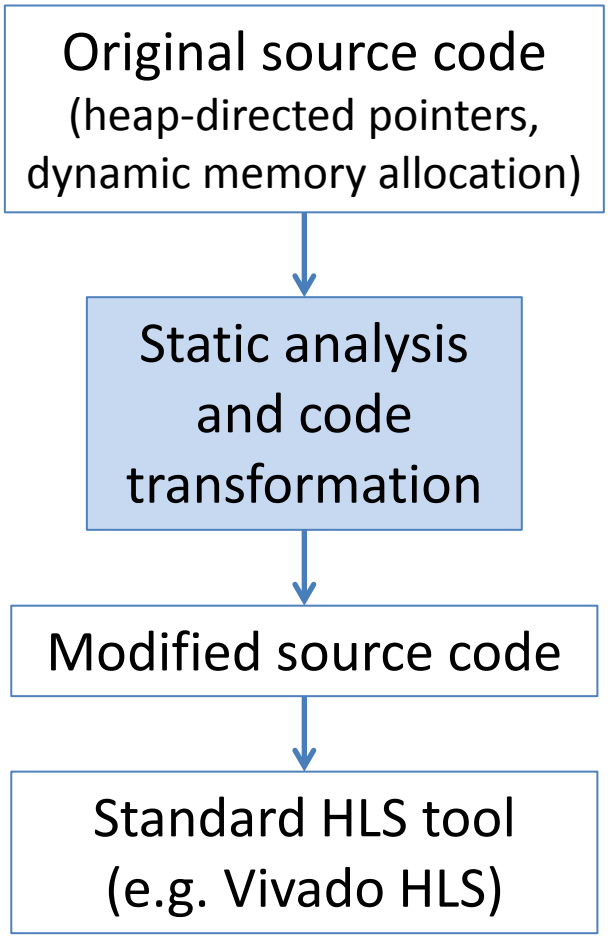
SW memory model



Outline

Executive summary

- Static program analysis
  - Analyse pointer-based memory accesses
  - Identify independent, disjoint regions in memory
- Source-to-source transformations
  - Partition heap across on-chip memory banks
  - Automatic parallelization

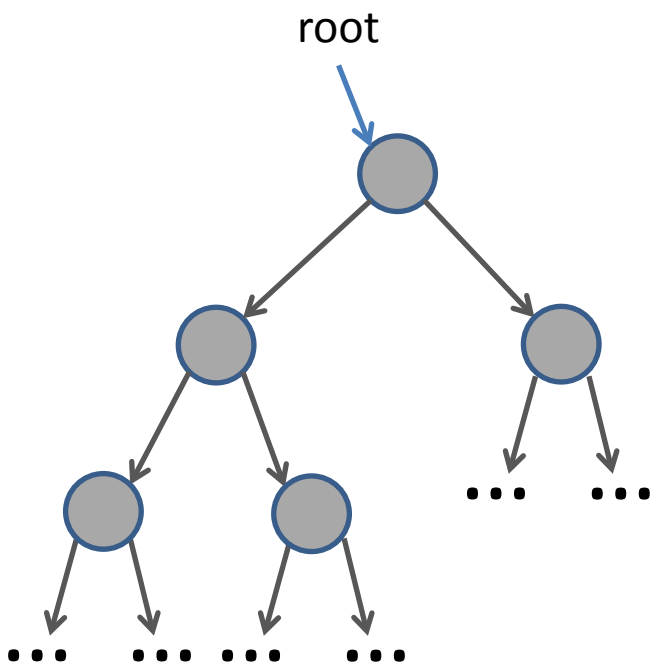


Motivating example

Can we parallelize this loop?

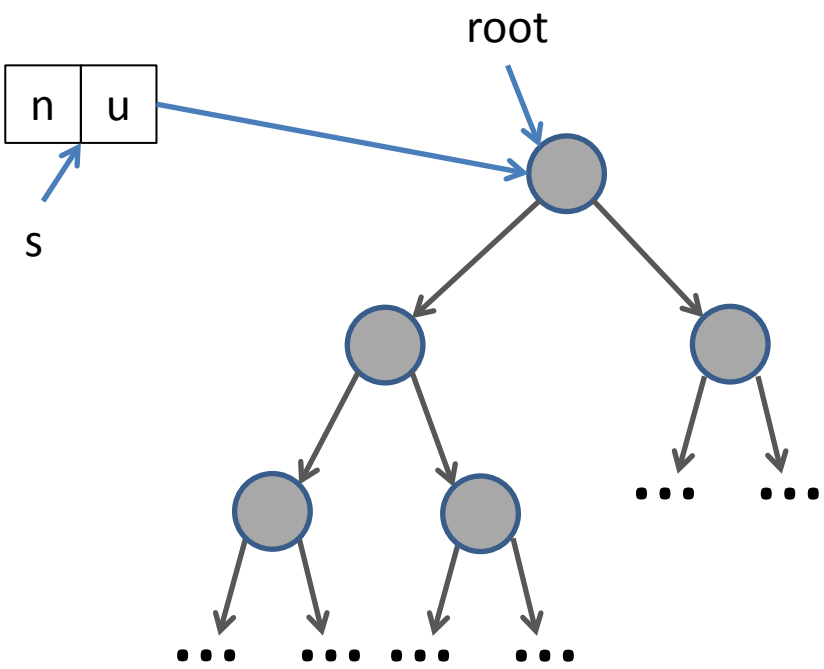
```
s = PUSH(root, s);
while s!=0 do
   s = POP(&u, s);
   … do something
   if (u->left!= 0) && (u->right!=0) then
      s = PUSH(u->right, s);
      s = PUSH(u->left, s);
   end if
   delete u;
end while
```

Motivating example



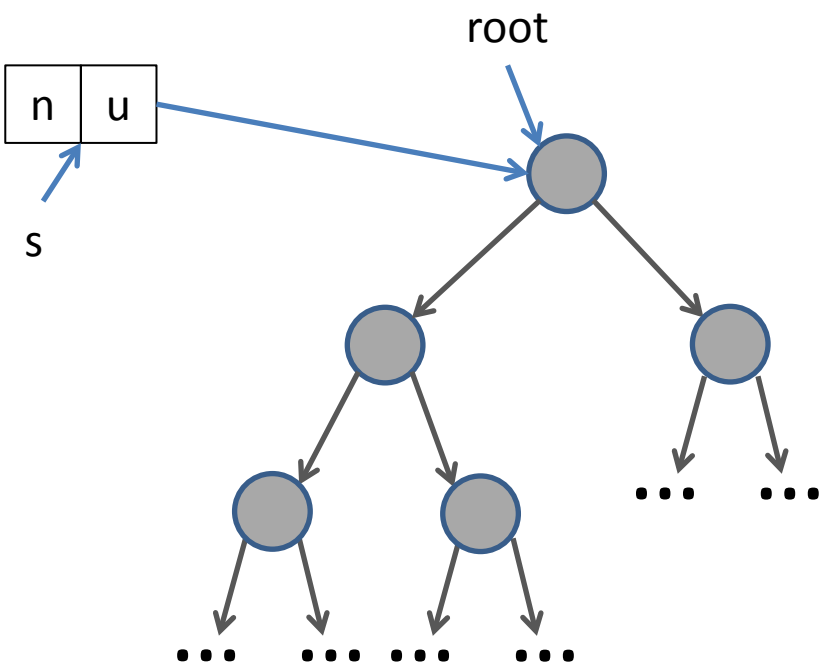

```
s = PUSH(root, s);
while s!=0 do
   s = POP(&u, s);
   … do something
   if (u->left!= 0) && (u->right!=0) then
      s = PUSH(u->right, s);
      s = PUSH(u->left, s);
   end if
   delete u;
end while
```

Motivating example



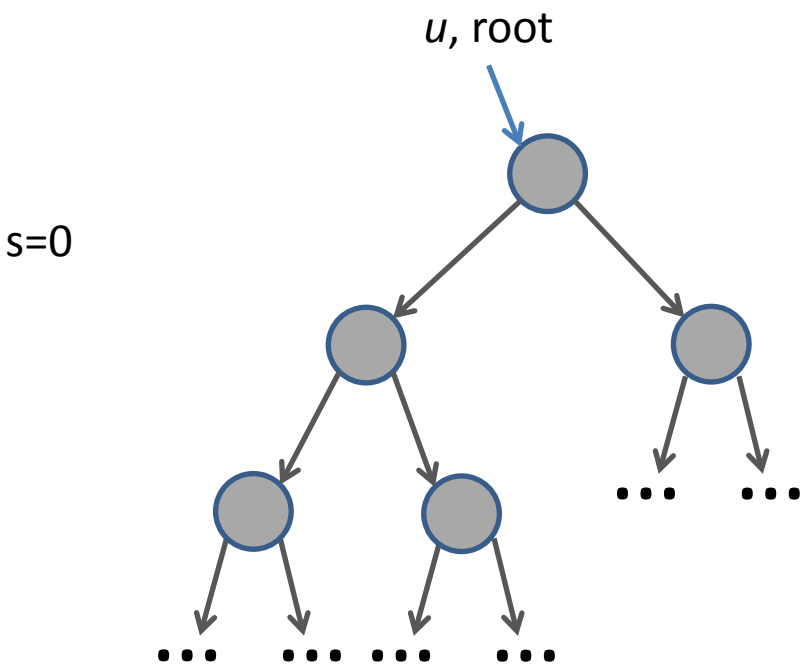

```
s = PUSH(root, s);
while s!=0 do
    s = POP(&u, s);
    … do something
    if (u->left!= 0) && (u->right!=0) then
        s = PUSH(u->right, s);
        s = PUSH(u->left, s);
    end if
    delete u;
end while
```

Motivating example



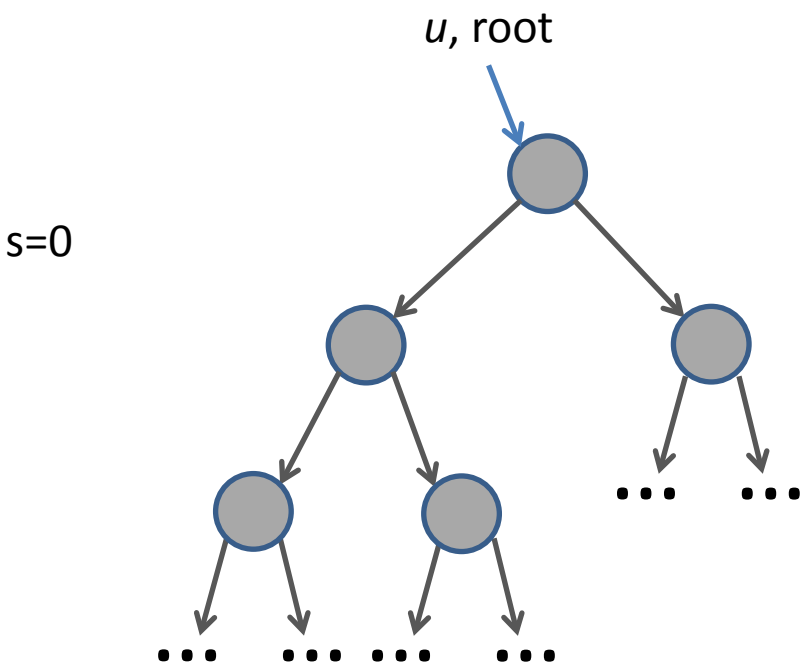

```
s = PUSH(root, s);
while s!=0 do
    s = POP(&u, s);
    … do something
    if (u->left!= 0) && (u->right!=0) then
        s = PUSH(u->right, s);
        s = PUSH(u->left, s);
    end if
    delete u;
end while
```

Motivating example



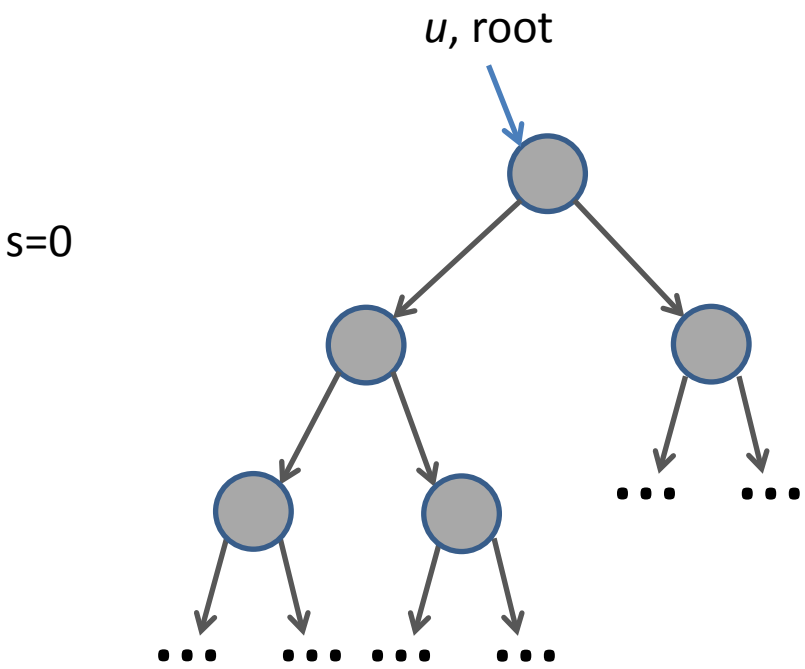

```
s = PUSH(root, s);
while s!=0 do
    s = POP(&u, s);
    … do something
    if (u->left!= 0) && (u->right!=0) then
        s = PUSH(u->right, s);
        s = PUSH(u->left, s);
    end if
    delete u;
end while
```

Motivating example



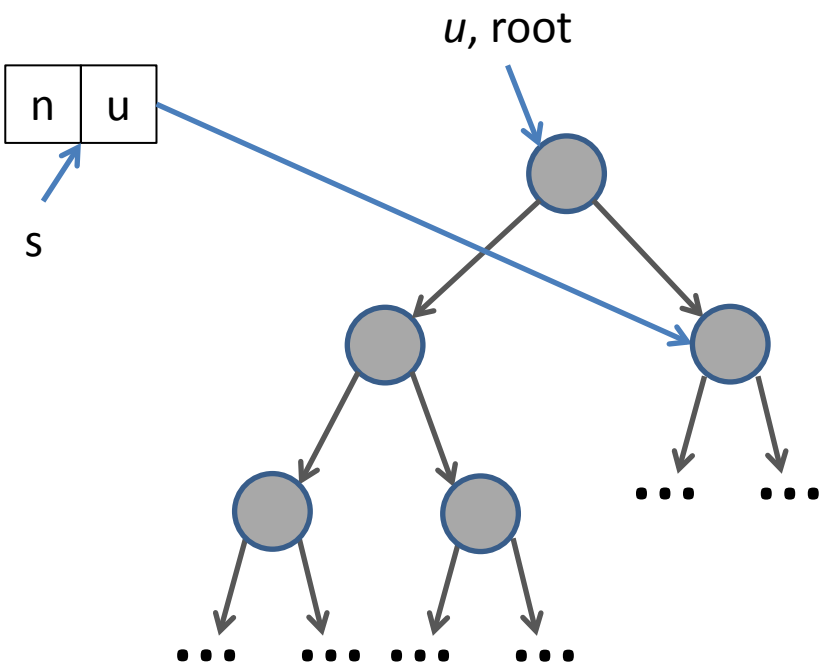

```
s = PUSH(root, s);
while s!=0 do
   s = POP(&u, s);
   … do something
   if (u->left!= 0) && (u->right!=0) then
      s = PUSH(u->right, s);
      s = PUSH(u->left, s);
   end if
   delete u;
end while
```

Motivating example



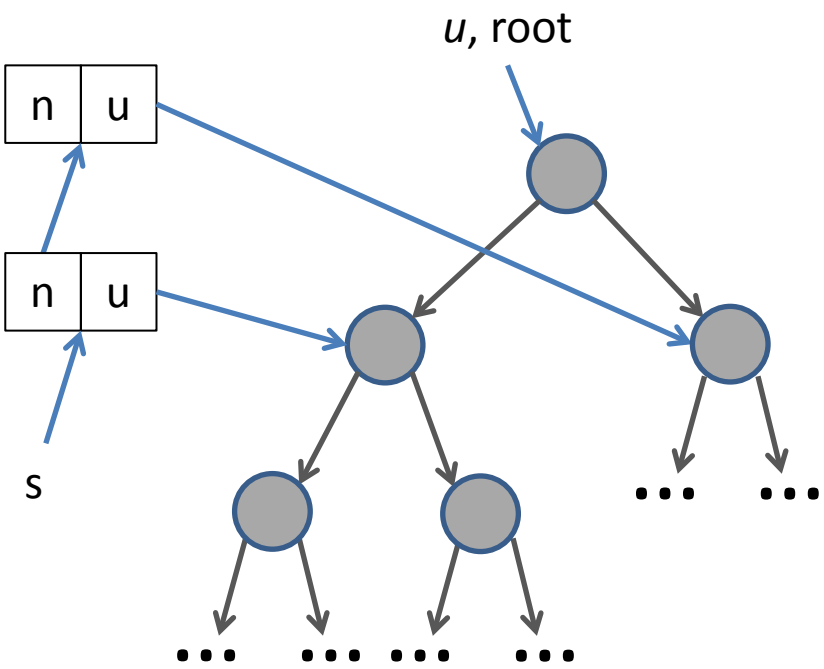

```
s = PUSH(root, s);
while s!=0 do
   s = POP(&u, s);
   … do something
   if (u->left!= 0) && (u->right!=0) then
      s = PUSH(u->right, s);
      s = PUSH(u->left, s);
   end if
   delete u;
end while
```

Motivating example



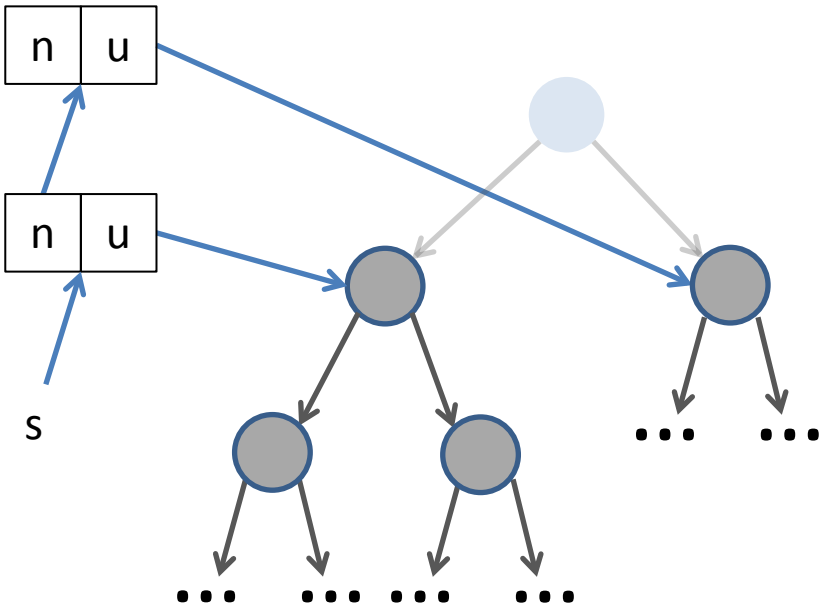

```
s = PUSH(root, s);
while s!=0 do
   s = POP(&u, s);
   … do something
   if (u->left!= 0) && (u->right!=0) then
      s = PUSH(u->right, s);
      s = PUSH(u->left, s);
   end if
   delete u;
end while
```

Motivating example



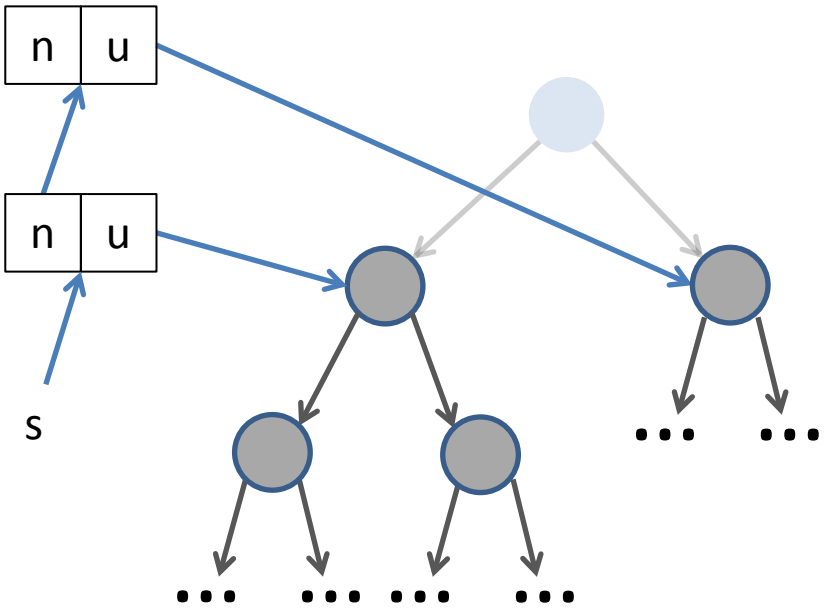

```
s = PUSH(root, s);
while s!=0 do
   s = POP(&u, s);
   … do something
   if (u->left!= 0) && (u->right!=0) then
      s = PUSH(u->right, s);
      s = PUSH(u->left, s);
   end if
   delete u;
end while
```

Motivating example




```
s = PUSH(root, s);
while s!=0 do
   s = POP(&u, s);
   … do something
   if (u->left!= 0) && (u->right!=0) then
      s = PUSH(u->right, s);
      s = PUSH(u->left, s);
   end if
   delete u;
end while
```

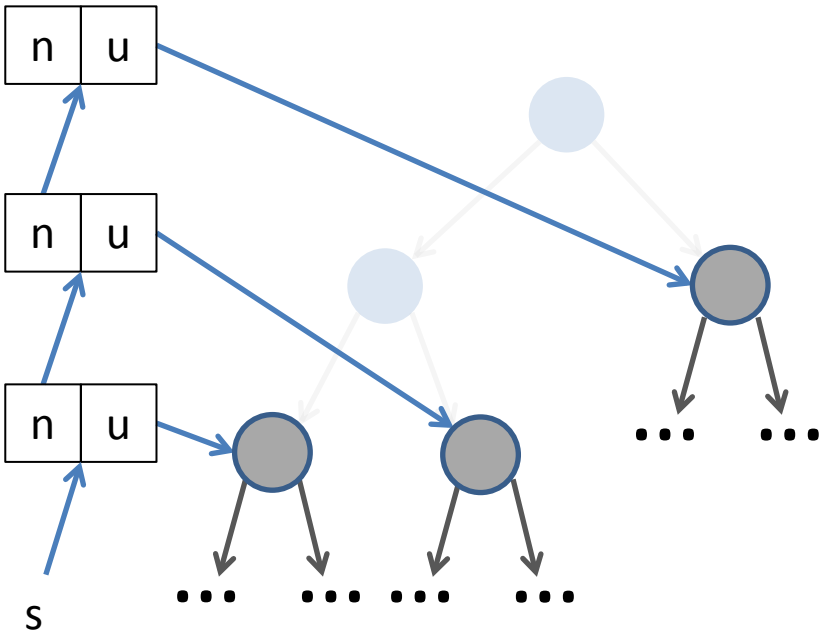
Motivating example




```
s = PUSH(root, s);
while s!=0 do
   s = POP(&u, s);
   … do something
   if (u->left!= 0) && (u->right!=0) then
      s = PUSH(u->right, s);
      s = PUSH(u->left, s);
   end if
   delete u;
end while
```

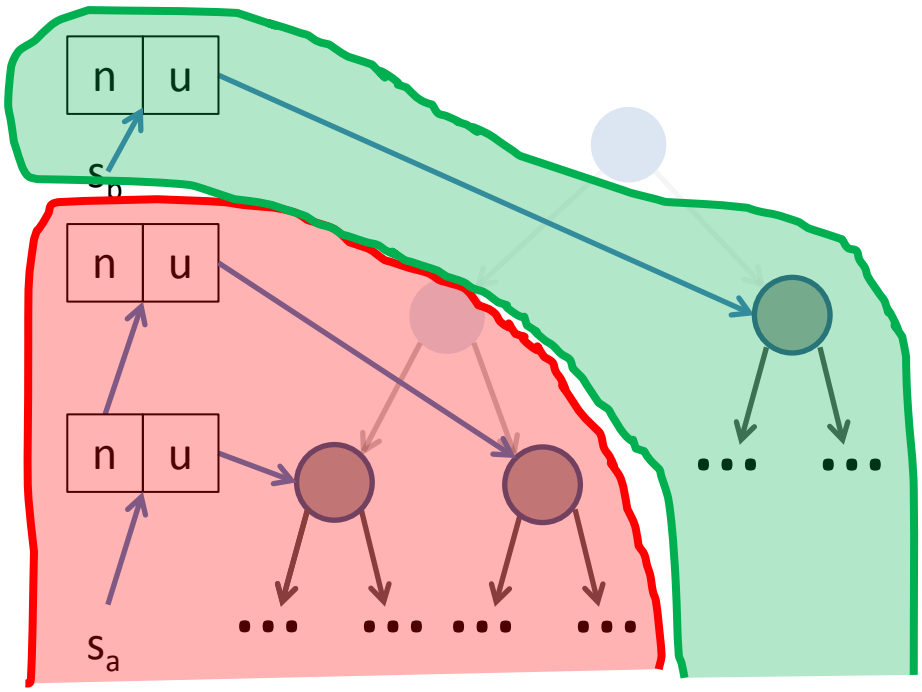
Motivating example

After two iterations…




```
s = PUSH(root, s);
while s!=0 do
   s = POP(&u, s);
   … do something
   if (u->left!= 0) && (u->right!=0) then
      s = PUSH(u->right, s);
      s = PUSH(u->left, s);
   end if
   delete u;
end while
```

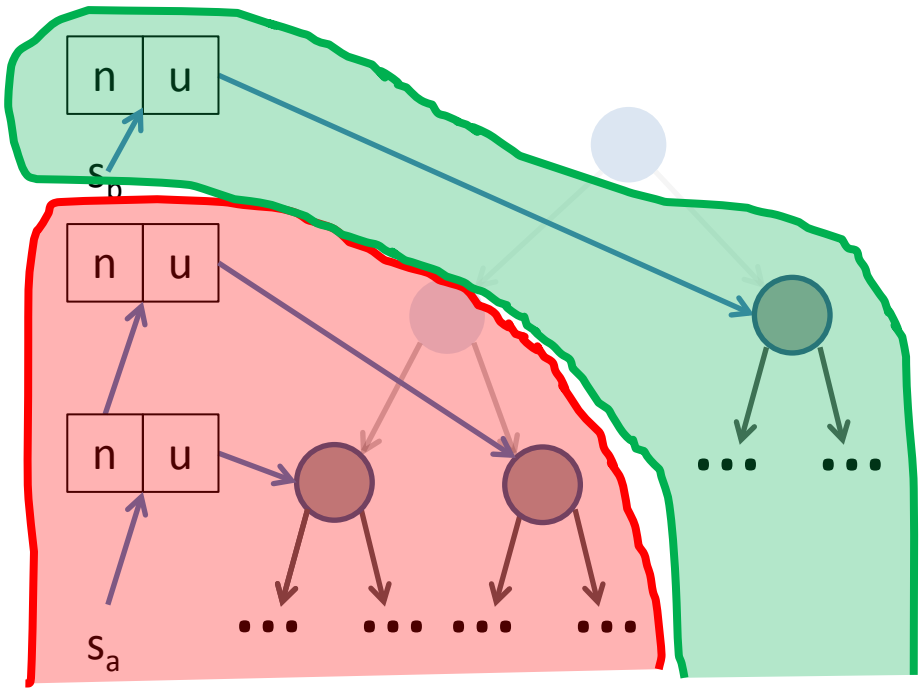
Motivating example




```
s = PUSH(root, s);
while s!=0 do
   s = POP(&u, s);
   … do something
   if (u->left!= 0) && (u->right!=0) then
      s = PUSH(u->right, s);
      s = PUSH(u->left, s);
   end if
   delete u;
end while
```

- Partition linked list and tree

Motivating example



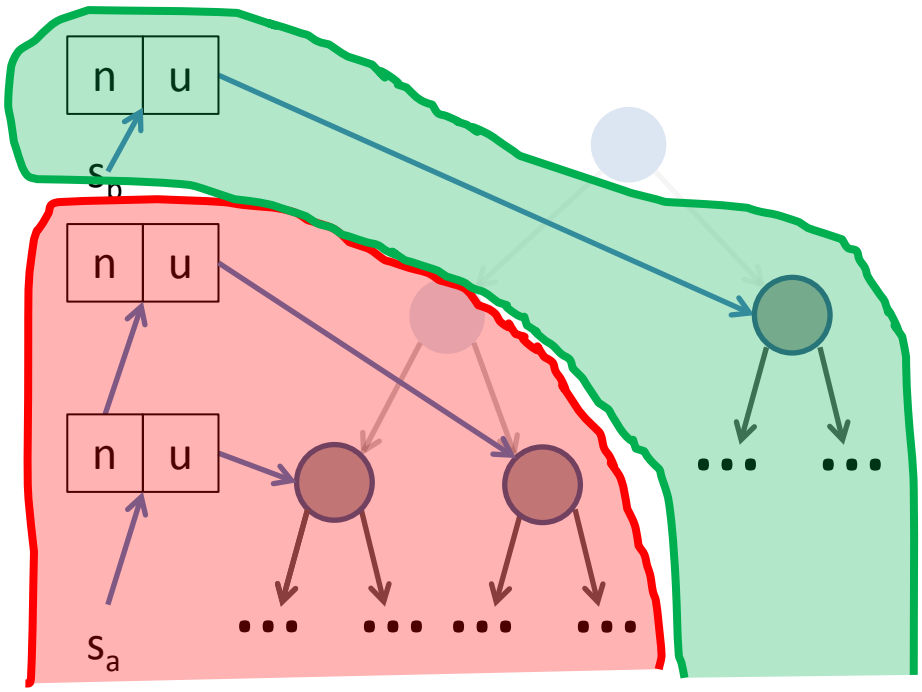
… preamble (accessing root node)

while s_a!=0 **do**
  … loop body (access left sub-tree)
end while

while s_b!=0 **do**
  … loop body (access right sub-tree)
end while

- Partition linked list and tree

Motivating example



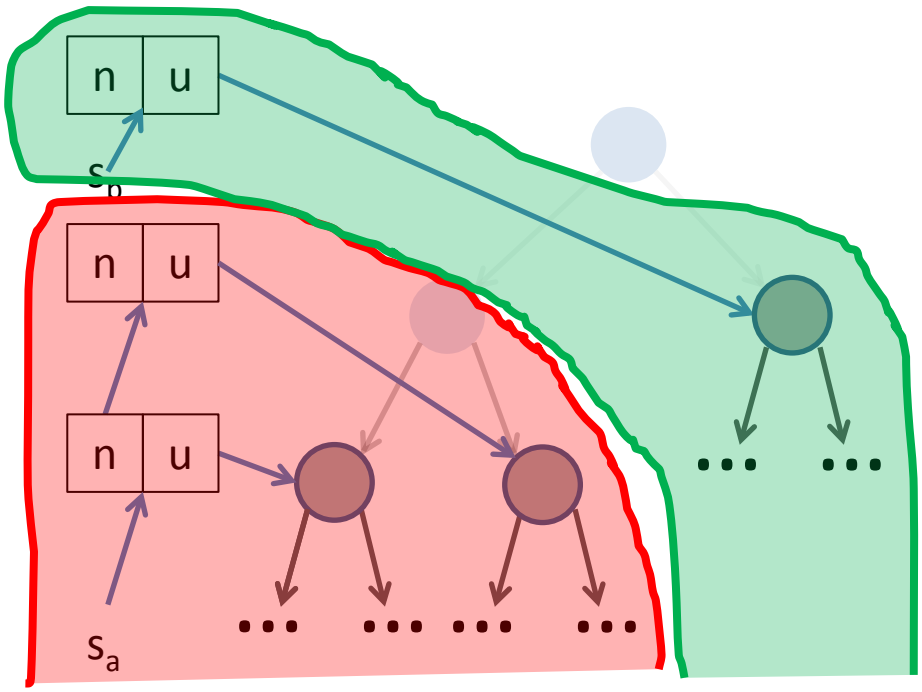
… preamble (accessing root node)

while s_a!=0 **do**
  … loop body (access left sub-tree)
end while

while s_b!=0 **do**
  … loop body (access right sub-tree)
end while

- Partition linked list and tree
- Will the red loop ever access data in the green partition?

Motivating example



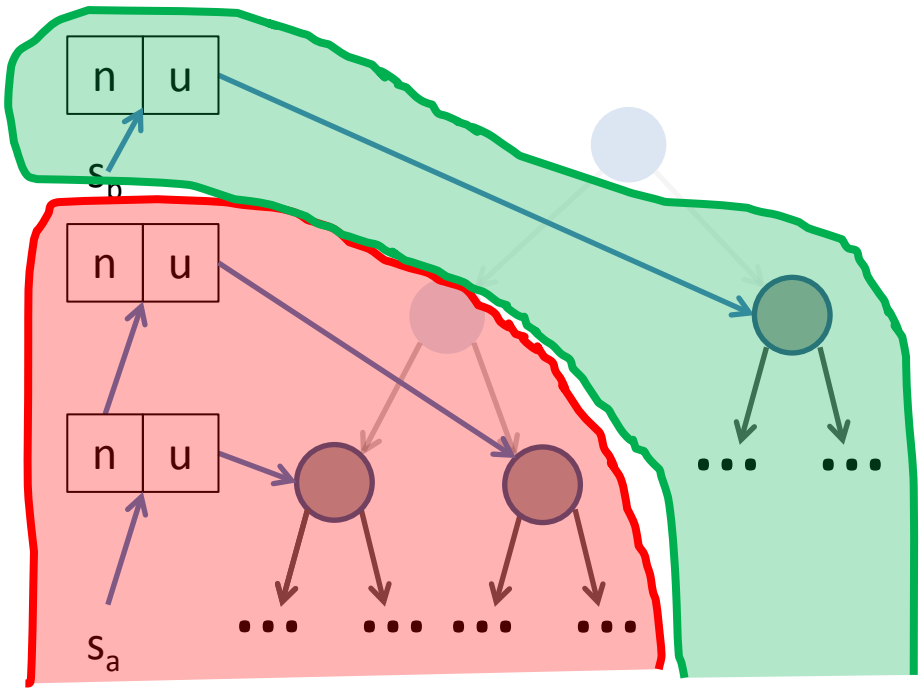
… preamble (accessing root node)

while s_a!=0 **do**
  … loop body (access left sub-tree)
end while

while s_b!=0 **do**
  … loop body (access right sub-tree)
end while

- Partition linked list and tree
- Will the red loop ever access data in the green partition? No!

Motivating example



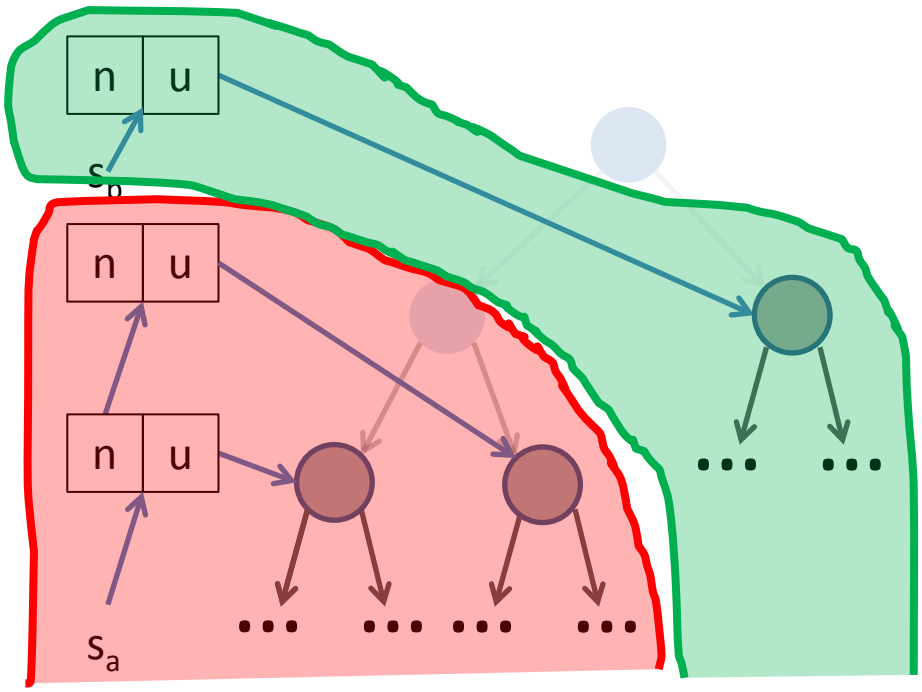
… preamble (accessing root node)

while s_a!=0 **do**
  … loop body (access left sub-tree)
end while

while s_b!=0 **do**
  … loop body (access right sub-tree)
end while

- Partition linked list and tree
- Will the red loop ever access data in the green partition? No!
- Parallelization is legal (does not violate data dependencies)

Motivating example



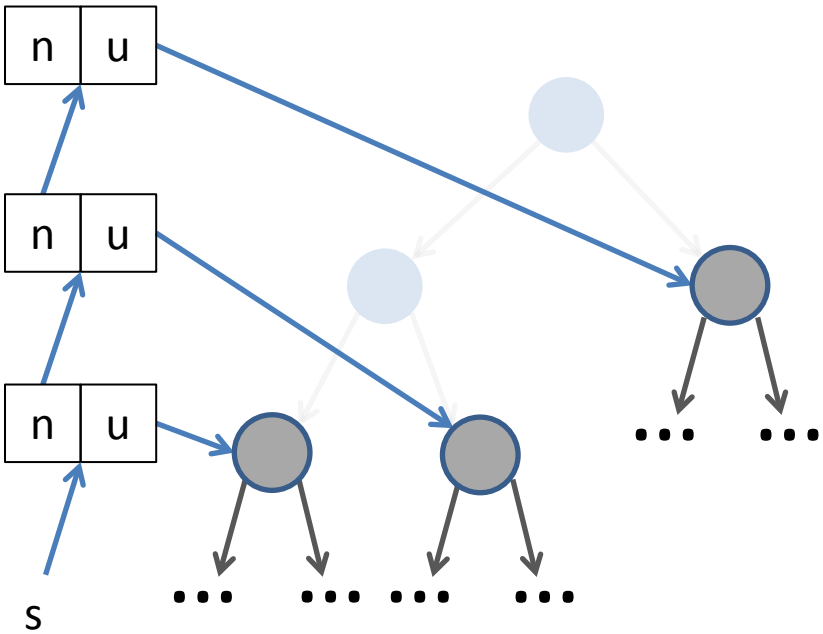
… preamble (accessing root node)

while s_a!=0 **do**
  … loop body (access left sub-tree)
end while

while s_b!=0 **do**
  … loop body (access right sub-tree)
end while

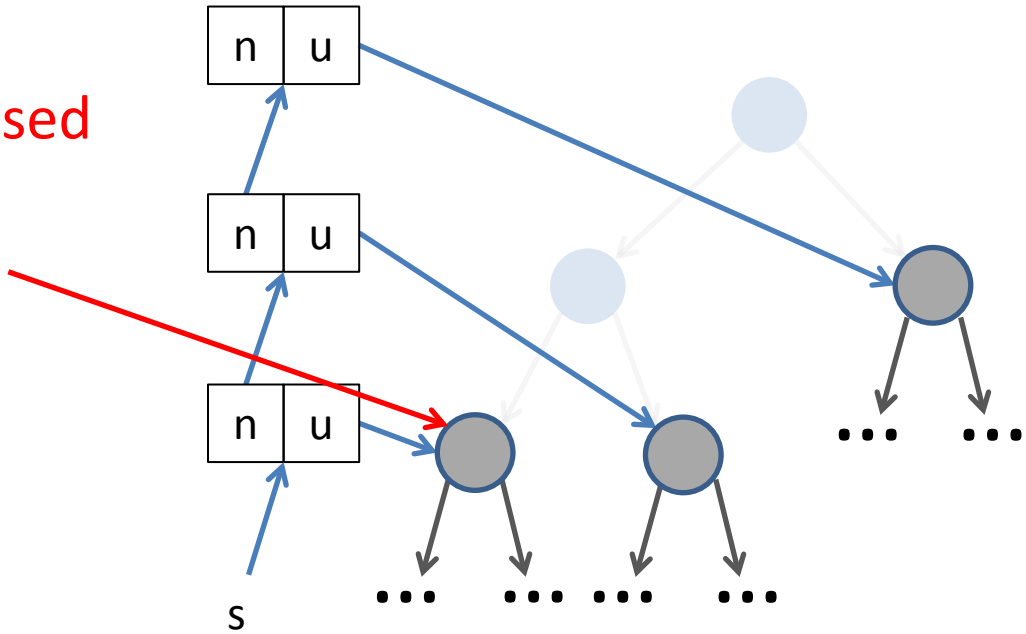
- Partition linked list and tree
- Will the red loop ever access data in the green partition? No!
- Parallelization is legal (does not violate data dependencies)
- Why is it hard for a tool to figure this out?

The problem

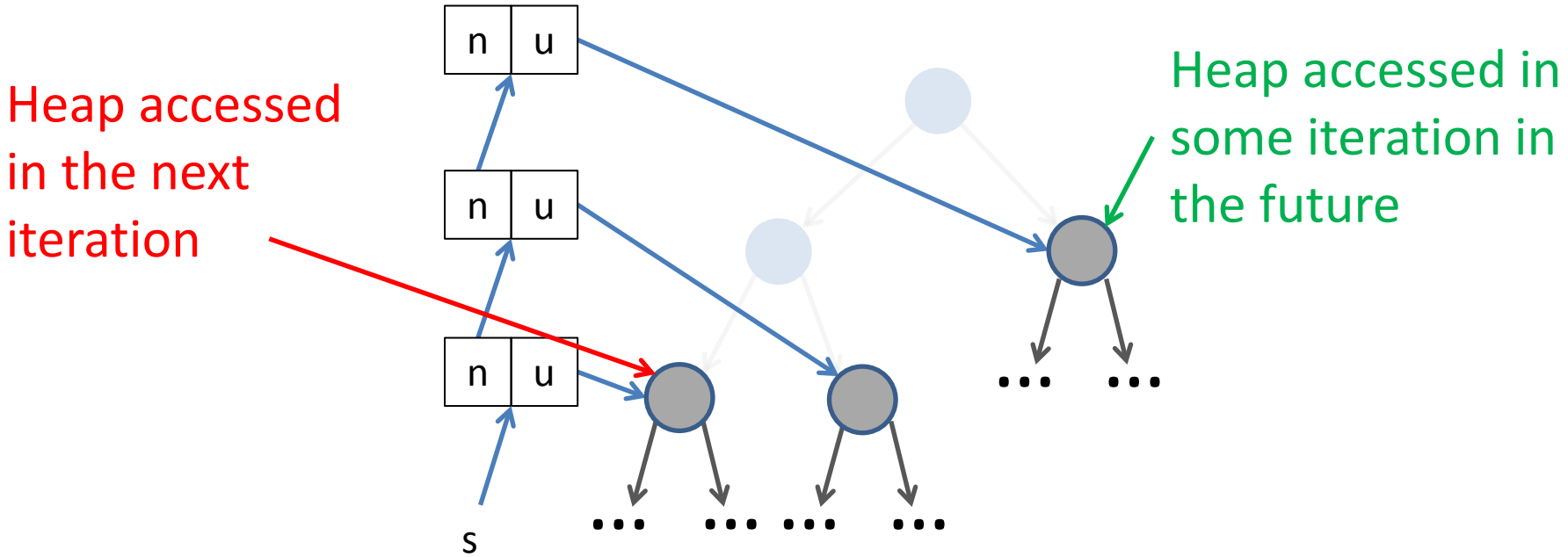


The problem

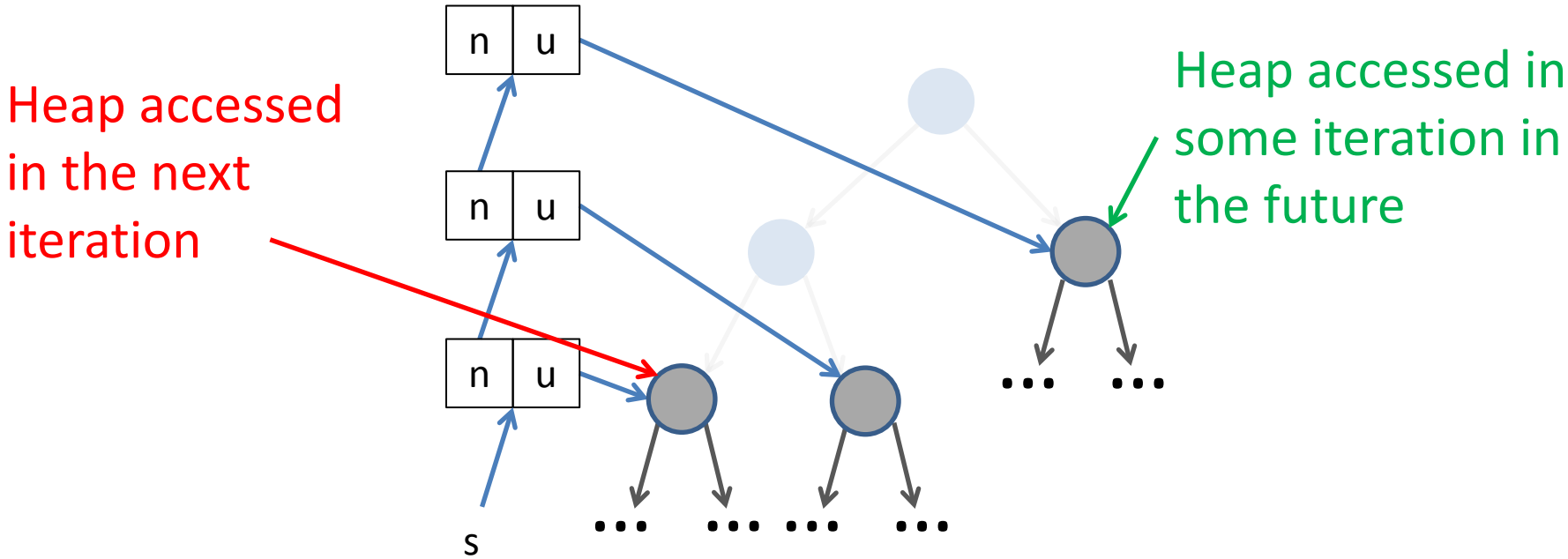
Heap accessed in the next iteration



The problem

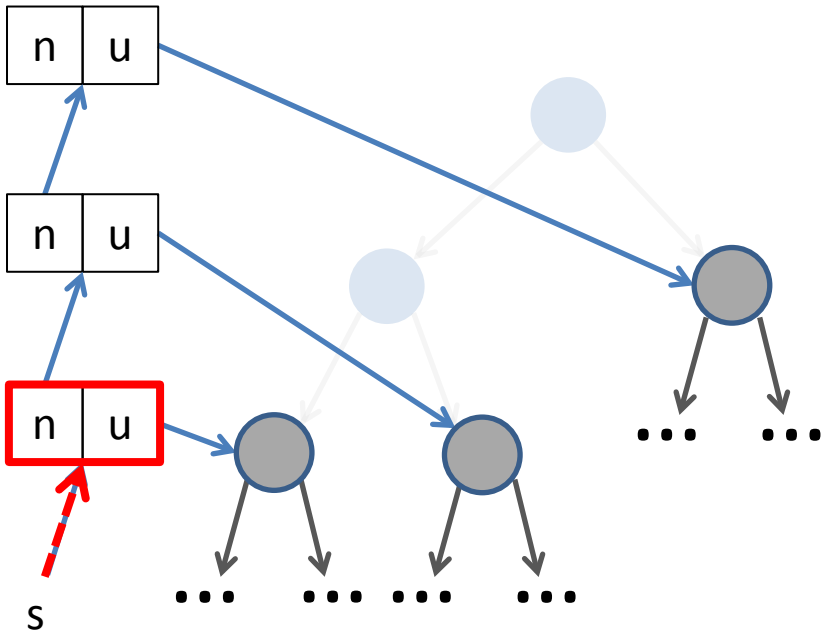

Heap accessed in the next iteration
Heap accessed in some iteration in the future
n
u
n
u
n
u
s

The problem



- Do these iterations access the same memory cell?

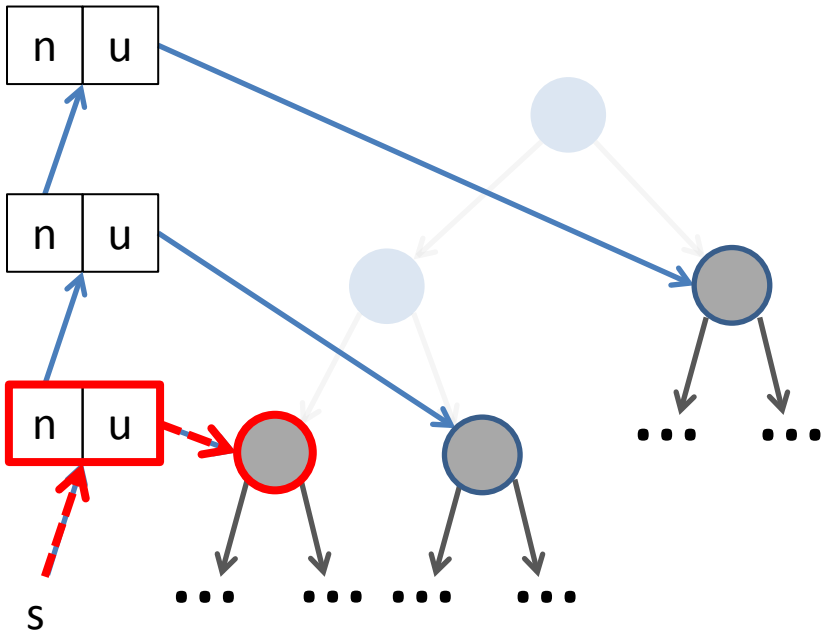
The problem



- Do these iterations access the same memory cell?

  heap[s]

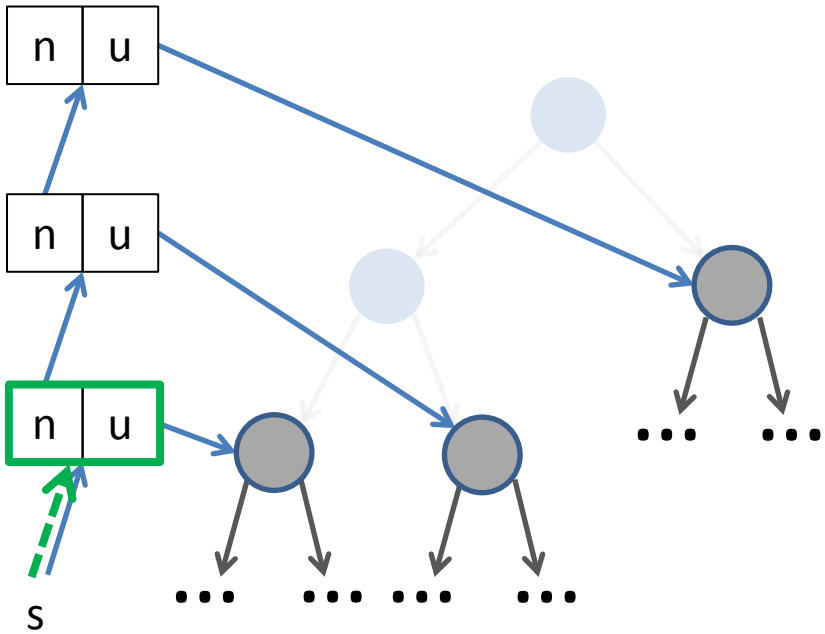
The problem



- Do these iterations access the same memory cell?

  heap[heap[s].u]

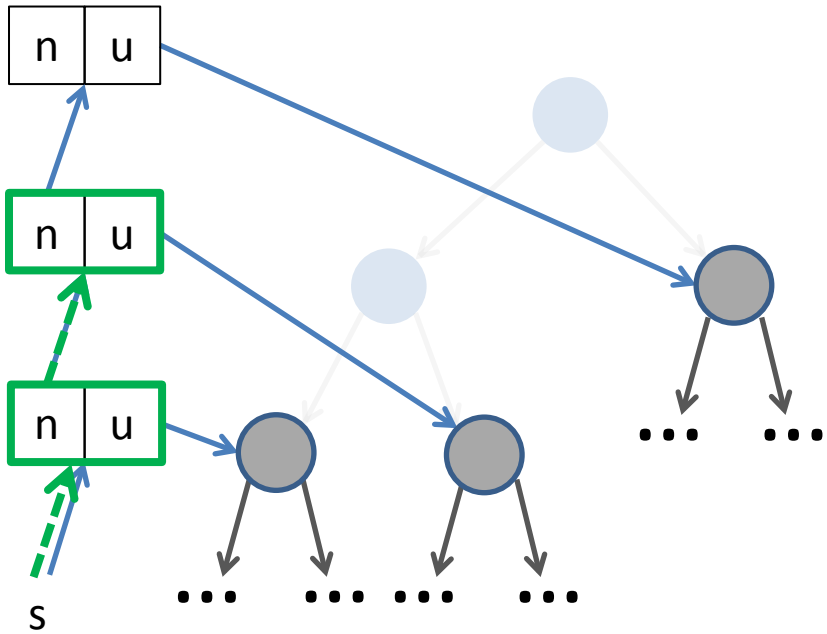
The problem



- Do these iterations access the same memory cell?

heap[heap[s].u] heap[s]

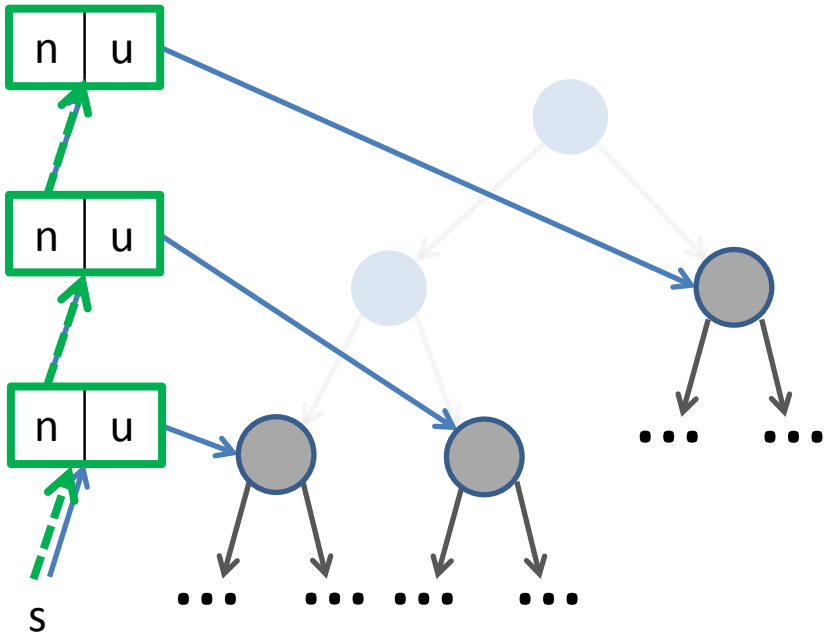
The problem



- Do these iterations access the same memory cell?

heap[heap[s].u] heap[heap[s].n]

The problem

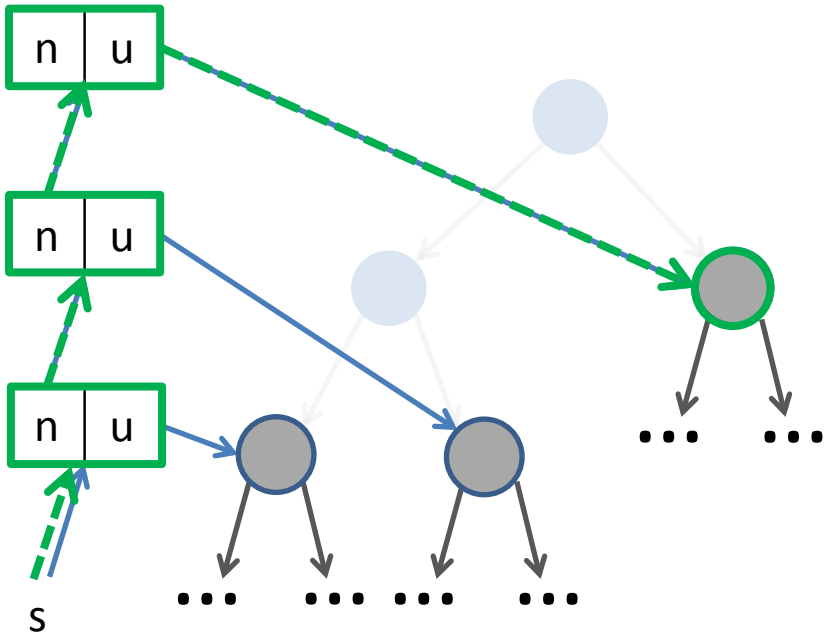


- Do these iterations access the same memory cell?

heap[heap[s].u]

heap[heap[heap[s].n].n]

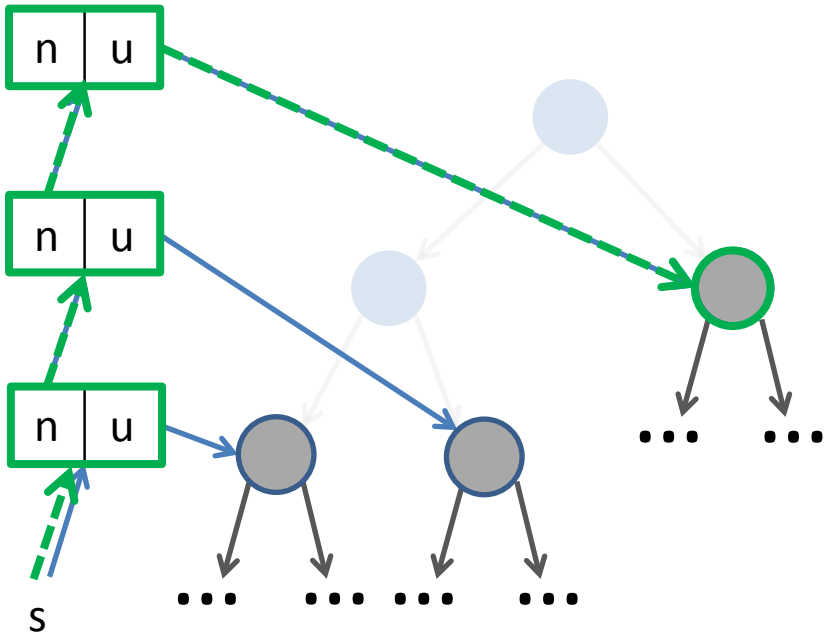
The problem



- Do these iterations access the same memory cell?

heap[heap[s].u] heap[heap[heap[heap[s].n].n].u]

The problem

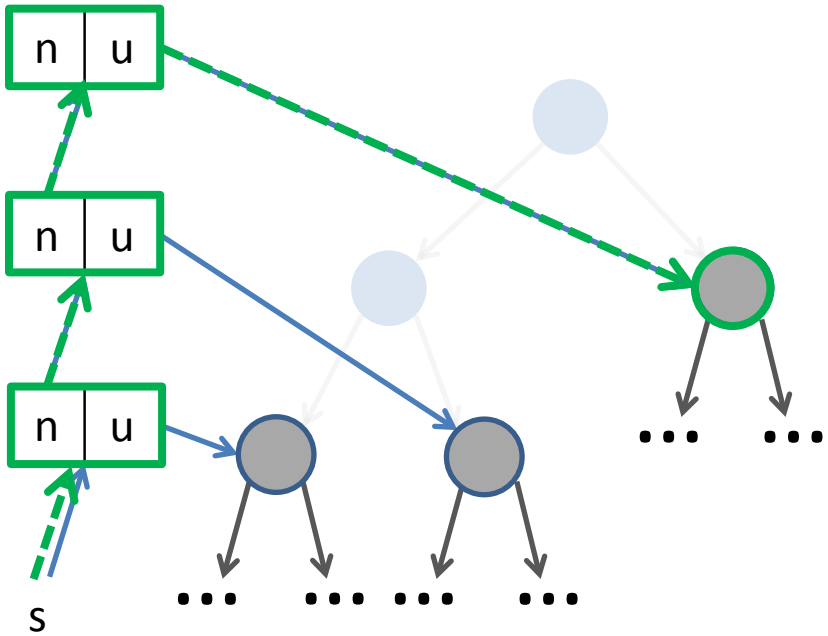


- Do these iterations access the same memory cell?

heap[heap[s].u] heap[heap[heap[heap[s].n].n].u]

= ?

The problem



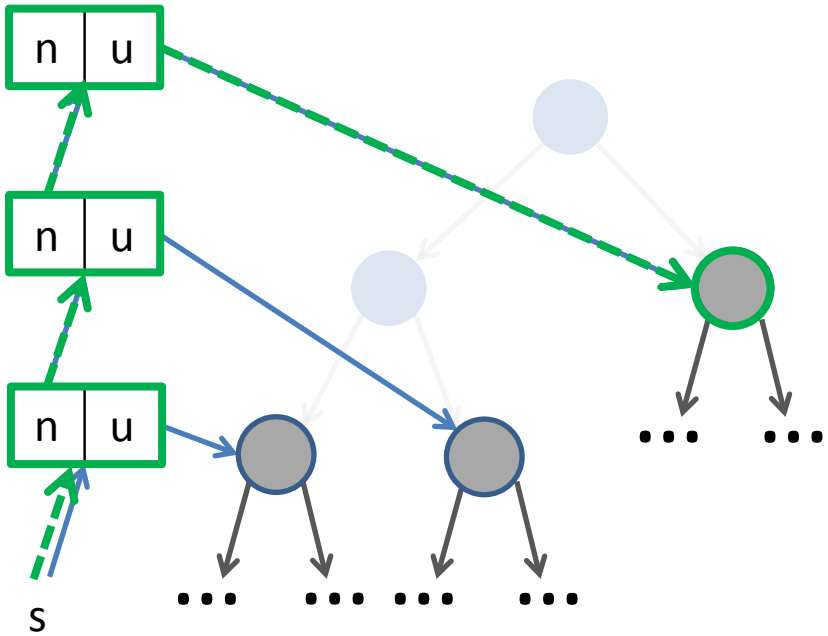
Traverse a linked list

- Do these iterations access the same memory cell?

heap[heap[s].u] heap[heap[heap[heap[s].n].n].u]

= ?

The problem

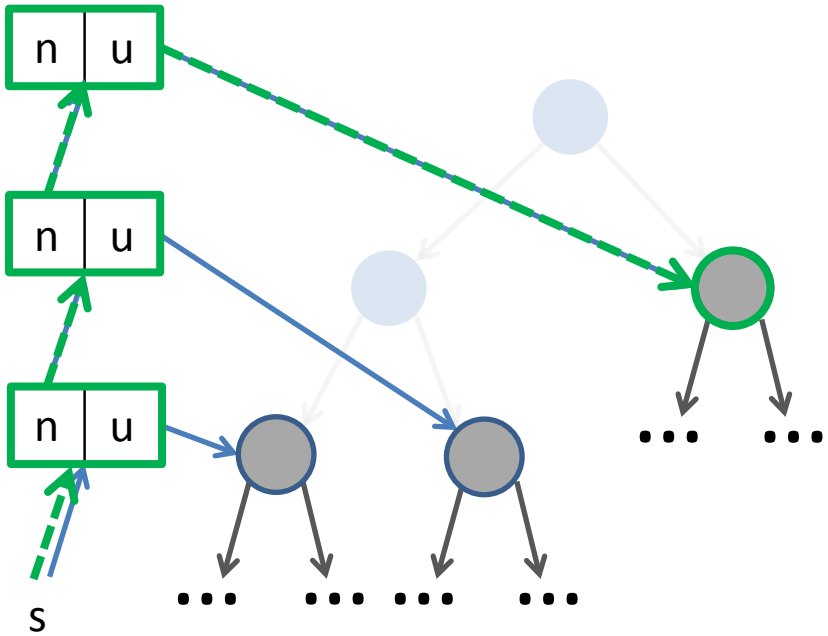


... which has links to sub-trees

- Do these iterations access the same memory cell?

heap[heap[s].u] = ? heap[heap[heap[heap[s].n].n].u]

The problem



… which has links to sub-trees

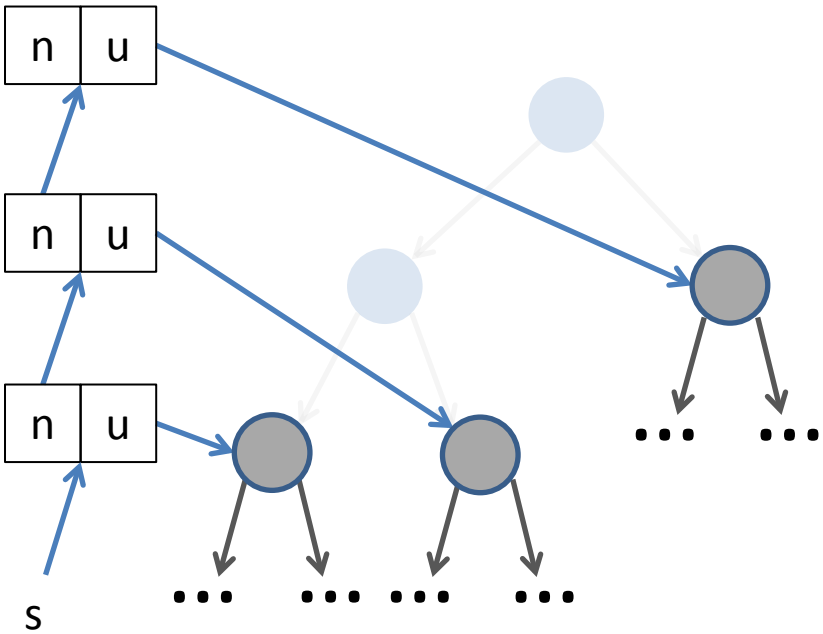
- Do these iterations access the same memory cell?

heap[heap[s].u] = ? heap[heap[heap[heap[s].n].n].u]

- Need to reason about structure, heap layout and disjointness
- None of this is explicit in the above representation

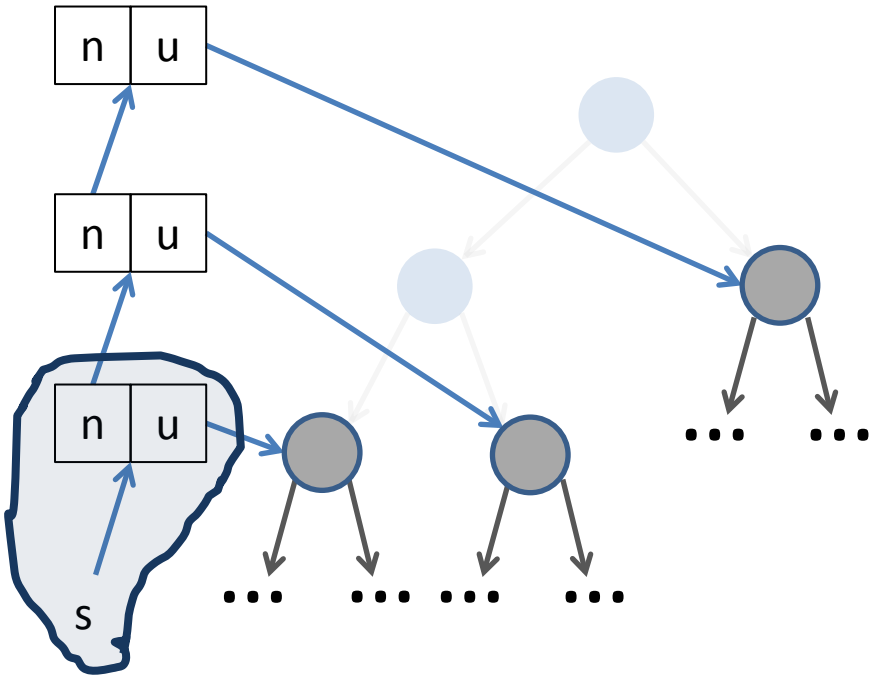
Classical first order logic

Describe heap layout with formulae



Classical first order logic

Describe heap layout with formulae

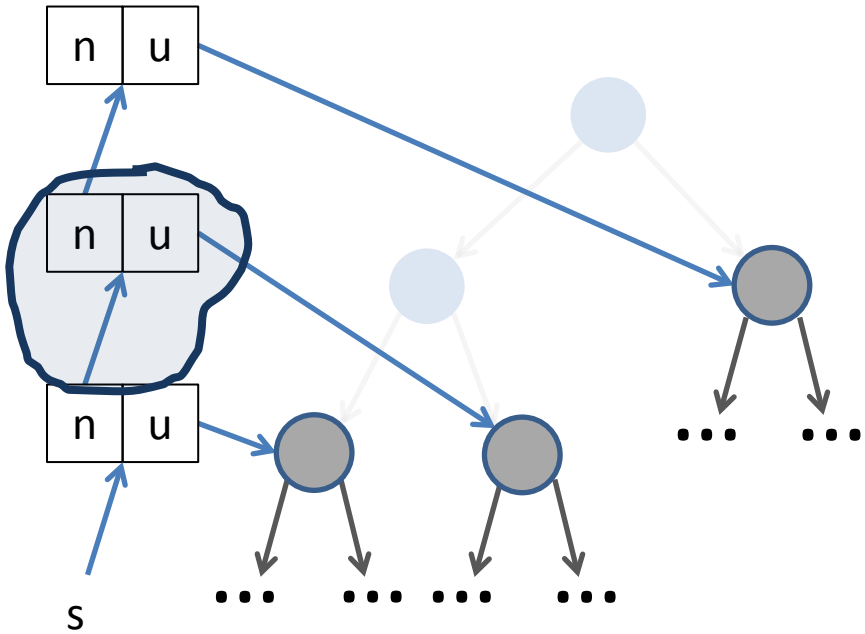


$$\underbrace{s \rightarrow}\ \underbrace{[u\colon u_1', n\colon s_1']}$$

"*s* points to a record with fields *u* and *n*"

Classical first order logic

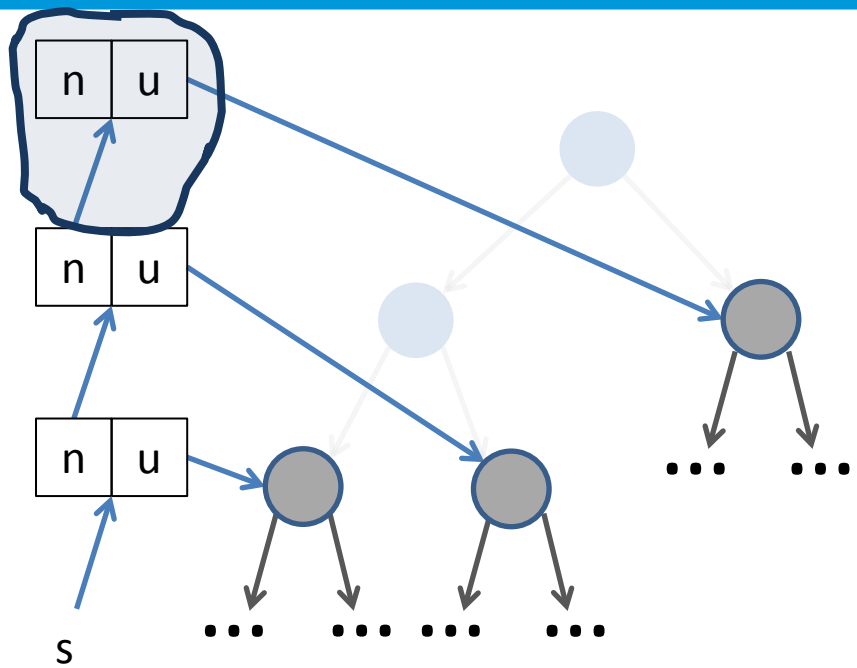
Describe heap layout with formulae



$$s \rightarrow [u: u_1', n: s_1'] \;\; \wedge \;\; s_1' \rightarrow [u: u_2', n: s_2']$$

Classical first order logic

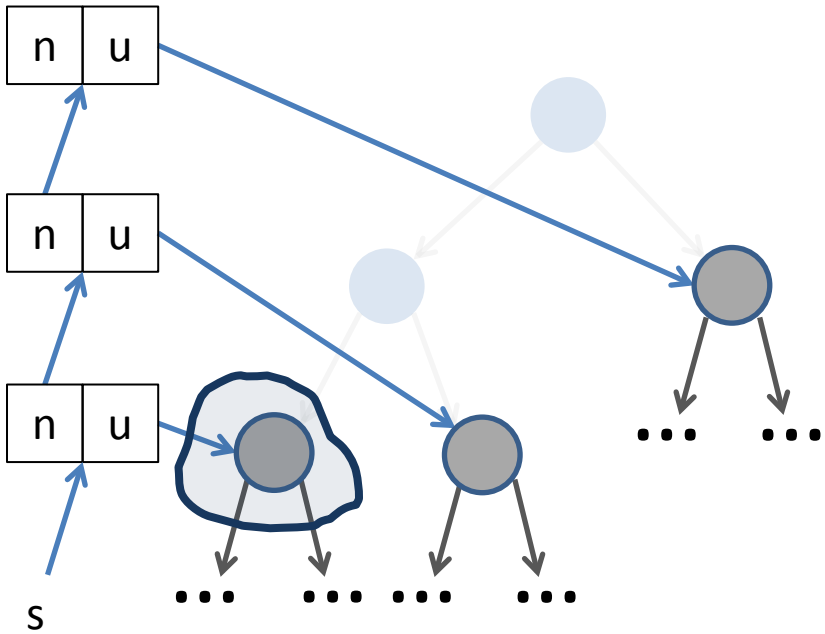
Describe heap layout with formulae



$$s \rightarrow [u: u_1', n: s_1'] \ \wedge \ s_1' \rightarrow [u: u_2', n: s_2'] \ \wedge \ s_2' \rightarrow [u: u_3', n: 0]$$

Classical first order logic

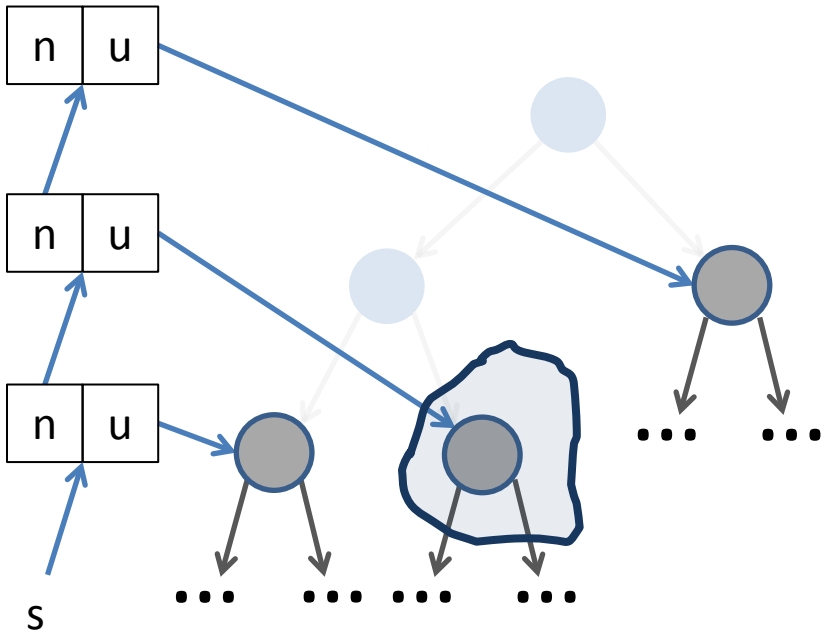
Describe heap layout with formulae



$$s \rightarrow [u: u_1', n: s_1'] \;\wedge\; s_1' \rightarrow [u: u_2', n: s_2'] \;\wedge\; s_2' \rightarrow [u: u_3', n: 0]$$
$$\wedge\; u_1' \rightarrow [l: u_4', r: u_5']$$

Classical first order logic

Describe heap layout with formulae

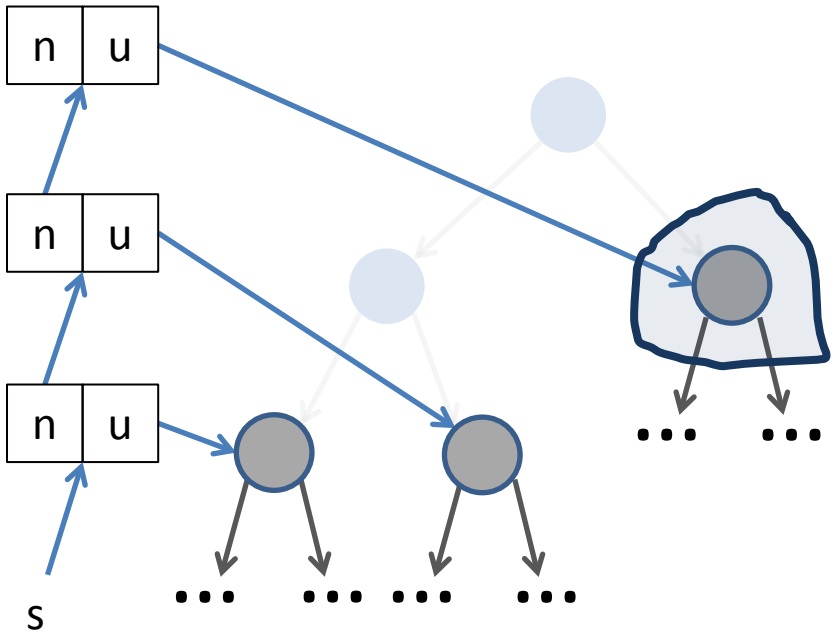


$$s \rightarrow [u: u_1', n: s_1'] \;\wedge\; s_1' \rightarrow [u: u_2', n: s_2'] \;\wedge\; s_2' \rightarrow [u: u_3', n: 0]$$

$$\wedge\; u_1' \rightarrow [l: u_4', r: u_5'] \;\wedge\; u_3' \rightarrow [l: u_8', r: u_9']$$

Classical first order logic

Describe heap layout with formulae

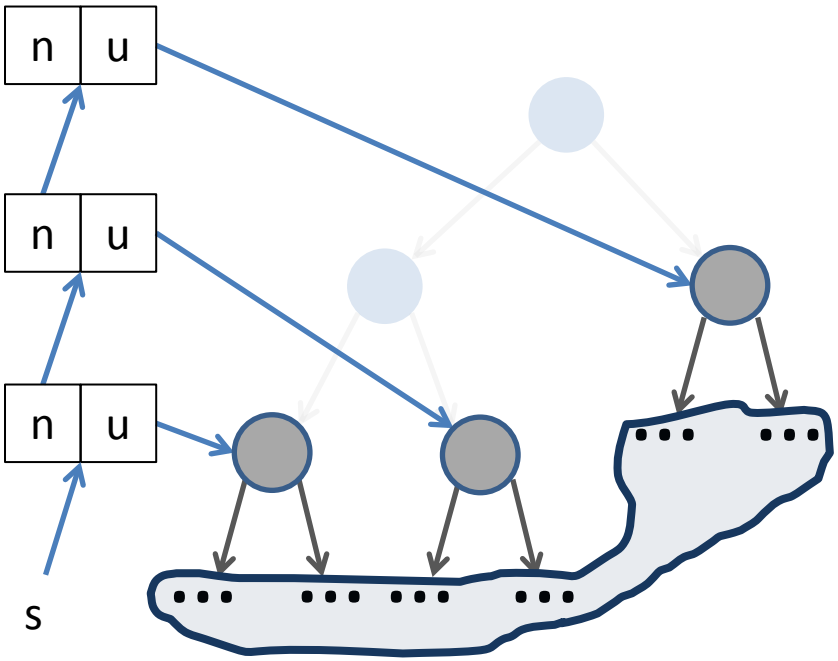


$$s \rightarrow [u: u_1', n: s_1'] \;\wedge\; s_1' \rightarrow [u: u_2', n: s_2'] \;\wedge\; s_2' \rightarrow [u: u_3', n: 0]$$

$$\wedge\; u_1' \rightarrow [l: u_4', r: u_5'] \;\wedge\; u_3' \rightarrow [l: u_8', r: u_9'] \;\wedge\; u_2' \rightarrow [l: u_6', r: u_7']$$

Classical first order logic

Describe heap layout with formulae



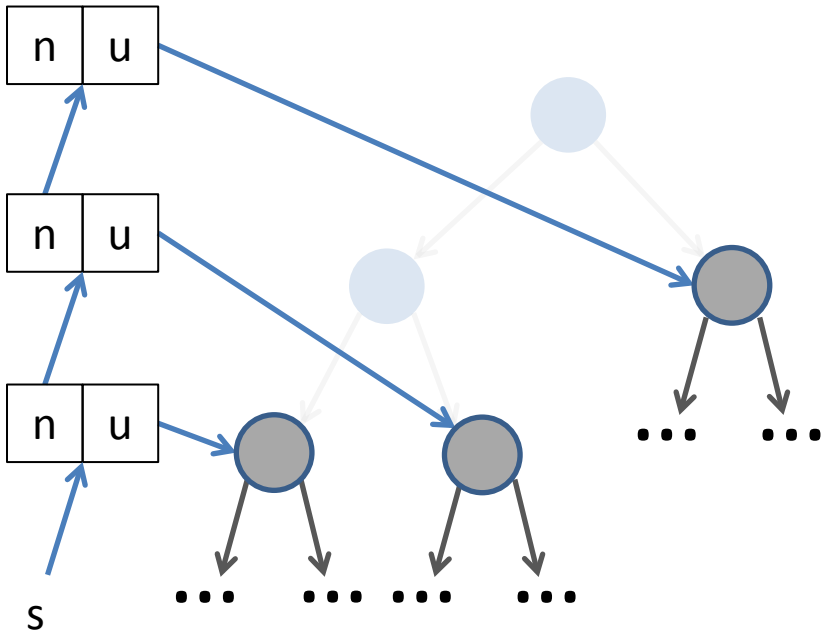
$$s \rightarrow [u: u_1', n: s_1'] \;\wedge\; s_1' \rightarrow [u: u_2', n: s_2'] \;\wedge\; s_2' \rightarrow [u: u_3', n: 0]$$

$$\wedge\; u_1' \rightarrow [l: u_4', r: u_5'] \;\wedge\; u_3' \rightarrow [l: u_8', r: u_9'] \;\wedge\; u_2' \rightarrow [l: u_6', r: u_7']$$

$$\wedge\; \ldots$$

Classical first order logic

Describe heap layout with formulae



Conjunction '∧' does not rule out aliasing!

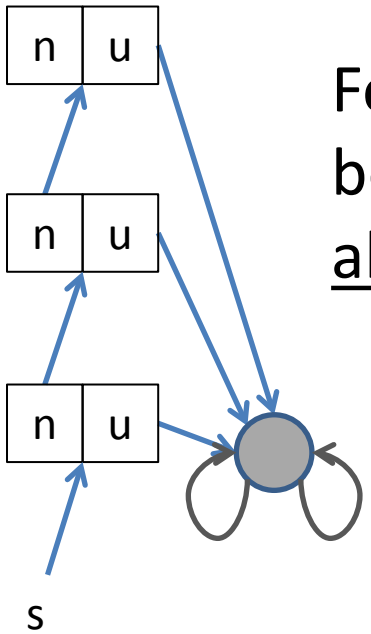
$$s \rightarrow [u: u_1', n: s_1'] \wedge s_1' \rightarrow [u: u_2', n: s_2'] \wedge s_2' \rightarrow [u: u_3', n: 0]$$

$$\wedge\, u_1' \rightarrow [l: u_4', r: u_5'] \wedge u_3' \rightarrow [l: u_8', r: u_9'] \wedge u_2' \rightarrow [l: u_6', r: u_7']$$

$$\wedge \ldots$$

Classical first order logic

Describe heap layout with formulae



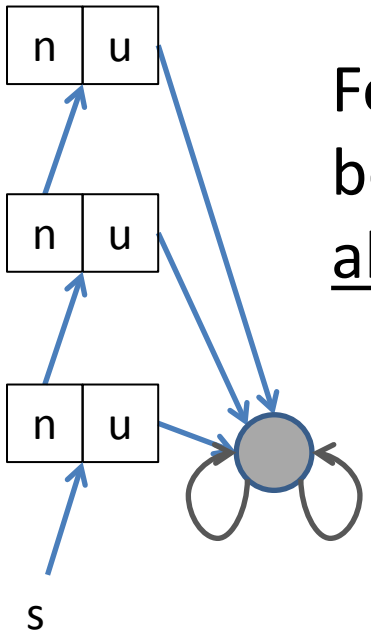
Formula below can <u>also</u> mean this

Conjunction '∧' does not rule out aliasing!

$$s \rightarrow [u: u_1', n: s_1'] \;\wedge\; s_1' \rightarrow [u: u_2', n: s_2'] \;\wedge\; s_2' \rightarrow [u: u_3', n: 0]$$
$$\wedge\; u_1' \rightarrow [l: u_4', r: u_5'] \;\wedge\; u_3' \rightarrow [l: u_8', r: u_9'] \;\wedge\; u_2' \rightarrow [l: u_6', r: u_7']$$
$$\wedge\; \ldots$$

Classical first order logic

Describe heap layout with formulae



Formula below can <u>also</u> mean this

Conjunction

All u-pointers alias

$$u'_1 = u'_2 = u'_3 = u'_4 = u'_5 = u'_6 = u'_7 = u'_8 = u'_9 = \cdots$$

aliasing!

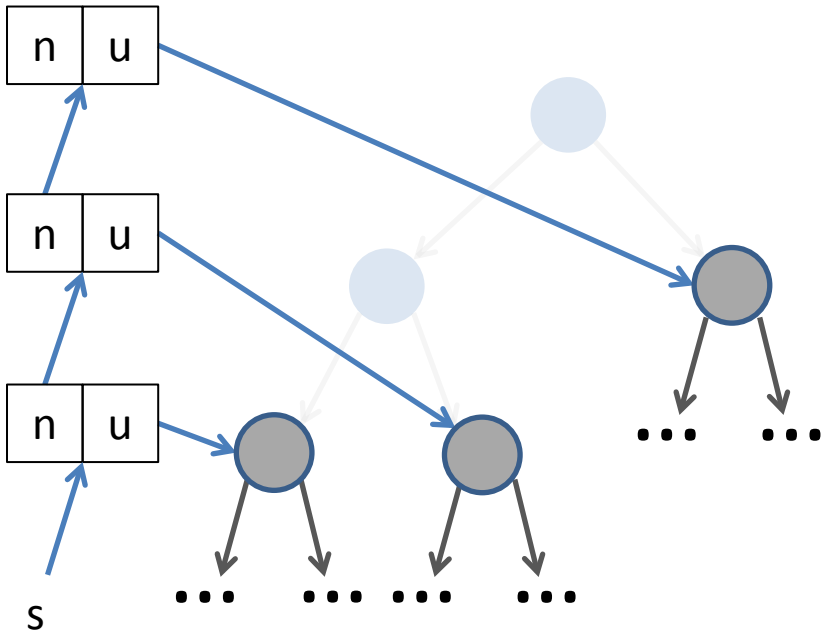
$$s \rightarrow [u: u'_1, n: s'_1] \;\wedge\; s'_1 \rightarrow [u: u'_2, n: s'_2] \;\wedge\; s'_2 \rightarrow [u: u'_3, n: 0]$$

$$\wedge\; u'_1 \rightarrow [l: u'_4, r: u'_5] \;\wedge\; u'_3 \rightarrow [l: u'_8, r: u'_9] \;\wedge\; u'_2 \rightarrow [l: u'_6, r: u'_7]$$

$$\wedge\; \ldots$$

Classical first order logic

Describe heap layout with formulae

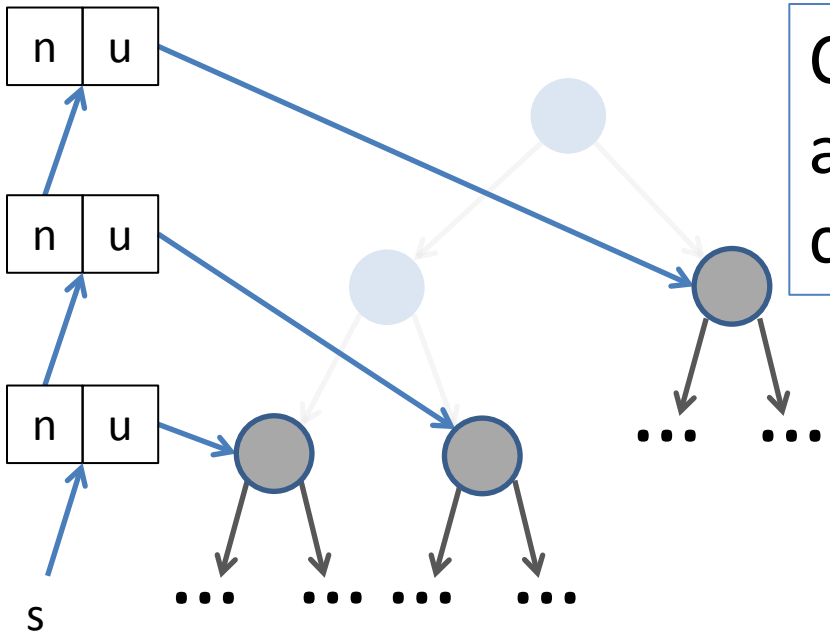


Conjunction '∧' does not rule out aliasing!

$$s \rightarrow [u: u_1', n: s_1'] \;\wedge\; s_1' \rightarrow [u: u_2', n: s_2'] \;\wedge\; s_2' \rightarrow [u: u_3', n: 0]$$
$$\wedge\; u_1' \rightarrow [l: u_4', r: u_5'] \;\wedge\; u_3' \rightarrow [l: u_8', r: u_9'] \;\wedge\; u_2' \rightarrow [l: u_6', r: u_7']$$
$$\wedge\; \ldots$$

Classical first order logic

Describe heap layout with formulae



Could add loads of constraints

Conjunction '∧' does not rule out aliasing!

$$s \rightarrow [u{:}\,u_1', n{:}\,s_1'] \;\wedge\; s_1' \rightarrow [u{:}\,u_2', n{:}\,s_2'] \;\wedge\; s_2' \rightarrow [u{:}\,u_3', n{:}\,0]$$

$$\wedge\; u_1' \rightarrow [l{:}\,u_4', r{:}\,u_5'] \;\wedge\; u_3' \rightarrow [l{:}\,u_8', r{:}\,u_9'] \;\wedge\; u_2' \rightarrow [l{:}\,u_6', r{:}\,u_7']$$

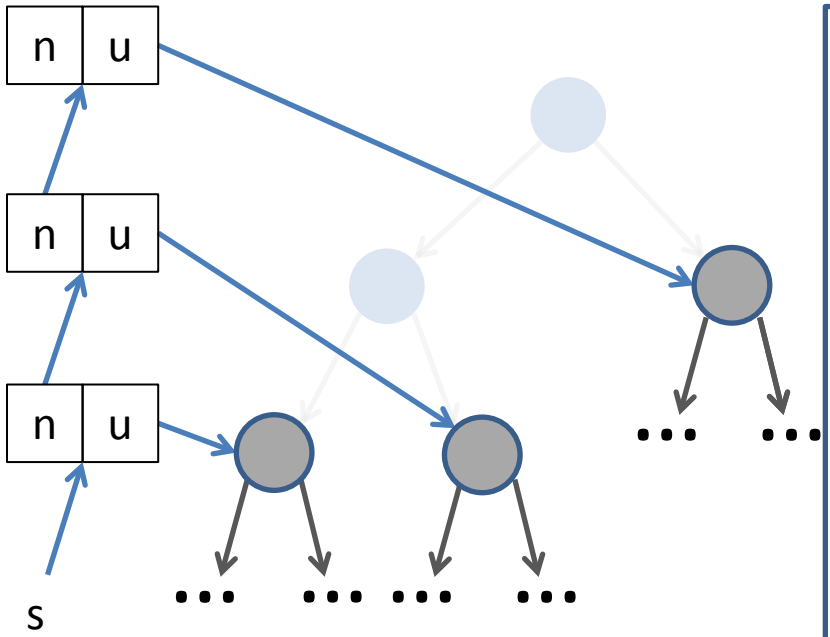
$$\wedge\; \ldots \wedge u_1' \neq u_2' \wedge u_1 \neq u_3' \wedge u_1' \neq u_4' \wedge u_1' \neq u_5' \wedge u_1' \neq u_6' \wedge u_1' \neq u_7'$$

$$\wedge\, u_1' \neq u_8' \wedge u_1 \neq u_9' \wedge u_3' \neq u_4' \wedge u_3' \neq u_5' \wedge u_3' \neq u_6' \wedge u_3' \neq u_7'$$

$$\wedge\, u_3' \neq u_8' \wedge u_2 \neq u_3' \wedge u_2' \neq u_4' \wedge u_2' \neq u_5' \wedge u_2' \neq u_6' \wedge \ldots$$

Separation logic

Describe heap layout with formulae



O'Hearn, Reynolds, Ishtiaq, Yang:

Separating conjunction '*'

$$s \rightarrow [u{:}\,u_1', n{:}\,s_1'] \;\wedge\; s_1' \rightarrow [u{:}\,u_2', n{:}\,s_2'] \;\wedge\; s_2' \rightarrow [u{:}\,u_3', n{:}\,0]$$

$$\wedge\; u_1' \rightarrow [l{:}\,u_4', r{:}\,u_5'] \;\wedge\; u_3' \rightarrow [l{:}\,u_8', r{:}\,u_9'] \;\wedge\; u_2' \rightarrow [l{:}\,u_6', r{:}\,u_7']$$

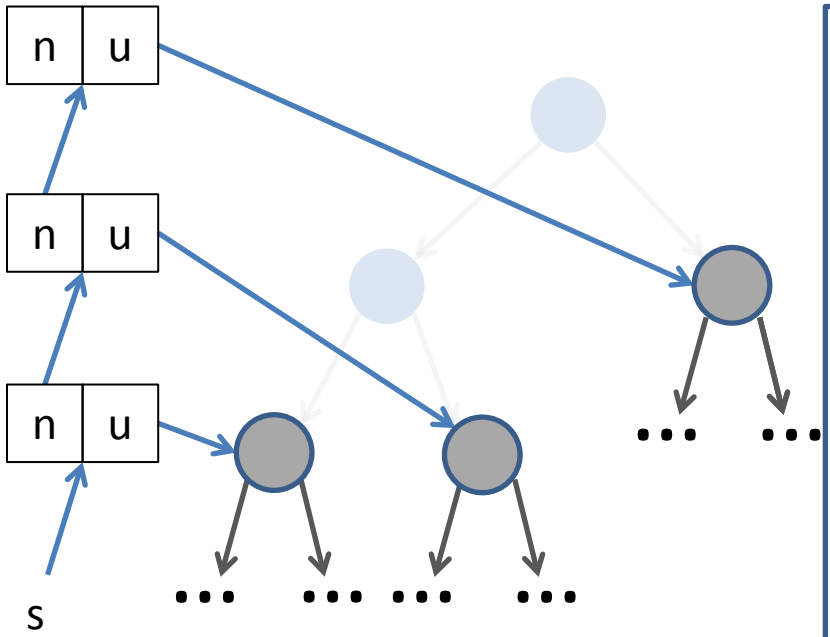
$$\wedge\; \ldots \wedge u_1' \neq u_2' \wedge u_1 \neq u_3' \wedge u_1' \neq u_4' \wedge u_1' \neq u_5' \wedge u_1' \neq u_6' \wedge u_1' \neq u_7'$$

$$\wedge\, u_1' \neq u_8' \wedge u_1 \neq u_9' \wedge u_3' \neq u_4' \wedge u_3' \neq u_5' \wedge u_3' \neq u_6' \wedge u_3' \neq u_7'$$

$$\wedge\, u_3' \neq u_8' \wedge u_2 \neq u_3' \wedge u_2' \neq u_4' \wedge u_2' \neq u_5' \wedge u_2' \neq u_6' \wedge \ldots$$

Separation logic

Describe heap layout with formulae



O'Hearn, Reynolds, Ishtiaq, Yang:

Separating conjunction '∗' rules out aliasing!

$$s \rightarrow [u{:}\,u_1', n{:}\,s_1'] \;\wedge\; s_1' \rightarrow [u{:}\,u_2', n{:}\,s_2'] \;\wedge\; s_2' \rightarrow [u{:}\,u_3', n{:}\,0]$$

$$\wedge\; u_1' \rightarrow [l{:}\,u_4', r{:}\,u_5'] \;\wedge\; u_3' \rightarrow [l{:}\,u_8', r{:}\,u_9'] \;\wedge\; u_2' \rightarrow [l{:}\,u_6', r{:}\,u_7']$$

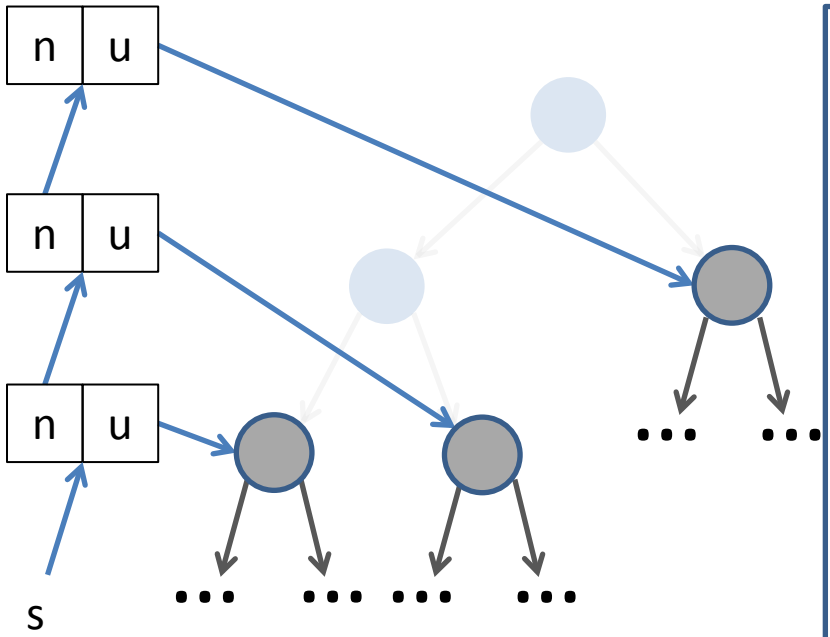
$$\wedge\; \ldots \wedge u_1' \neq u_2' \wedge u_1 \neq u_3' \wedge u_1' \neq u_4' \wedge u_1' \neq u_5' \wedge u_1' \neq u_6' \wedge u_1' \neq u_7'$$

$$\wedge\, u_1' \neq u_8' \wedge u_1 \neq u_9' \wedge u_3' \neq u_4' \wedge u_3' \neq u_5' \wedge u_3' \neq u_6' \wedge u_3' \neq u_7'$$

$$\wedge\, u_3' \neq u_8' \wedge u_2 \neq u_3' \wedge u_2' \neq u_4' \wedge u_2' \neq u_5' \wedge u_2' \neq u_6' \wedge \ldots$$

Separation logic

Describe heap layout with formulae



O'Hearn, Reynolds, Ishtiaq, Yang:

Separating conjunction '∗' rules out aliasing!

$$s \rightarrow [u{:}\,u_1', n{:}\,s_1'] \;*\; s_1' \rightarrow [u{:}\,u_2', n{:}\,s_2'] \;*\; s_2' \rightarrow [u{:}\,u_3', n{:}\,0]$$

$$*\; u_1' \rightarrow [l{:}\,u_4', r{:}\,u_5'] \;*\; u_3' \rightarrow [l{:}\,u_8', r{:}\,u_9'] \;*\; u_2' \rightarrow [l{:}\,u_6', r{:}\,u_7']$$

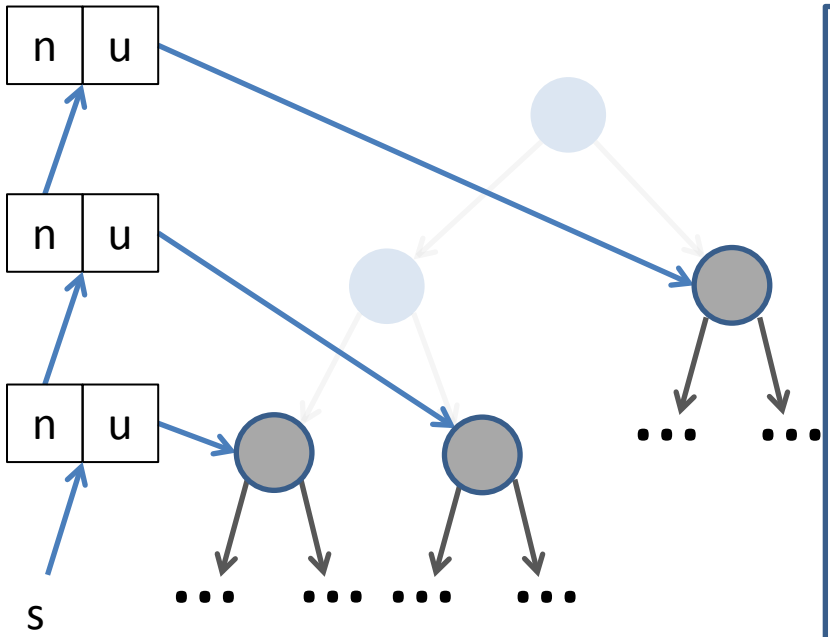
$$\wedge\; \ldots \wedge u_1' \neq u_2' \wedge u_1 \neq u_3' \wedge u_1' \neq u_4' \wedge u_1' \neq u_5' \wedge u_1' \neq u_6' \wedge u_1' \neq u_7'$$

$$\wedge\, u_1' \neq u_8' \wedge u_1 \neq u_9' \wedge u_3' \neq u_4' \wedge u_3' \neq u_5' \wedge u_3' \neq u_6' \wedge u_3' \neq u_7'$$

$$\wedge\, u_3' \neq u_8' \wedge u_2 \neq u_3' \wedge u_2' \neq u_4' \wedge u_2' \neq u_5' \wedge u_2' \neq u_6' \wedge \ldots$$

Separation logic

Describe heap layout with formulae



O'Hearn, Reynolds, Ishtiaq, Yang:

Separating conjunction '$*$' rules out aliasing!

$$s \rightarrow [u{:}\,u_1', n{:}\,s_1'] \;*\; s_1' \rightarrow [u{:}\,u_2', n{:}\,s_2'] \;*\; s_2' \rightarrow [u{:}\,u_3', n{:}\,0]$$

$$*\; u_1' \rightarrow [l{:}\,u_4', r{:}\,u_5'] \;*\; u_3' \rightarrow [l{:}\,u_8', r{:}\,u_9'] \;*\; u_2' \rightarrow [l{:}\,u_6', r{:}\,u_7']$$

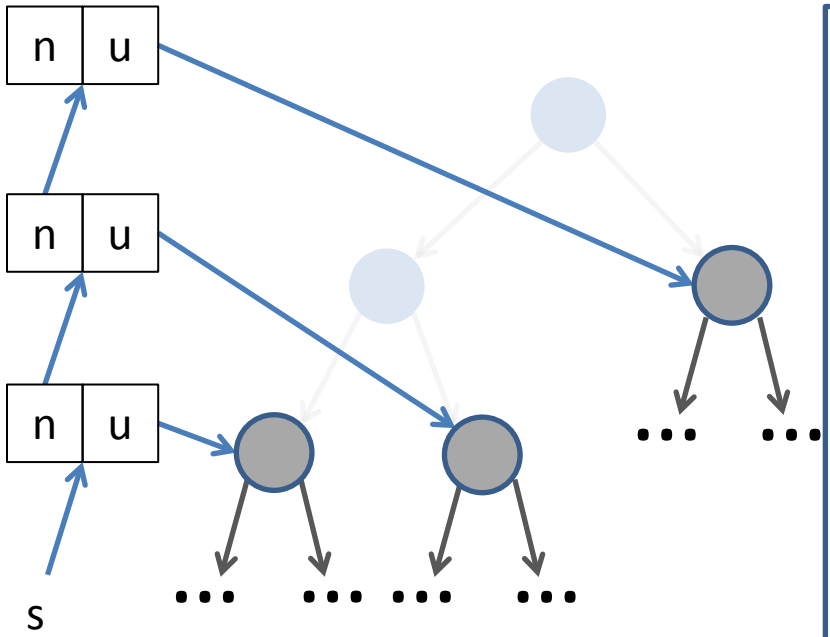
$$\wedge\; \ldots \wedge u_1' \neq u_2' \wedge u_1 \neq u_3' \wedge u_1' \neq u_4' \wedge u_1' \neq u_5' \wedge u_1' \neq u_6' \wedge u_1' \neq u_7'$$

$$\wedge\, u_1' \neq u_8' \wedge u_1 \neq u_9' \wedge u_3' \neq u_4' \wedge u_3' \neq u_5' \wedge u_3' \neq u_6' \wedge u_3' \neq u_7'$$

$$\wedge\, u_3' \neq u_8' \wedge u_2 \neq u_3' \wedge u_2' \neq u_4' \wedge u_2' \neq u_5' \wedge u_2' \neq u_6' \wedge \ldots$$

Separation logic

Describe heap layout with formulae



O'Hearn, Reynolds, Ishtiaq, Yang:

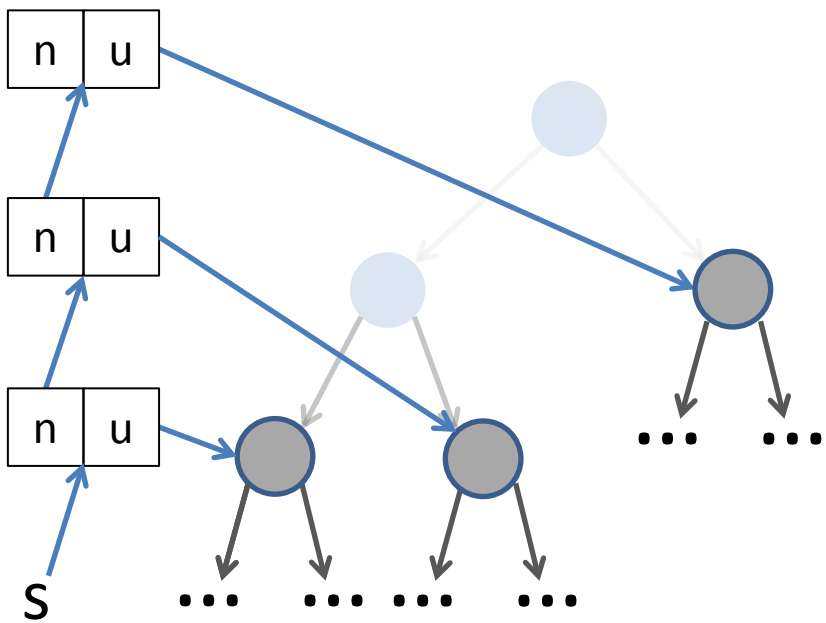
Separating conjunction '∗' rules out aliasing!

$$s \rightarrow [u: u_1', n: s_1'] \ * \ s_1' \rightarrow [u: u_2', n: s_2'] \ * \ s_2' \rightarrow [u: u_3', n: 0]$$

$$* \ u_1' \rightarrow [l: u_4', r: u_5'] \ * \ u_3' \rightarrow [l: u_8', r: u_9'] \ * \ u_2' \rightarrow [l: u_6', r: u_7']$$

- Tractable heap analysis – very popular in SW verification
- We use it to prove disjointness of heap regions

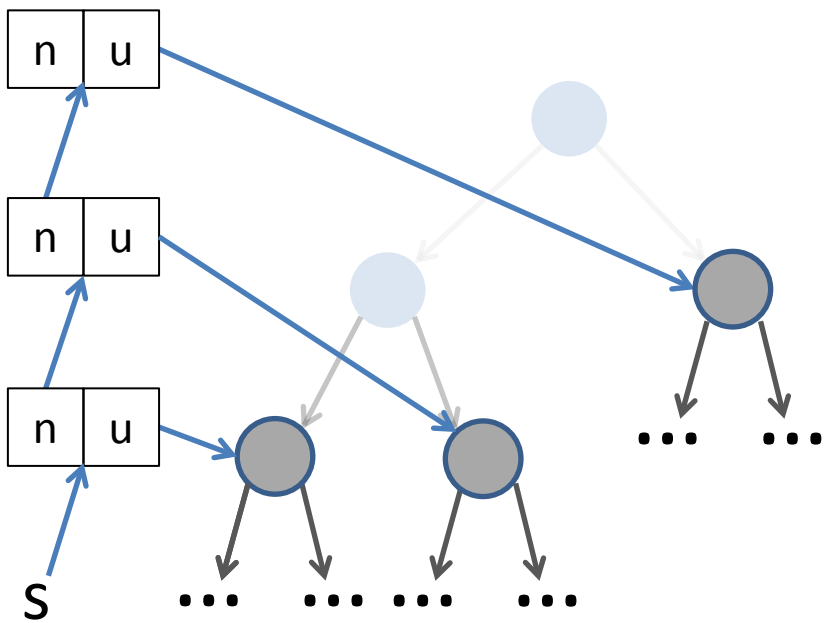
Back to our partitioning task



$$s \rightarrow [u: u'_1, n: s'_1] \quad * \quad s'_1 \rightarrow [u: u'_2, n: s'_2] \quad * \quad s'_2 \rightarrow [u: u'_3, n: 0]$$

$$* \quad u'_1 \rightarrow [l: u'_4, r: u'_5] \quad * \quad u'_3 \rightarrow [l: u'_8, r: u'_9] \quad * \quad u'_2 \rightarrow [l: u'_6, r: u'_7]$$

Back to our partitioning task

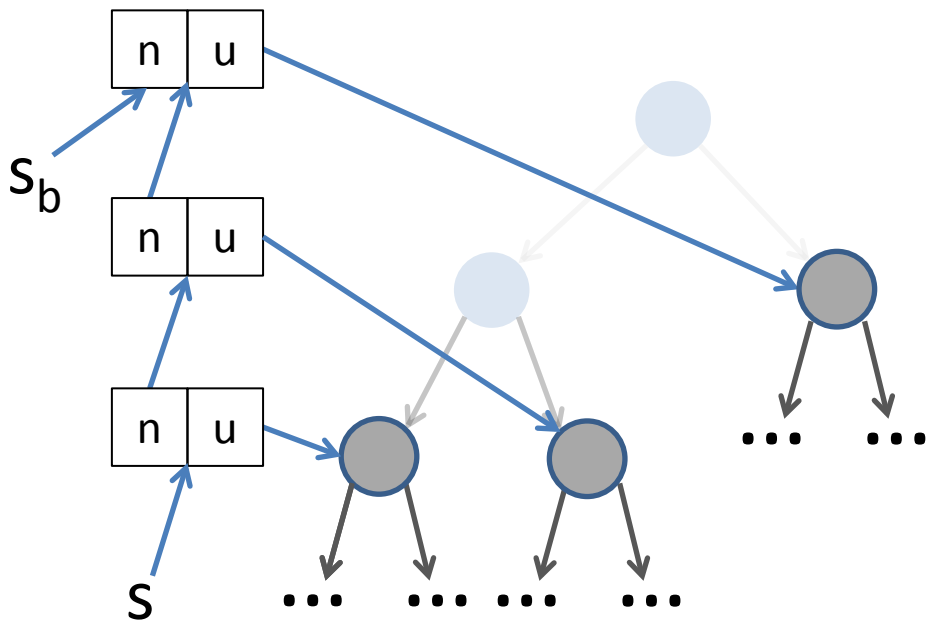


- Partitioning the heap = partitioning the formula describing it

$$s \rightarrow [u: u_1', n: s_1'] \ * \ s_1' \rightarrow [u: u_2', n: s_2'] \ * \ s_2' \rightarrow [u: u_3', n: 0]$$

$$* \ u_1' \rightarrow [l: u_4', r: u_5'] \ * \ u_3' \rightarrow [l: u_8', r: u_9'] \ * \ u_2' \rightarrow [l: u_6', r: u_7']$$

Back to our partitioning task

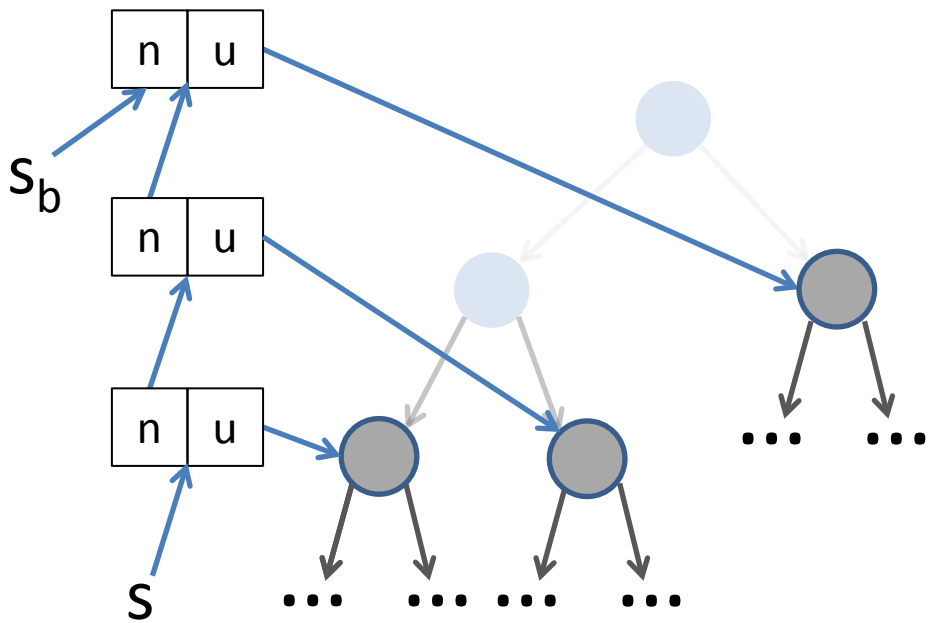


- Partitioning the heap = partitioning the formula describing it
- Add a second 'hook' into the data structure

$$s \rightarrow [u: u_1', n: s_1'] \ * \ s_1' \rightarrow [u: u_2', n: s_2'] \ * \ s_2' \rightarrow [u: u_3', n: 0]$$

$$* \ u_1' \rightarrow [l: u_4', r: u_5'] \ * \ u_3' \rightarrow [l: u_8', r: u_9'] \ * \ u_2' \rightarrow [l: u_6', r: u_7']$$

Back to our partitioning task

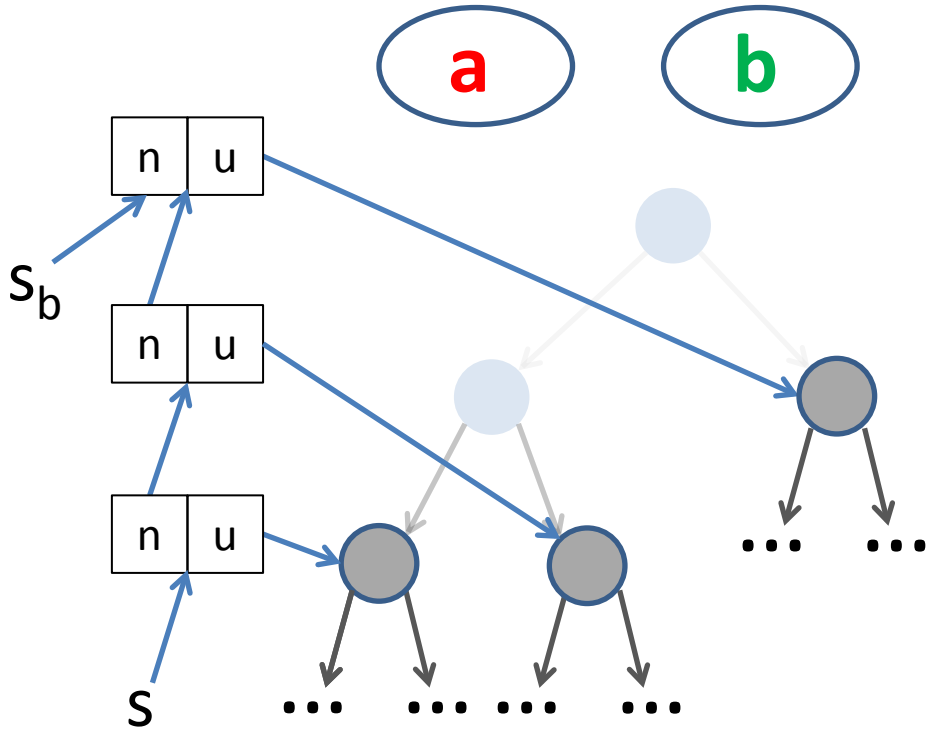


- Partitioning the heap = partitioning the formula describing it
- Add a second 'hook' into the data structure
- 'Symbolically' step through loop iterations

$$s \rightarrow [u: u_1', n: s_1'] \ * \ s_1' \rightarrow [u: u_2', n: s_2'] \ * \ s_2' \rightarrow [u: u_3', n: 0]$$

$$* \ u_1' \rightarrow [l: u_4', r: u_5'] \ * \ u_3' \rightarrow [l: u_8', r: u_9'] \ * \ u_2' \rightarrow [l: u_6', r: u_7']$$

Back to our partitioning task

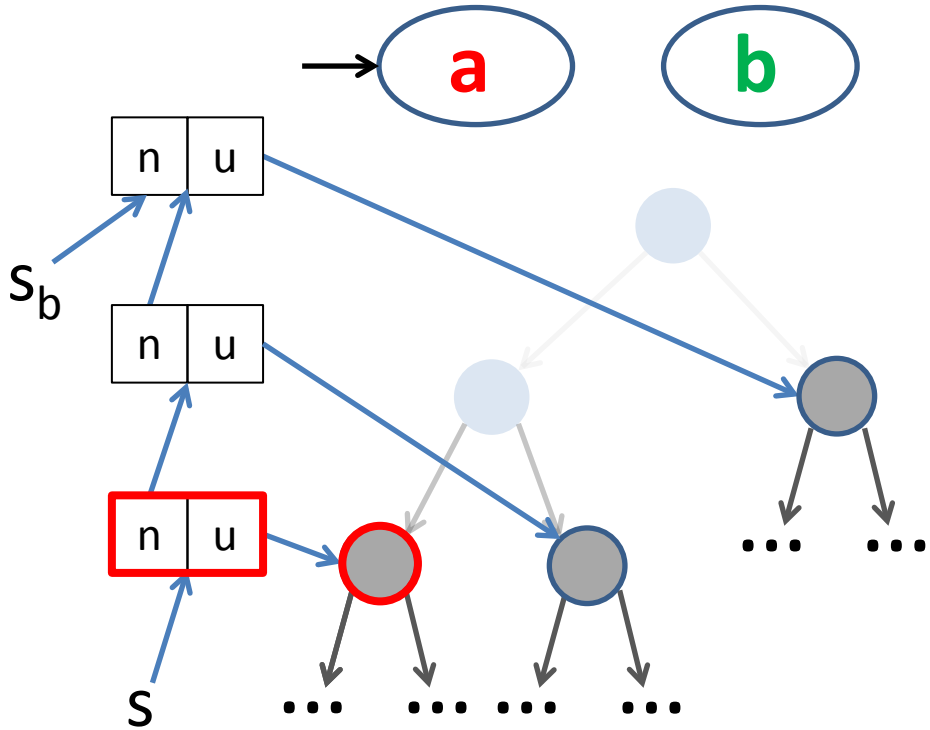


- Partitioning the heap = partitioning the formula describing it
- Add a second 'hook' into the data structure
- 'Symbolically' step through loop iterations

$$s \rightarrow [u: u_1', n: s_1'] \ * \ s_1' \rightarrow [u: u_2', n: s_2'] \ * \ s_2' \rightarrow [u: u_3', n: 0]$$

$$* \ u_1' \rightarrow [l: u_4', r: u_5'] \ * \ u_3' \rightarrow [l: u_8', r: u_9'] \ * \ u_2' \rightarrow [l: u_6', r: u_7']$$

Back to our partitioning task

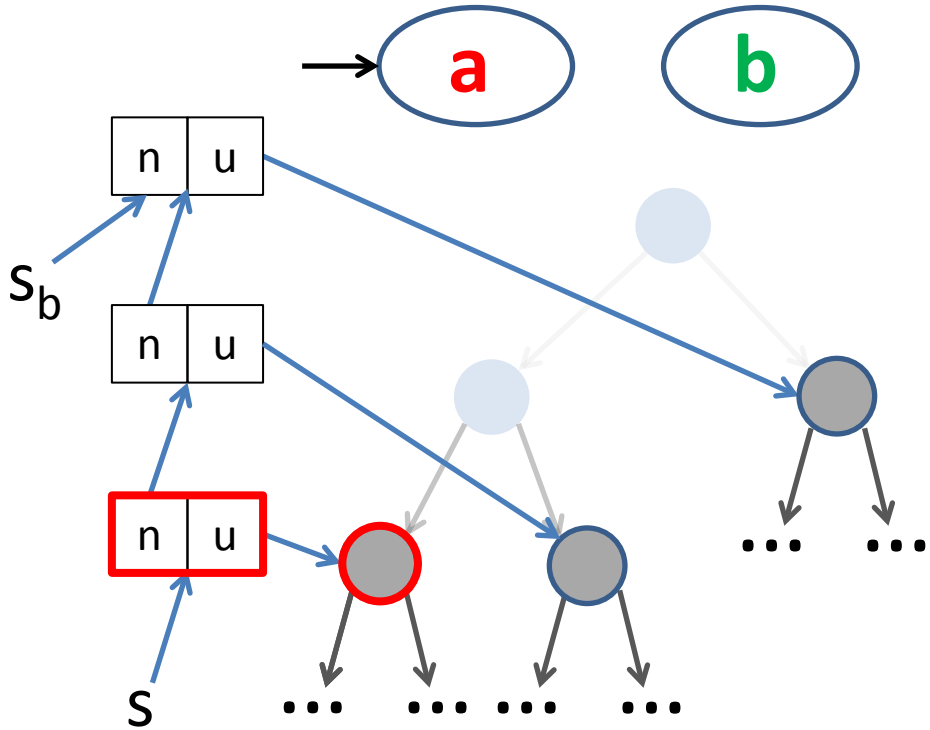


- Partitioning the heap = partitioning the formula describing it
- Add a second 'hook' into the data structure
- 'Symbolically' step through loop iterations

$$s \rightarrow [u: u_1', n: s_1'] \ * \ s_1' \rightarrow [u: u_2', n: s_2'] \ * \ s_2' \rightarrow [u: u_3', n: 0]$$

$$* \ u_1' \rightarrow [l: u_4', r: u_5'] \ * \ u_3' \rightarrow [l: u_8', r: u_9'] \ * \ u_2' \rightarrow [l: u_6', r: u_7']$$

Back to our partitioning task

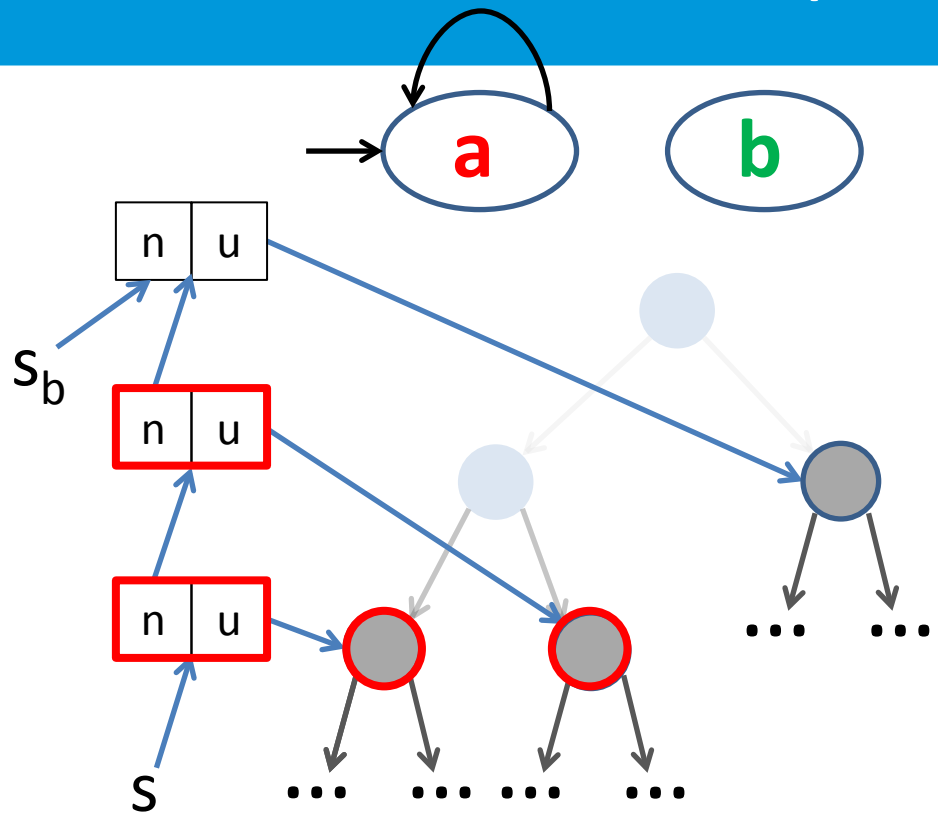


- Partitioning the heap = partitioning the formula describing it
- Add a second 'hook' into the data structure
- 'Symbolically' step through loop iterations
- Attach labels to heaplets

$$s \rightarrow [u{:}\,u_1', n{:}\,s_1']_{\mathbf{a}} * s_1' \rightarrow [u{:}\,u_2', n{:}\,s_2'] * s_2' \rightarrow [u{:}\,u_3', n{:}\,0]$$

$$* \; u_1' \rightarrow [l{:}\,u_4', r{:}\,u_5']_{\mathbf{a}} * u_3' \rightarrow [l{:}\,u_8', r{:}\,u_9'] * u_2' \rightarrow [l{:}\,u_6', r{:}\,u_7']$$

Back to our partitioning task

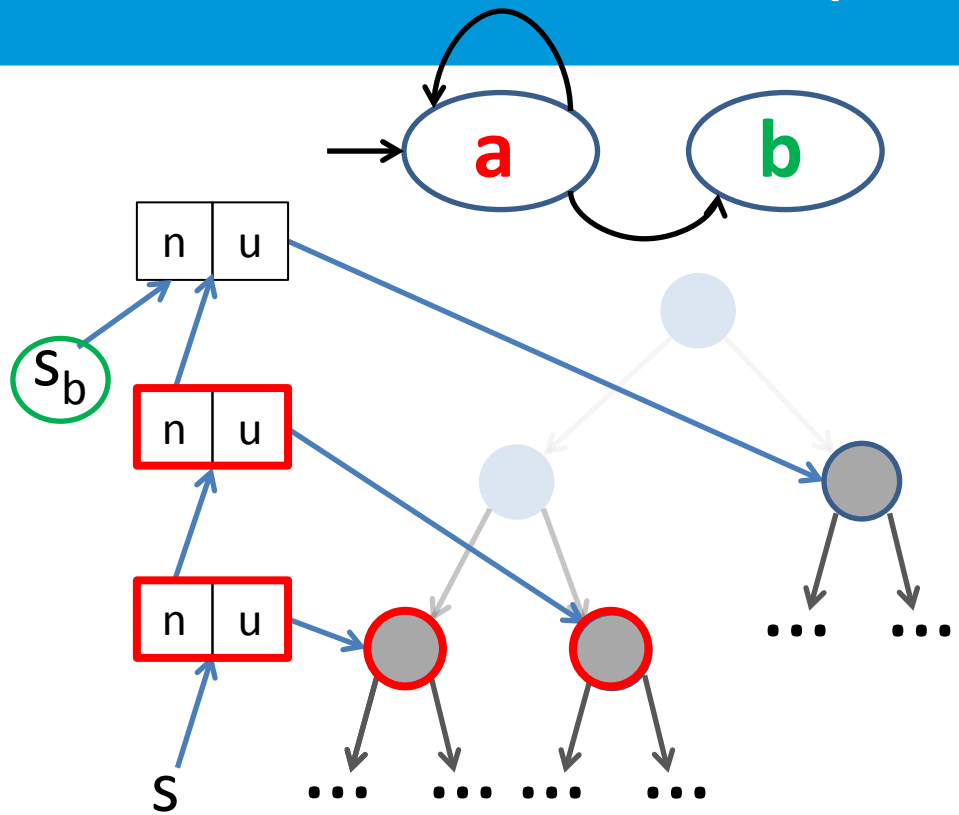


- Partitioning the heap = partitioning the formula describing it
- Add a second 'hook' into the data structure
- 'Symbolically' step through loop iterations
- Attach labels to heaplets

$$s \to [u{:}\,u'_1, n{:}\,s'_1]_{\mathbf{a}} * s'_1 \to [u{:}\,u'_2, n{:}\,s'_2]_{\mathbf{a}} * s'_2 \to [u{:}\,u'_3, n{:}\,0]$$

$$* \; u'_1 \to [l{:}\,u'_4, r{:}\,u'_5]_{\mathbf{a}} * u'_3 \to [l{:}\,u'_8, r{:}\,u'_9]_{\mathbf{a}} * u'_2 \to [l{:}\,u'_6, r{:}\,u'_7]$$

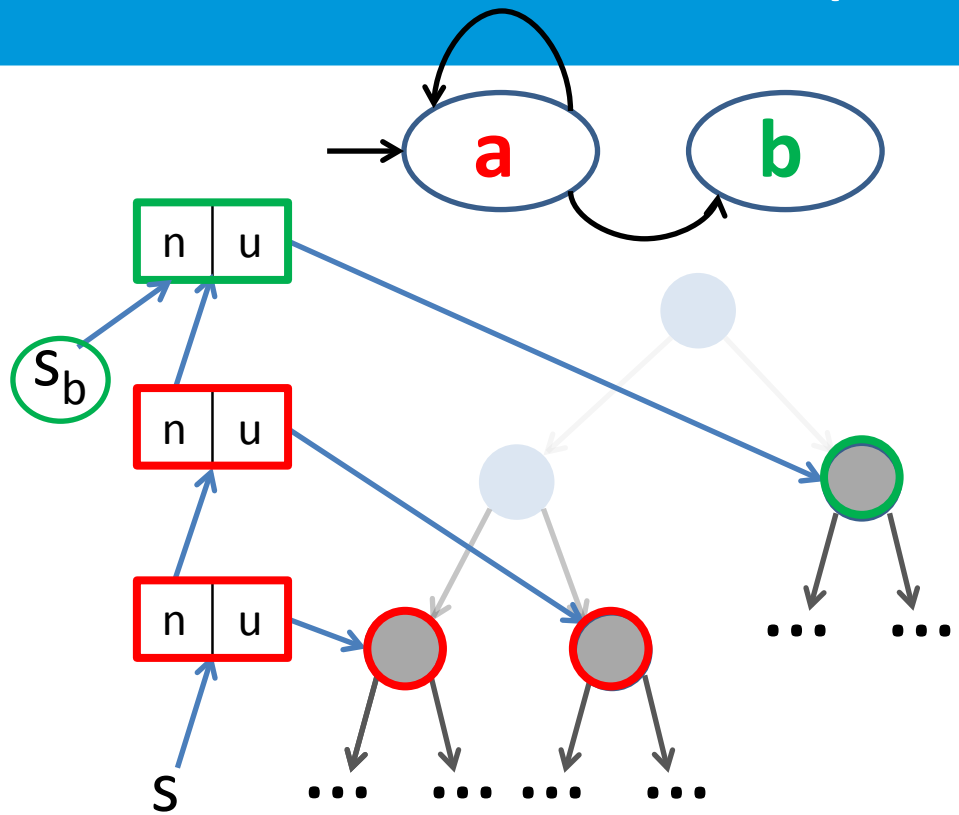
Back to our partitioning task



- Partitioning the heap = partitioning the formula describing it
- Add a second 'hook' into the data structure
- 'Symbolically' step through loop iterations
- Attach labels to heaplets

$$s \to [u: u_1', n: s_1']_{\mathbf{a}} * s_1' \to [u: u_2', n: s_2']_{\mathbf{a}} * s_2' \to [u: u_3', n: 0]$$
$$* \; u_1' \to [l: u_4', r: u_5']_{\mathbf{a}} * u_3' \to [l: u_8', r: u_9']_{\mathbf{a}} * u_2' \to [l: u_6', r: u_7']$$

Back to our partitioning task

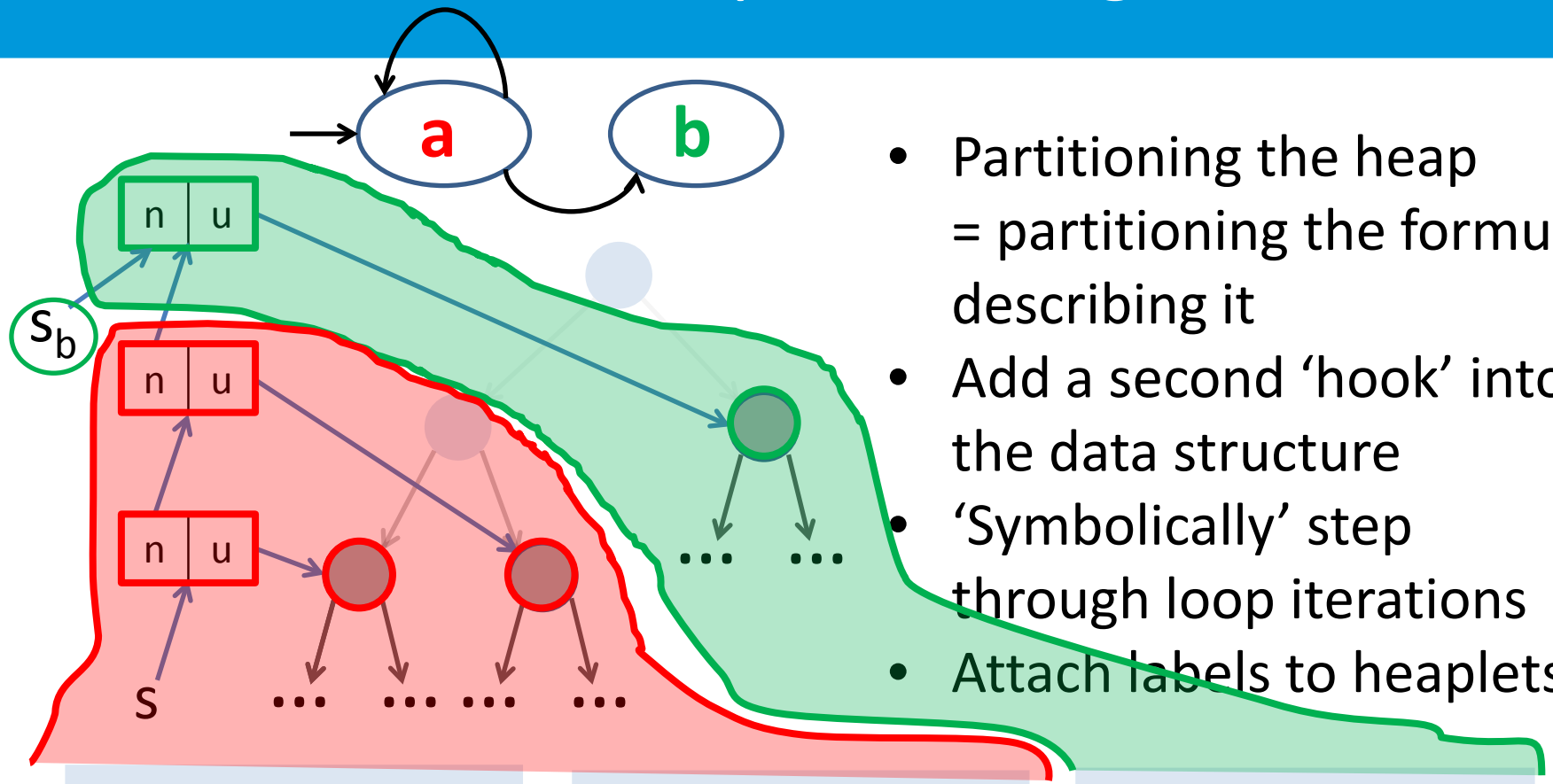


- Partitioning the heap = partitioning the formula describing it
- Add a second 'hook' into the data structure
- 'Symbolically' step through loop iterations
- Attach labels to heaplets

$$s \to [u: u_1', n: s_1']_{\mathbf{a}} * s_1' \to [u: u_2', n: s_2']_{\mathbf{a}} * s_2' \to [u: u_3', n: 0]_{\mathbf{b}}$$

$$* \; u_1' \to [l: u_4', r: u_5']_{\mathbf{a}} * u_3' \to [l: u_8', r: u_9']_{\mathbf{a}} * u_2' \to [l: u_6', r: u_7']_{\mathbf{b}}$$

Back to our partitioning task

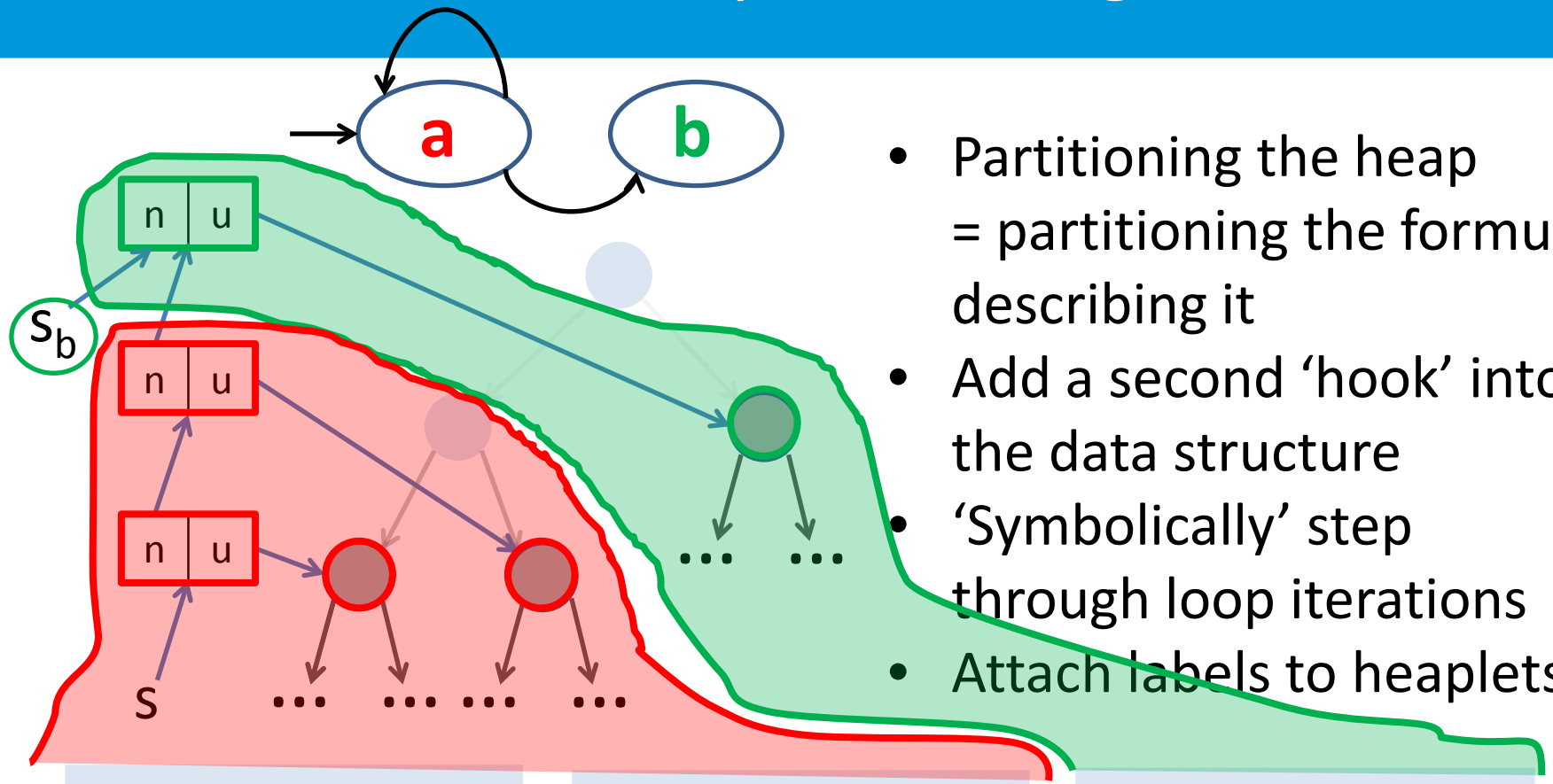


- Partitioning the heap = partitioning the formula describing it
- Add a second 'hook' into the data structure
- 'Symbolically' step through loop iterations
- Attach labels to heaplets

$$s \to [u: u_1', n: s_1']_{\mathbf{a}} * s_1' \to [u: u_2', n: s_2']_{\mathbf{a}} * s_2' \to [u: u_3', n: 0]_{\mathbf{b}}$$

$$* \; u_1' \to [l: u_4', r: u_5']_{\mathbf{a}} * u_3' \to [l: u_8', r: u_9']_{\mathbf{a}} * u_2' \to [l: u_6', r: u_7']_{\mathbf{b}}$$

Back to our partitioning task



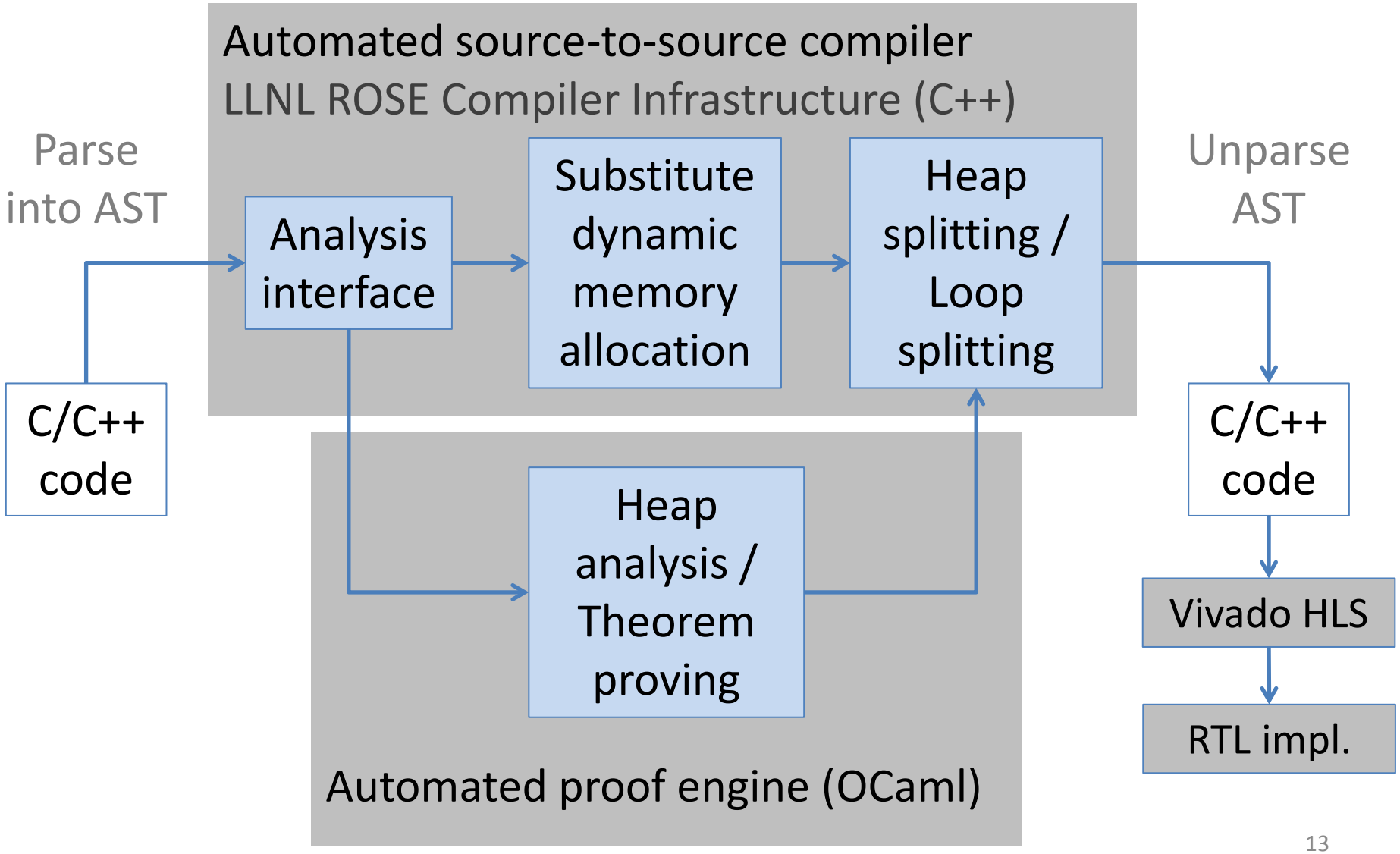
- Partitioning the heap = partitioning the formula describing it
- Add a second 'hook' into the data structure
- 'Symbolically' step through loop iterations
- Attach labels to heaplets

$$s \to [u: u'_1, n: s'_1]_{\mathbf{a}} * s'_1 \to [u: u'_2, n: s'_2]_{\mathbf{a}} * s'_2 \to [u: u'_3, n: 0]_{\mathbf{b}}$$

$$* \; u'_1 \to [l: u'_4, r: u'_5]_{\mathbf{a}} * u'_3 \to [l: u'_8, r: u'_9]_{\mathbf{a}} * u'_2 \to [l: u'_6, r: u'_7]_{\mathbf{b}}$$

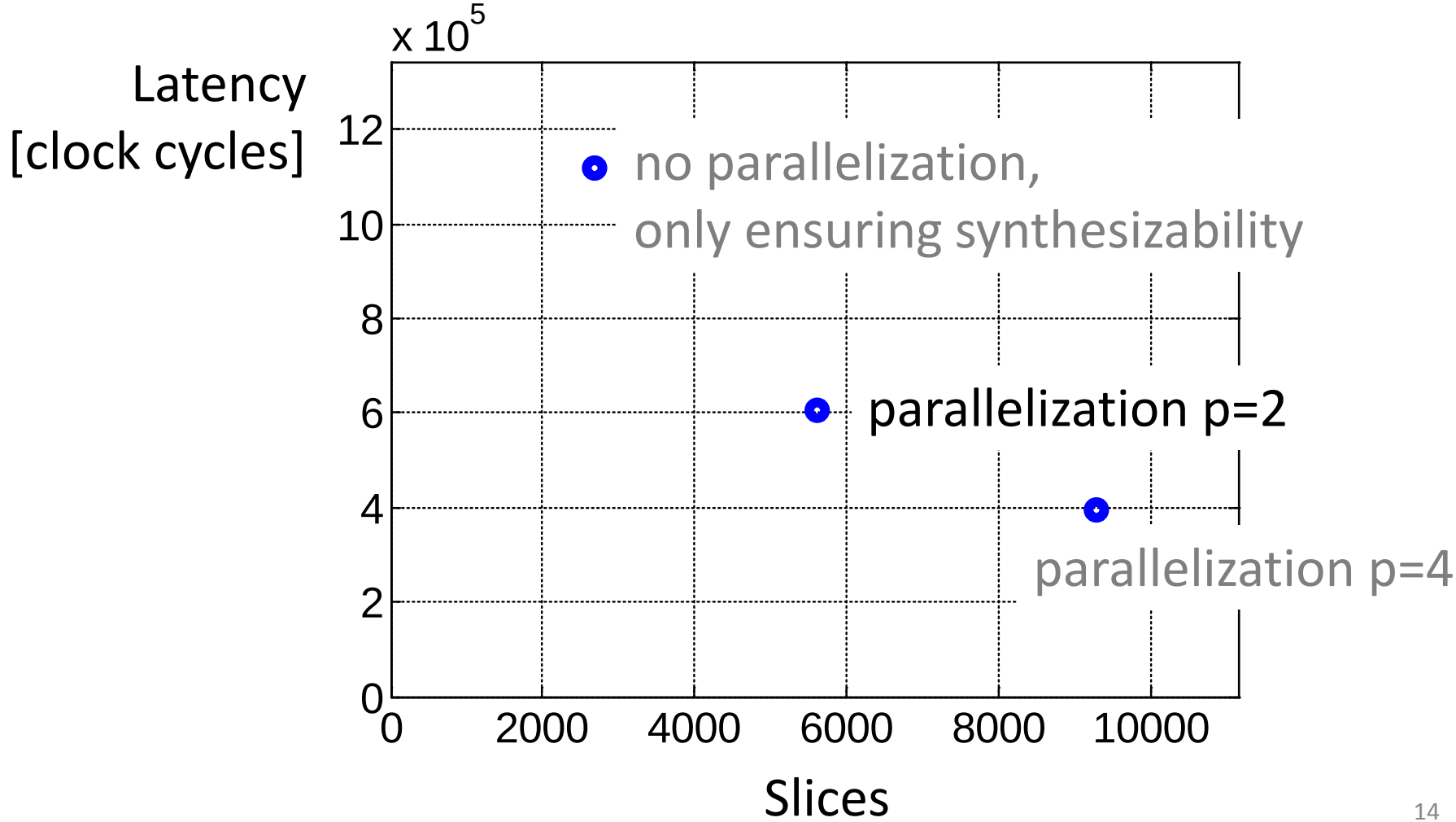
Communication-free parallelism: Never $\ldots_{\mathbf{ab}}$

Implementation



Case Studies (I)

Tree-based *K*-means clustering



Case Studies (II)

		P	Slices	Clock	Cycles	
1	**Merger (linked lists)**					
	Baseline (no par.)	1	574	9.0 ns	21167k	x4
	Autom. Parallelization	4	965	8.7 ns	5483k	
2	**Tree deletion (tree, linked list)**					
	Baseline (no par.)	1	1521	5.2 ns	901k	x2
	Autom. Parallelization	2	4069	6.0 ns	487k	
3	***K*-means (tree, linked list, single heap records)**					
	Baseline (no par.)	1	2694	6.1 ns	1120k	x2
	Autom. Parallelization	2	5618	7.0 ns	606k	

Case Studies (II)

		P	Slices	Clock	Cycles	
1	**Merger (linked lists)**					
	Baseline (no par.)	1	574	9.0 ns	21167k	x4
	Autom. Parallelization	4	965	8.7 ns	5483k	
2	**Tree deletion (tree, linked list)**					
	Baseline (no par.)	1	1521	5.2 ns	901k	x2
	Autom. Parallelization	2	4069	6.0 ns	487k	
3	***K*-means (tree, linked list, single heap records)**					
	Baseline (no par.)	1	2694	6.1 ns	1120k	x2
	Autom. Parallelization	2	5618	7.0 ns	606k	

Manual loop flattening, pipelining, custom bit widths, data streaming directives, data packing, …

Hand-optimized HLS	2	5492	5.5 ns	165k	

Case Studies (II)

		P	Slices	Clock	Cycles	
1	**Merger (linked lists)**					
	Baseline (no par.)	1	574	9.0 ns	21167k	x4
	Autom. Parallelization	4	965	8.7 ns	5483k	
2	**Tree deletion (tree, linked list)**					
	Baseline (no par.)	1	1521	5.2 ns	901k	x2
	Autom. Parallelization	2	4069	6.0 ns	487k	
3	***K*-means (tree, linked list, single heap records)**					
	Baseline (no par.)	1	2694	6.1 ns	1120k	x2
	Autom. Parallelization	2	5618	7.0 ns	606k	
	Manual loop flattening, pipelining, custom bit widths, data streaming directives, data					x3.6
	Hand-optimized HLS	2	5492	5.5 ns	165k	

Conclusion

- Static analysis of heap-manipulating programs
  - Leveraging recent advances in separation logic
  - Distribute heap across on-chip memory banks
  - Loop parallelization
- Tool implementation
  - Automated heap analyzer
  - Automated source-to-source transformations (synthesizability and parallelization)
  - Successful parallelization using standard HLS tools
- Future work
  - Use this analysis for efficient loop pipelining
  - … and automatic on-chip buffer insertion in interface to external memory
  - Compute worst-case/average-case bounds on heap usage

Thank you for listening.

http://cas.ee.ic.ac.uk/people/fw1811/