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

Felix Winterstein

14 May 2014

Contact: f.winterstein12@imperial.ac.uk

Executive summary

Executive summary

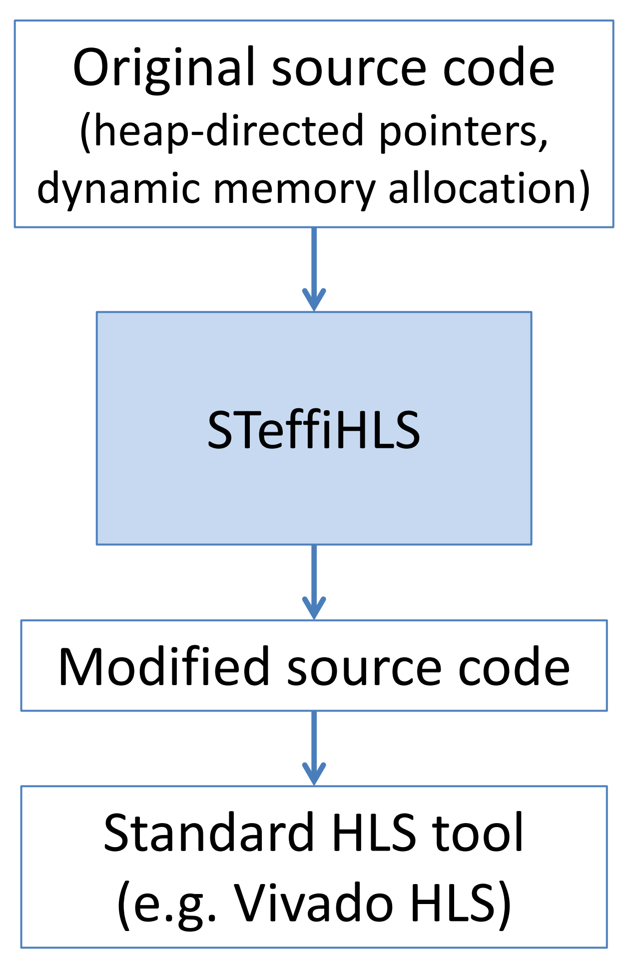
- HLS tools require manual source code refactoring…
  - … to map pointer-manipulating programs efficiently into HW
- Static program analysis
  - Analyse pointer-based memory accesses and heap layout
  - Identify disjoint, independent regions in heap memory
- Source-to-source transformations
  - Partition heap across on-chip memory banks
  - Automatic loop parallelization

Imperial College London

Executive summary

Executive summary

- HLS tools require manual source code refactoring...
  - ... to map pointer-manipulating programs efficiently into HW
- Static program analysis
  - Analyse pointer-based memory accesses and heap layout
  - Identify disjoint, independent regions in heap memory
- Source-to-source transformations
  - Partition heap across on-chip memory banks
  - Automatic loop parallelization



FPGAs everywhere

Some examples …

… of COTS FPGAs in ESA's Ground Station Division

© S. Marti, www.esa.int

Imperial College London

FPGAs everywhere

Some examples …
… of COTS FPGAs in ESA's Ground Station Division

Device
Mono-static space surveillance radar
Bi-static space surveillance radar
Phased array radar demonstrator (in-house cross verification)
Ground station modem system
NextGen tracking, telemetry and command processor
Combined high-rate telemetry and ranging transceiver
OPS-SAT: experimental satellite in low Earth orbit
SARAS: direction finding for rapid signal acquisition

Imperial College London

FPGAs everywhere

Some examples …

… of COTS FPGAs in ESA's Ground Station Division

Device
Mono-static space surveillance radar
SARAS: direction finding for rapid signal acquisition

Modern FPGAs allow us to map increasingly complex applications to reconfigurable logic

If they weren't so hard to program…

Imperial College London

HLS tools (examples)

Tool	Input language
Cadence C-to-Silicon	C
Synopsys Synphony C Compiler	C
Mentor Graphics Catapult C	C
Impulse CoDeveloper	C
Xilinx Vivado HLS	C
Bluespec	BSV
National Instruments LabVIEW FPGA	LabVIEW schematic
Xilinx System Generator for DSP	Matlab/Simulink
DEFACTO	C
ROCCC	C
LegUP	C
Chisel	Scala

Imperial College London

HLS – Applications

Examples of HLS applications

Author	Application
Meeus et al. (2012)	Image processing
Sarkar et al. (2009)	Stream-based video processing
BDTI (2010)	- Stream-based video processing - Stream-based signal processing
Hammani et al. (2008)	- Image and signal processing - Ray casting
Cong et al. (2013)	Image processing
Cong et al. (2011)	Image processing

- Mainly regular control flow/ memory access
- Pointers, dynamic memory allocation, linked data structures? Worth considering at all?

Imperial College London

Outline

- **Case study: High-level synthesis of dynamic data structures**
- Challenge
- Motivating example
- Leveraging recent advances in software verification
- Implementation and results
- Outlook

Imperial College London

Case study: *K*-means clustering

Case study:

- Compare computational properties of two algorithms for *K*-means clustering
- SW/ RTL/ HLS implementations
- Code available on GitHub (Vivado-KMeans)

Imperial College
London

K-means clustering

600
400
200
0
-200
-400
-600
-800
-1000
-1200
-200
0
200
400
600
800
1000

N data points

Imperial College London

K-means clustering

- Automatic partitioning

-200 0 200 400 600 800 1000

600 400 200 0 -200 -400 -600 -800 -1000 -1200

N data points

Imperial College London

K-means clustering

- Automatic partitioning
- *K* is a known parameter (e.g. *K*=4)
- Clusters represented by their centres
- Centre position determines cluster assignment
- Find optimal positioning

1 2 3 4

$\|x_i - z_1\|^2$

N data points

Imperial College London

K-means clustering

- Automatic partitioning
- K is a known parameter (e.g. K=4)
- Clusters represented by their centres
- Centre position determines cluster assignment
- Find optimal positioning

600
400
200
0
-200
-400
-600
-800
-1000
-1200

-200 0 200 400 600 800 1000

1 2 3 4

N data points

Imperial College London

K-means clustering

- Automatic partitioning
- K is a known parameter (e.g. K=4)
- Clusters represented by their centres
- Centre position determines cluster assignment
- Find optimal positioning

600
400
200
0
-200
-400
-600
-800
-1000
-1200

-200 0 200 400 600 800 1000

1 2 3 4

N data points

Imperial College London

K-means clustering

- Automatic partitioning
- *K* is a known parameter (e.g. *K*=4)
- Clusters represented by their centres
- Centre position determines cluster assignment
- Find optimal positioning

600
400
200
0
-200
-400
-600
-800
-1000
-1200
-200 0 200 400 600 800 1000
1 2 3 4
N data points

Imperial College London

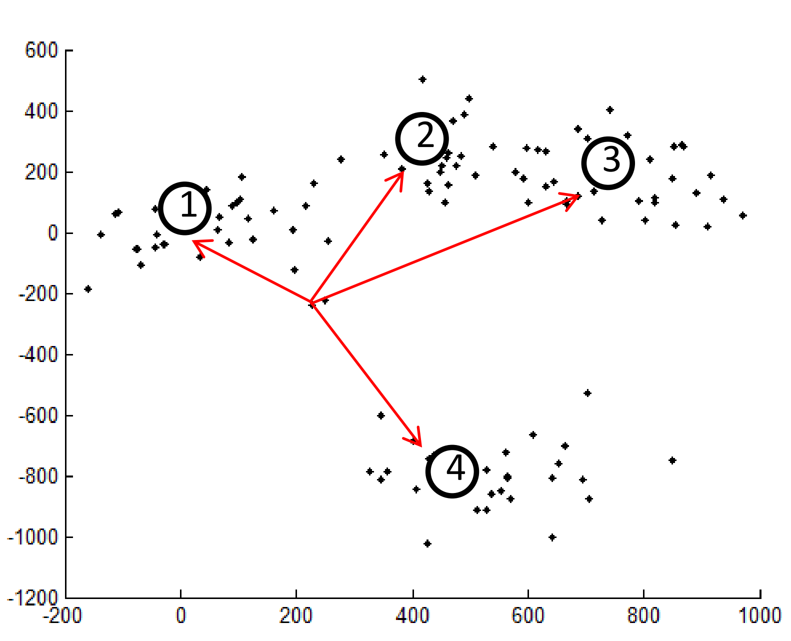
K-means clustering

- Automatic partitioning
- *K* is a known parameter (e.g. *K*=4)
- Clusters represented by their centres
- Centre position determines cluster assignment
- Find optimal positioning

N data points

Imperial College London

Brute-force algorithm

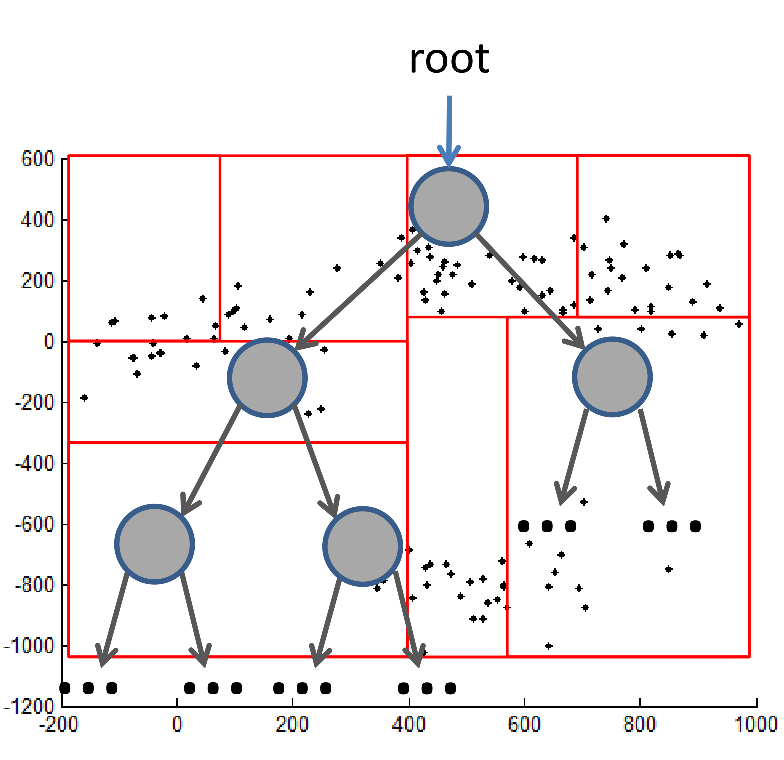


for all $x_i \in data\ points$ **do**
    for all $z_j \in centers$ **do**
        $compute\ distance\ \|x_i - z_j\|^2$
        $pick\ \&\ update\ closest\ center$
    end for
end for

- For each data point …
- search among *K* candidates for the closest center
- Popular for HW implementation
  Leeser 2001, Estlick 2001, Wang 2007, Kutty 2013, …

Imperial College London

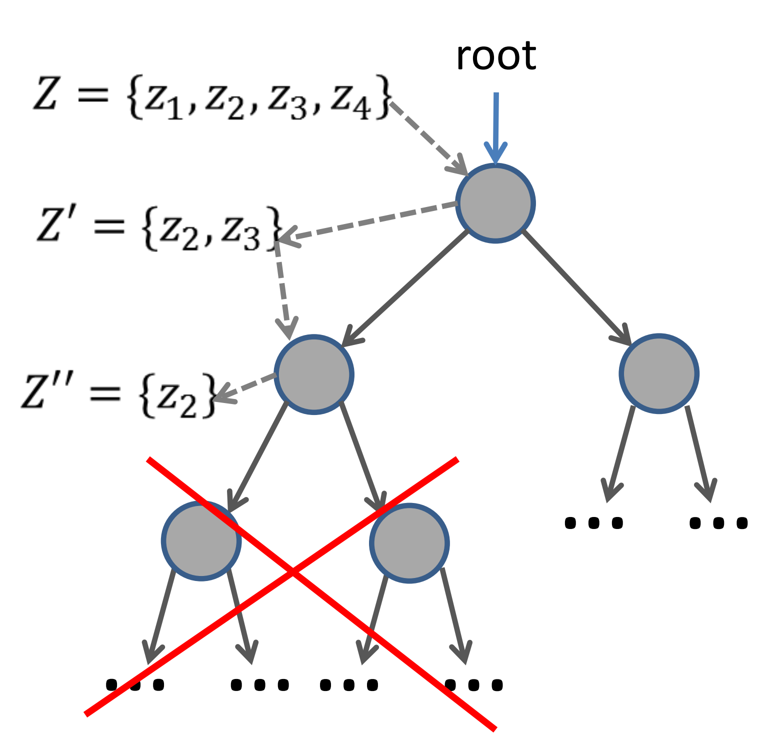
Tree-based algorithm*



- Recursively split data set
- Build a pointer-linked tree data structure

* "The Filtering Algorithm", Kanungo et al., 2002

Tree-based algorithm*

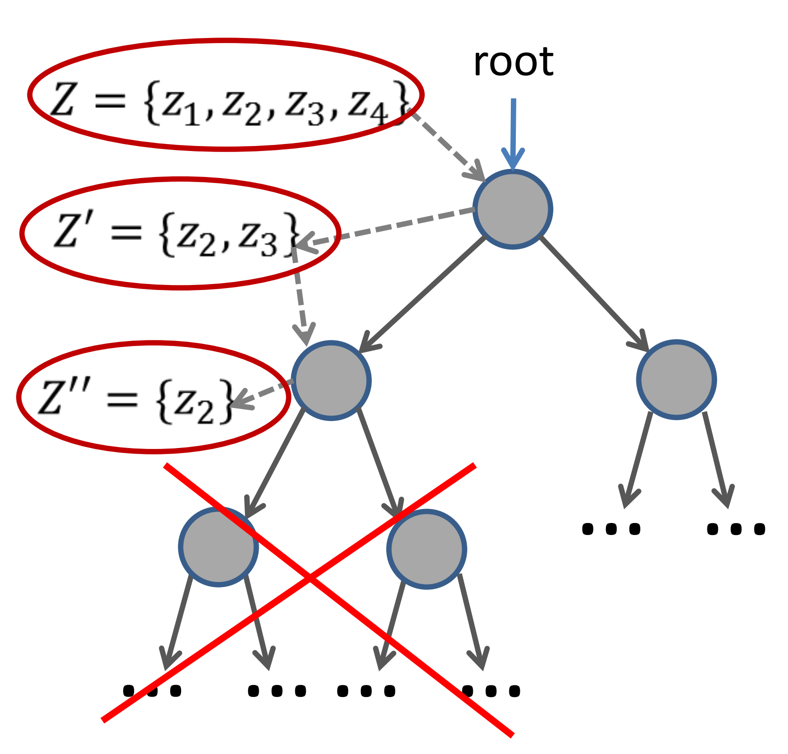


- Recursively split data set
- Build a pointer-linked tree data structure
- Clustering: Recursive tree traversal
- Acceleration through
  - Considering ONLY PROMISING candidates in the search for the closest center
  - Search space pruning (sub-trees)

* "The Filtering Algorithm", Kanungo et al., 2002

Imperial College London

Tree-based algorithm*

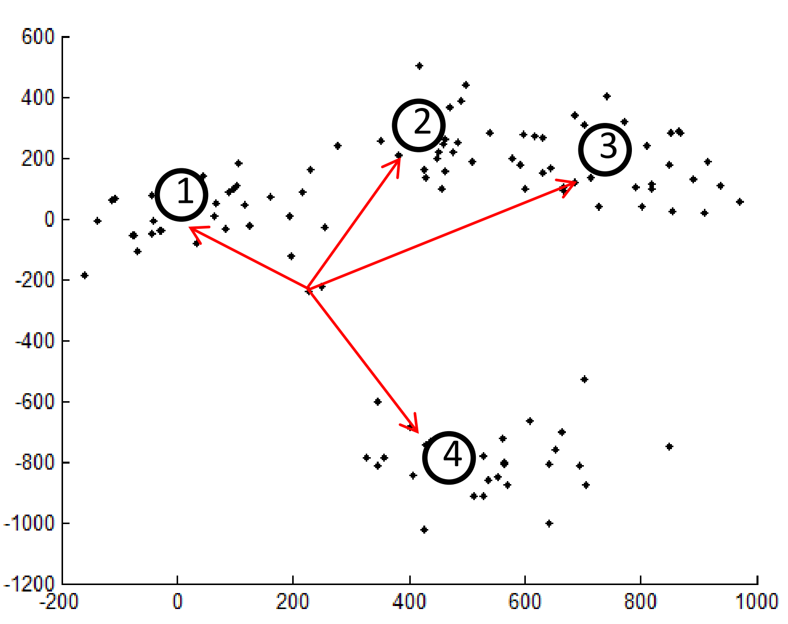


- Recursively split data set
- Build a pointer-linked tree data structure
- Clustering: Recursive tree traversal
- Acceleration through
  - Considering ONLY PROMISING candidates in the search for the closest center
  - Search space pruning (sub-trees)
- Dynamically (de-)allocate memory to store intermediate results

* "The Filtering Algorithm", Kanungo et al., 2002

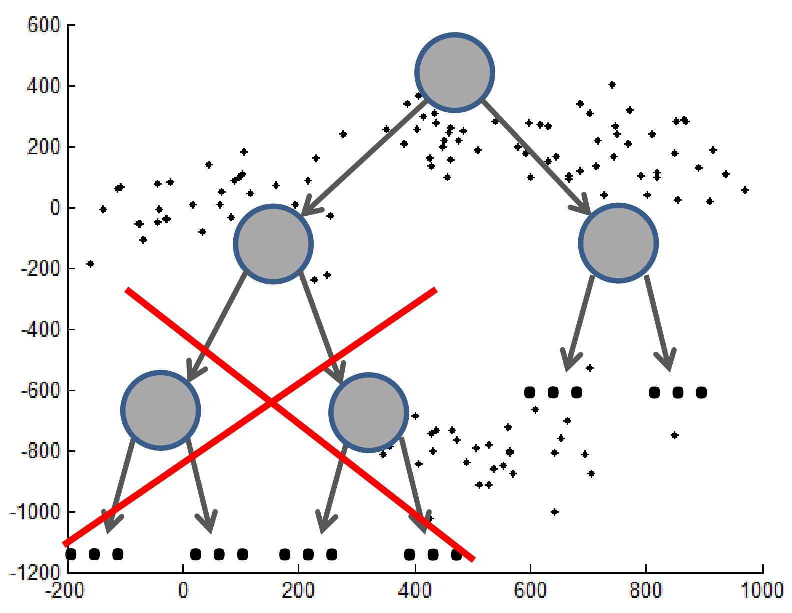
Imperial College London

Same result – two algorithms



Brute-force algorithm

- Computationally expensive
- Simple control flow
- Embarrassingly parallel



Tree-based algorithm

- Data-dependent control flow
- Pointer-based tree traversal
- Dynamic memory allocation

Results

The battlefield

Implementation

10^{-1} 10^{0} 10^{1} 10^{2}

Time per clustering iteration [ms]

Identical area constraint for FPGA implementations: 6500 slices

Results

The battlefield

Implementation

Software
C++, GCC -O3
Intel i7, 3.4GHz

Tree-based

Brute-force

$^1/_{6.3}$

10^{-1} 10^{0} 10^{1} 10^{2}

Time per clustering iteration [ms]

Identical area constraint for FPGA implementations: 6500 slices

Imperial College London

Results

The battlefield

Implementation

Software
C++, GCC -O3
Intel i7, 3.4GHz

Hand-crafted RTL
Virtex7 FPGA

Identical area constraint for FPGA implementations: 6500 slices

Imperial College London

Results

The battlefield

Identical area constraint for FPGA implementations: 6500 slices

Imperial College London

Results

The battlefield

Identical area constraint for FPGA implementations: 6500 slices

Imperial College London

Results

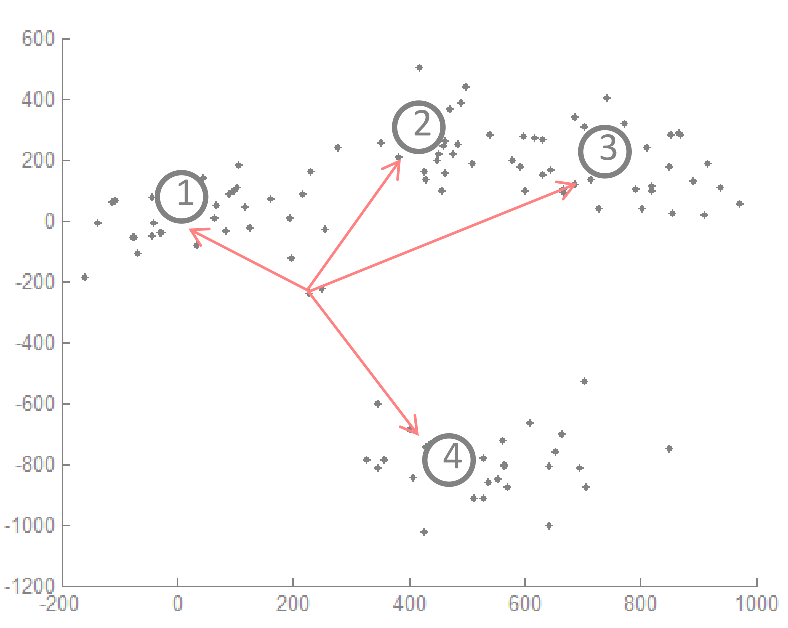
The battlefield

Tree-based algorithm:
7.2x improvement after extensive source code refactoring

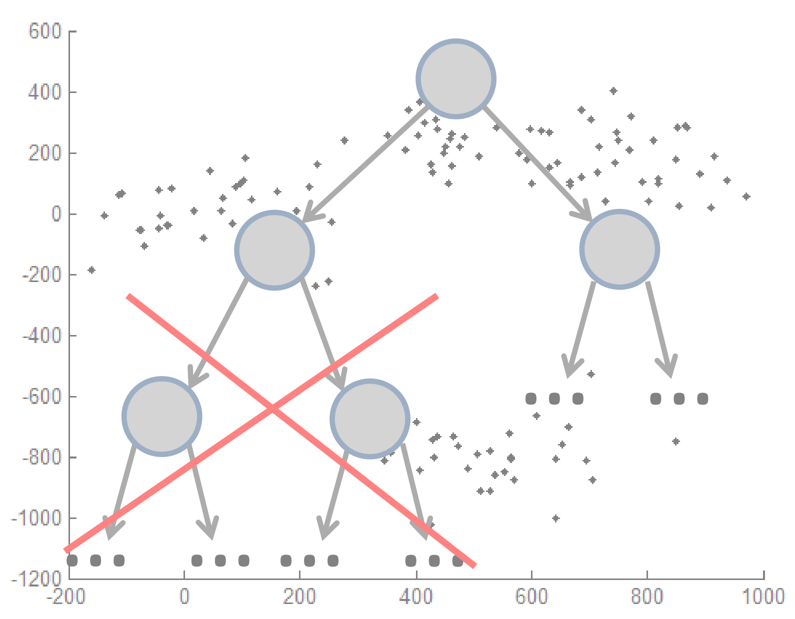
Identical area constraint for FPGA implementations: 6500 slices

Imperial College London

Same result – two algorithms



- Computationally expensive
- Simple control flow
- Embarrassingly parallel
- Seamless C-to-FPGA implementation



Tree-based algorithm

- Data-dependent control flow
- Pointer-based tree traversal
- Dynamic memory allocation
- C-to-FPGA requires substantial code modifications

Imperial College London

Code transformations

Tree-based algorithm:

- Memory partitioning
- Parallelization
- Custom implementation of dynamic memory allocation
- Loop flattening
- Loop distribution
- Custom bit widths
- …

Code transformations

Automate

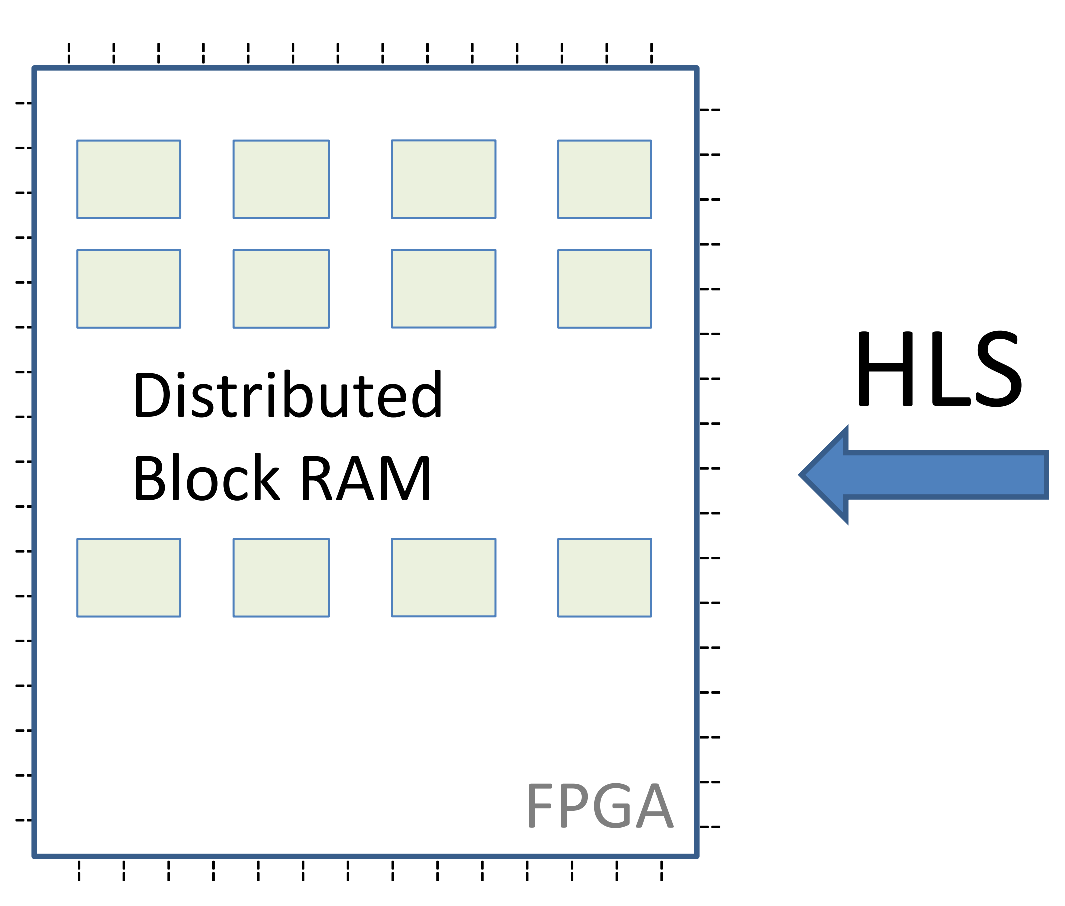
- Memory partitioning
- Parallelization
- Custom implementation of dynamic memory allocation
- Loop flattening
- Loop distribution
- Custom bit widths
- …

Outline

- Case study: High-level synthesis of dynamic data structures
- **Challenge**
- Motivating example
- Leveraging recent advances in software verification
- Implementation and results
- Outlook

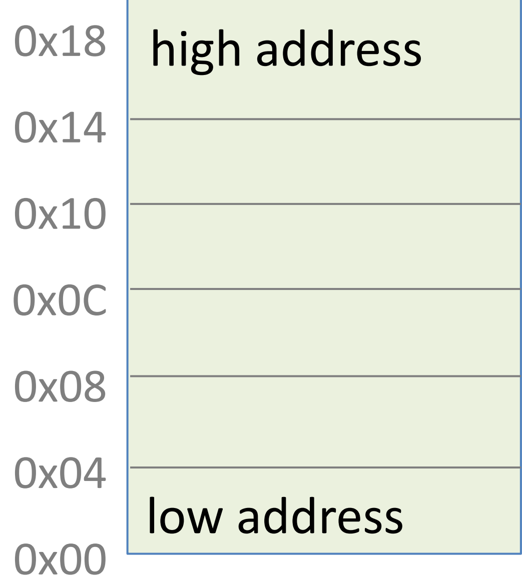
Imperial College London

Challenge



HLS

SW memory model




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

Works well for 'regular' memory accesses
(polyhedral model, ex. Cong, Pouchet..)

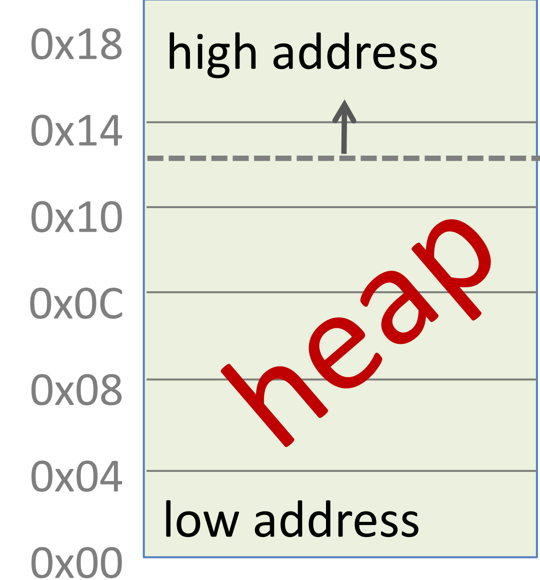
Imperial College London

Challenge

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;
  ...
}
```

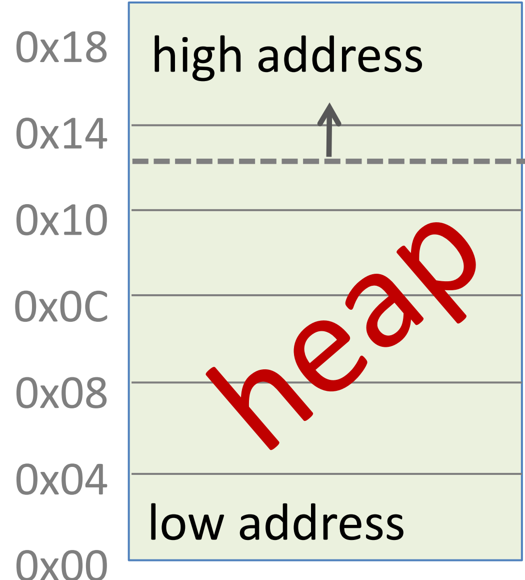
Challenge

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

STeffiHLS takes a step towards closing this gap

SW memory model



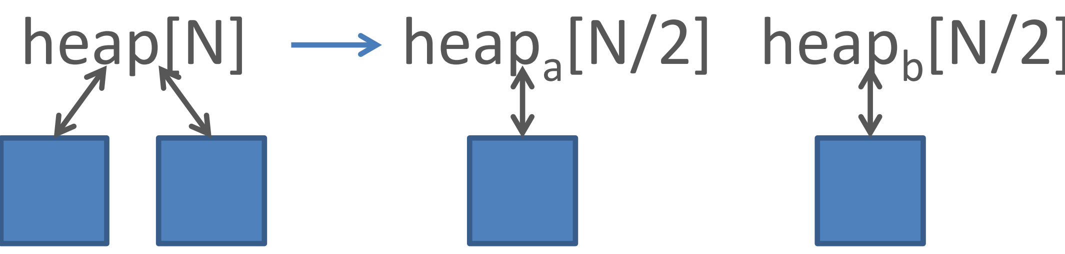

```
int main() {
  x = A[i];
  p = new int;
  *p = 3;
  …
}
```

Imperial College London

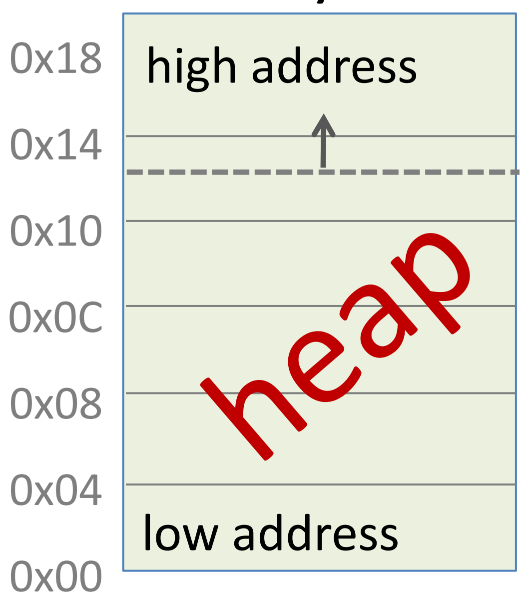
Challenge

Our goal

- Partition heap-allocated data structures ('heaplets')
- Synthesize a parallel implementation



SW memory model

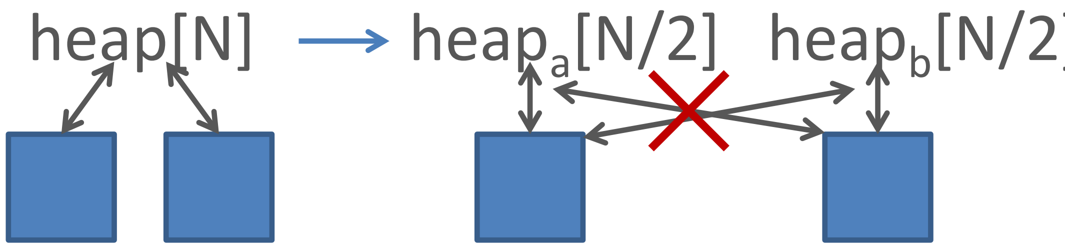


Imperial College London

Challenge

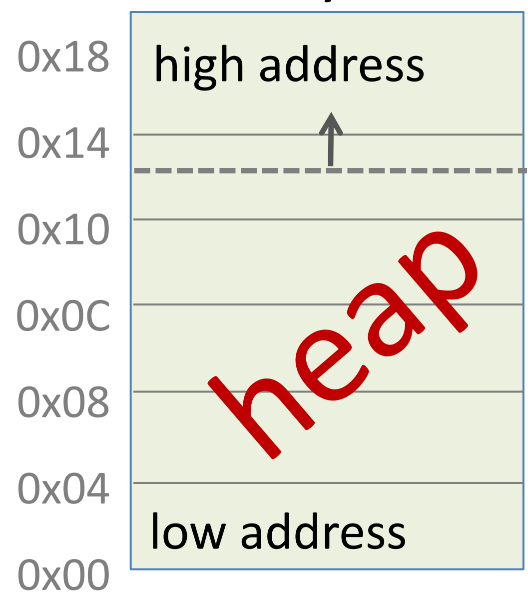
<u>Our goal</u>

- Partition heap-allocated data structures ('heaplets')
- Synthesize a parallel implementation



- Ensure that heap partitions are 'private'

SW memory model



Outline

- Case study: High-level synthesis of dynamic data structures
- Challenge
- **Motivating example**
- Leveraging recent advances in software verification
- Implementation and results
- Outlook

Imperial College London

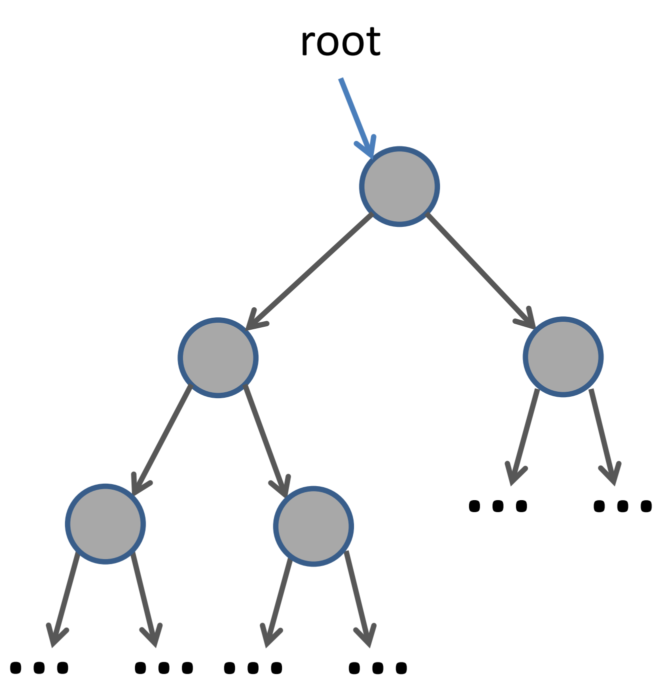
Motivating example

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
```

Imperial College London

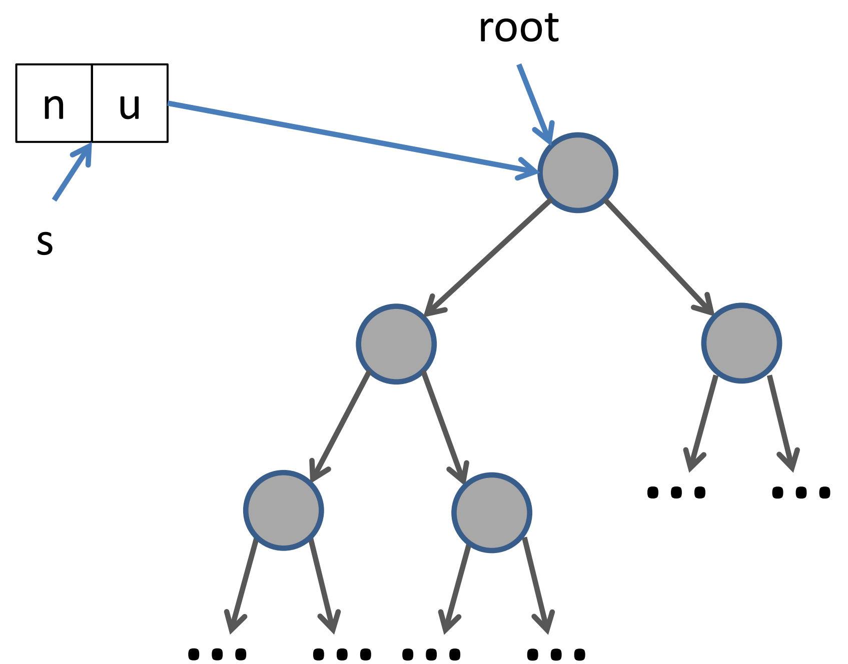
Motivating example




```
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
```

Imperial College London

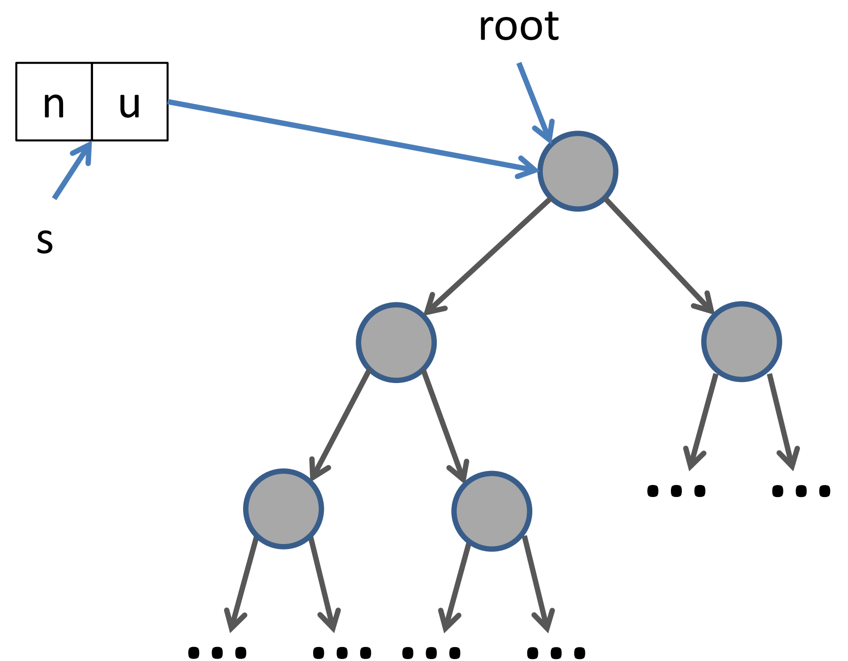
Motivating example




```
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
```

Imperial College London

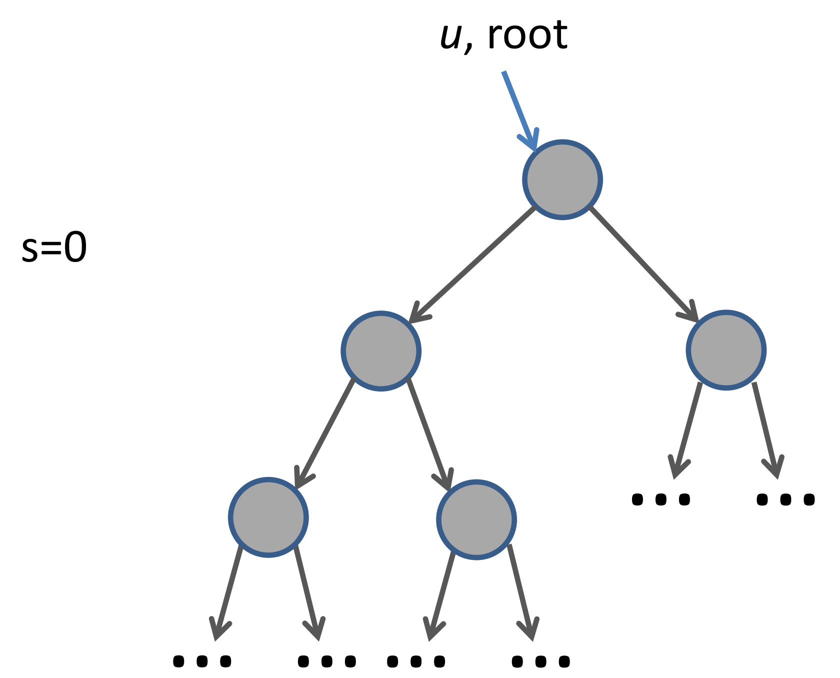
Motivating example




```
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
```

Imperial College London

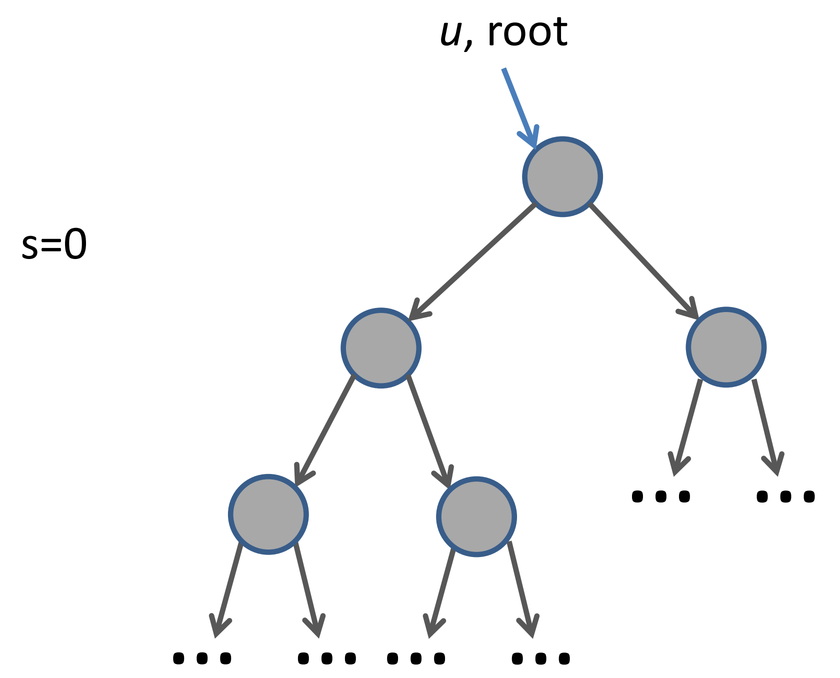
Motivating example




```
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
```

Imperial College London

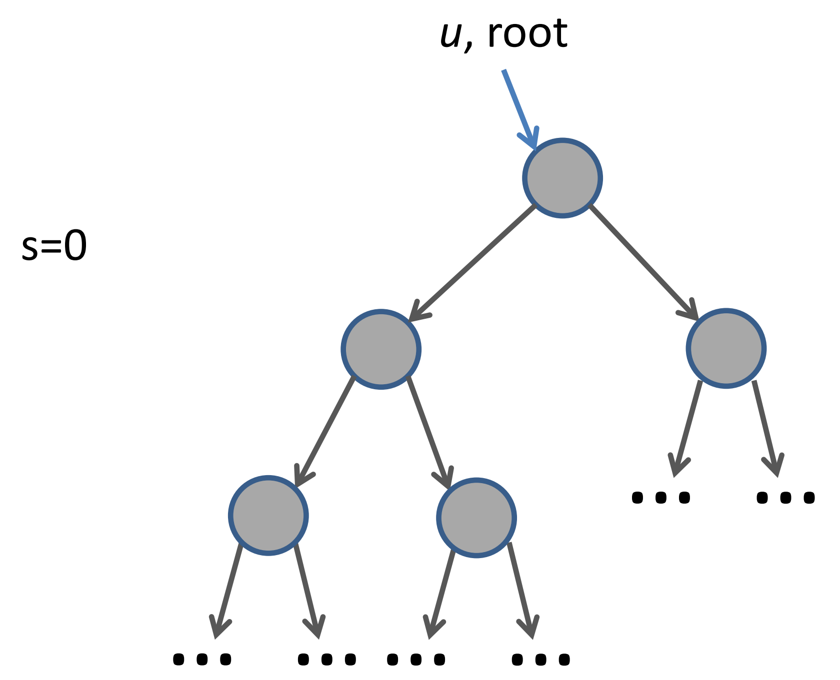
Motivating example




```
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
```

Imperial College London

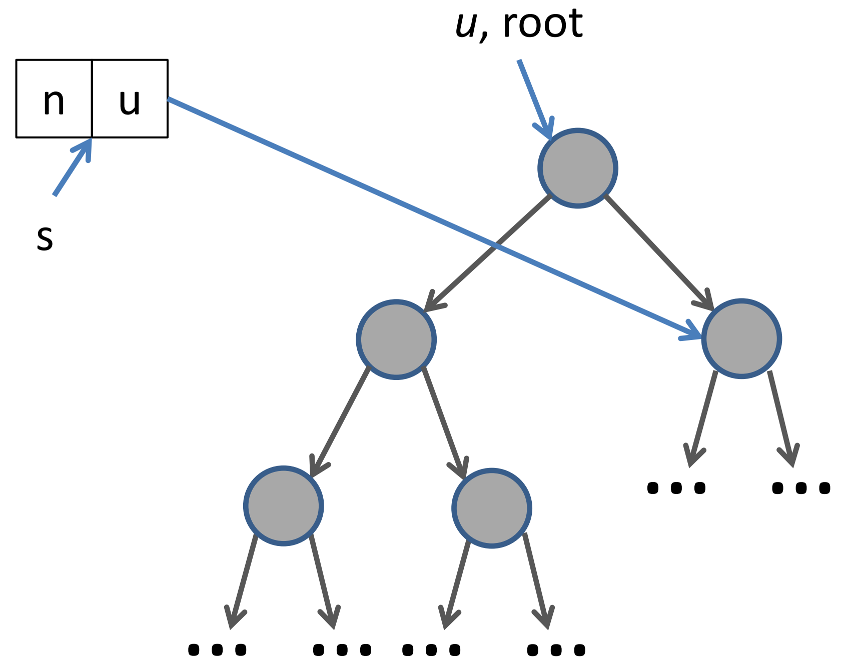
Motivating example




```
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
```

Imperial College London

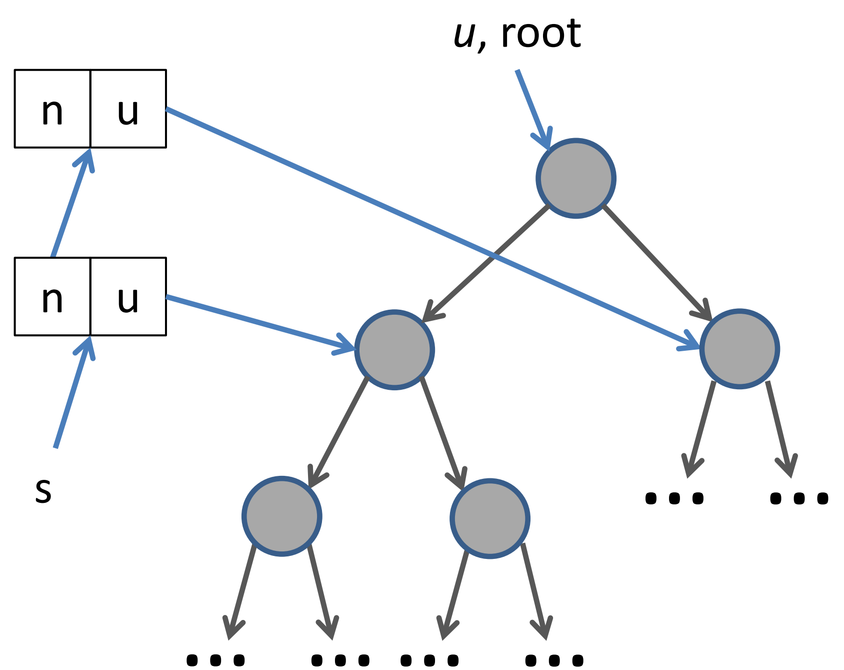
Motivating example




```
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
```

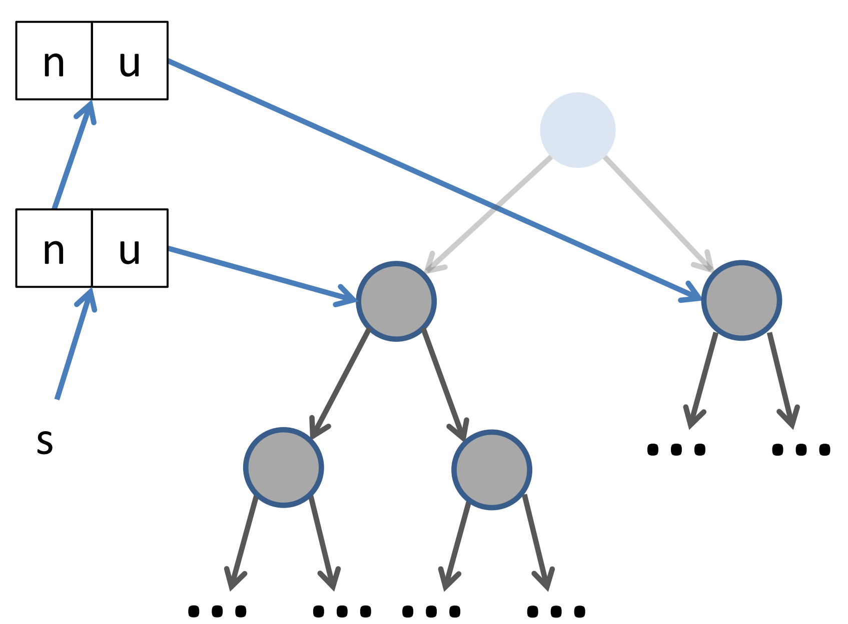
Imperial College London

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
```

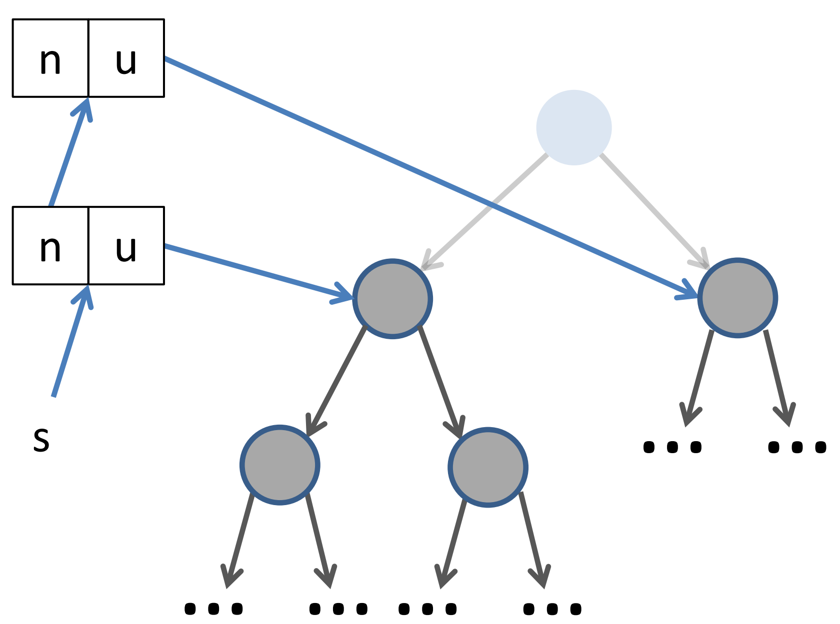
Motivating example




```
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
```

Imperial College London

Motivating example

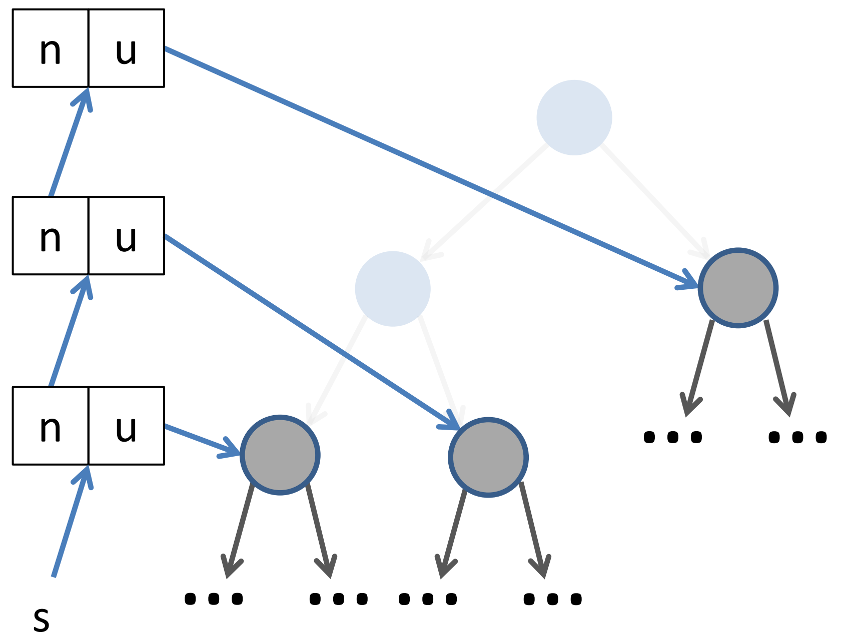



```
s = PUSH(root, s);
while s!=0 do
    s = POP(&u, s);
    ... do something
    if (u->left!= 0) && (u->right!=0) then
        s = PUSH(u->right, s);
        s = PUSH(u->left, s);
    end if
    delete u;
end while
```

Imperial College London

Motivating example

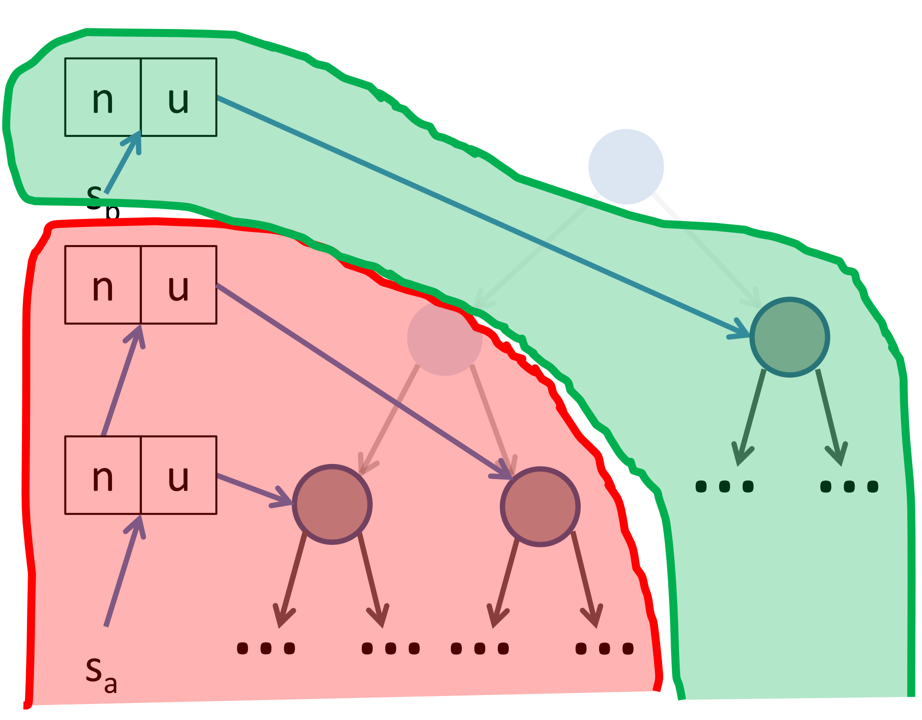
After two iterations…




```
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
```

Imperial College London

Motivating example

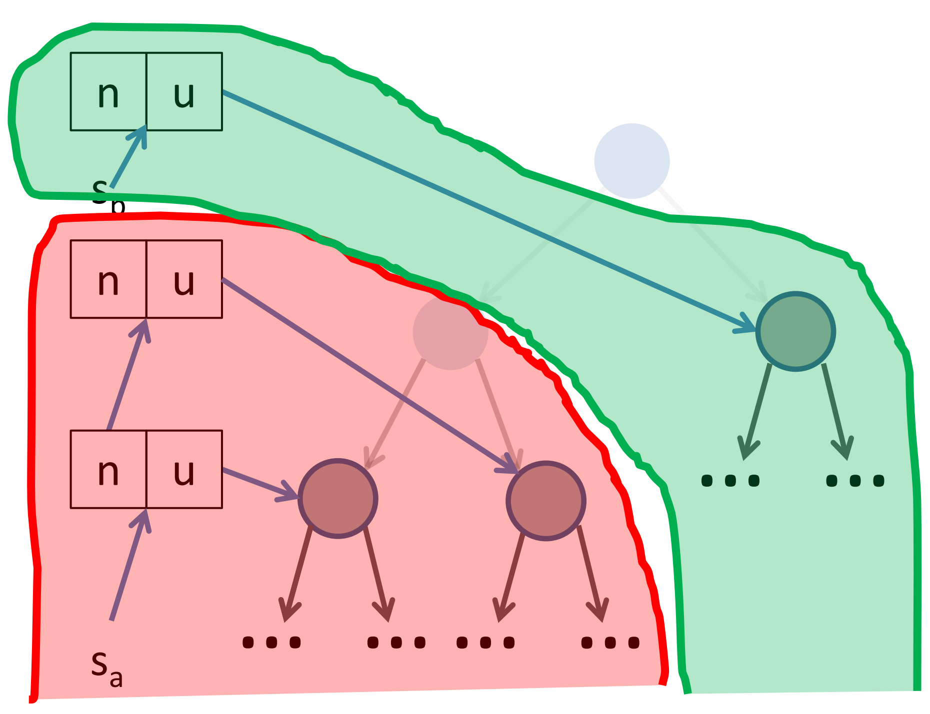


- Partition linked list and tree

```
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
```

Imperial College London

Motivating example



- Partition linked list and tree

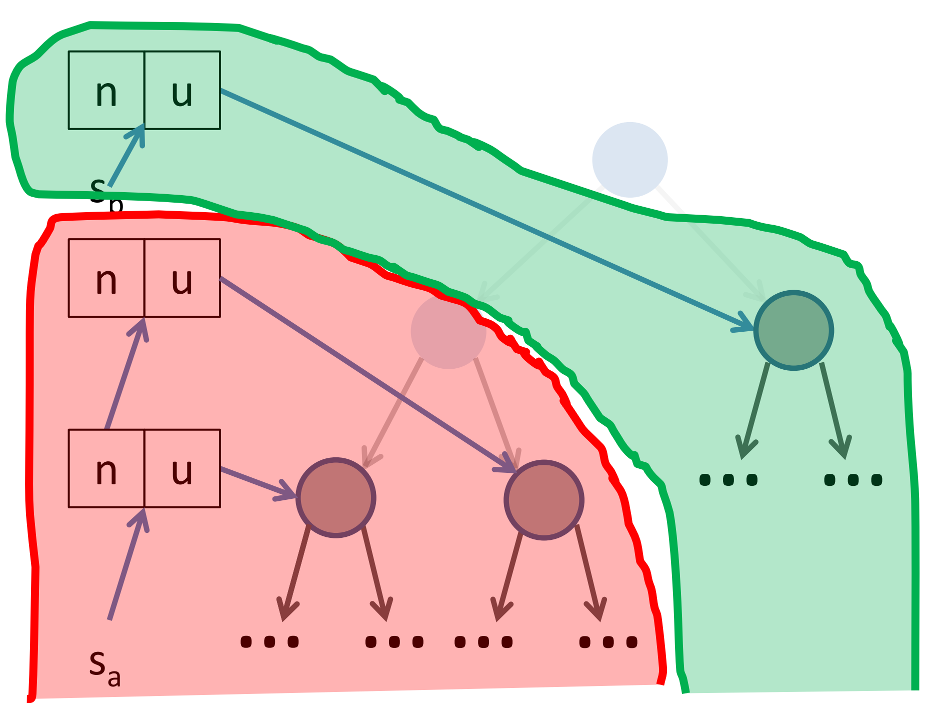
... preamble (accessing root node)

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

Imperial College London

Motivating example



... preamble (accessing root node)

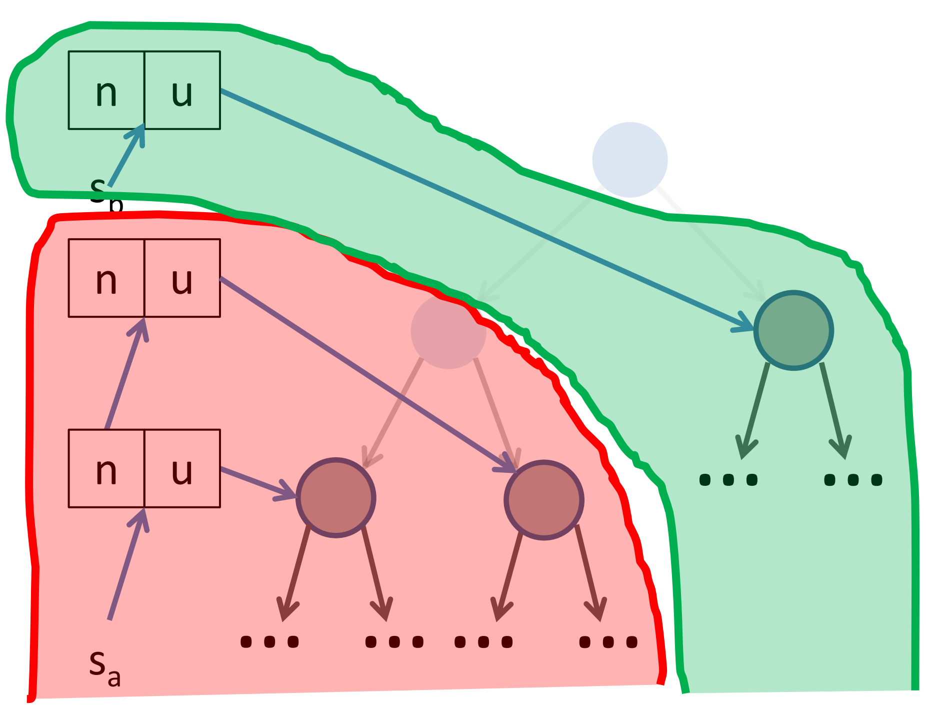
while s_a!=0 **do**
 ... loop body (access left sub-tree)
end while

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?

Imperial College London

Motivating example



... preamble (accessing root node)

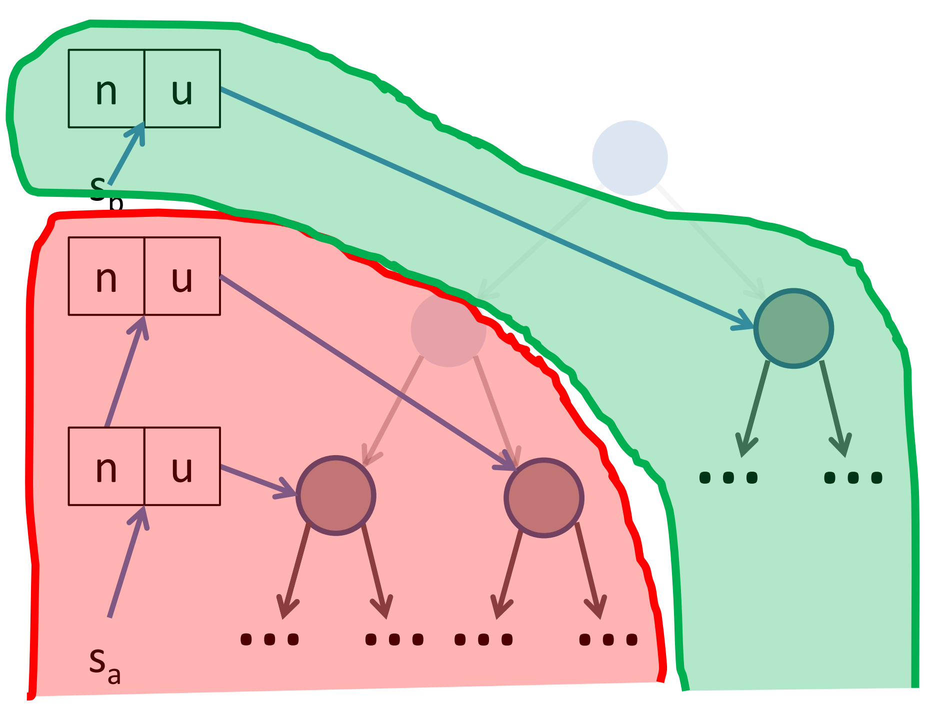
while s_a!=0 **do**
  ... loop body (access left sub-tree)
end while

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!

Imperial College London

Motivating example



... preamble (accessing root node)

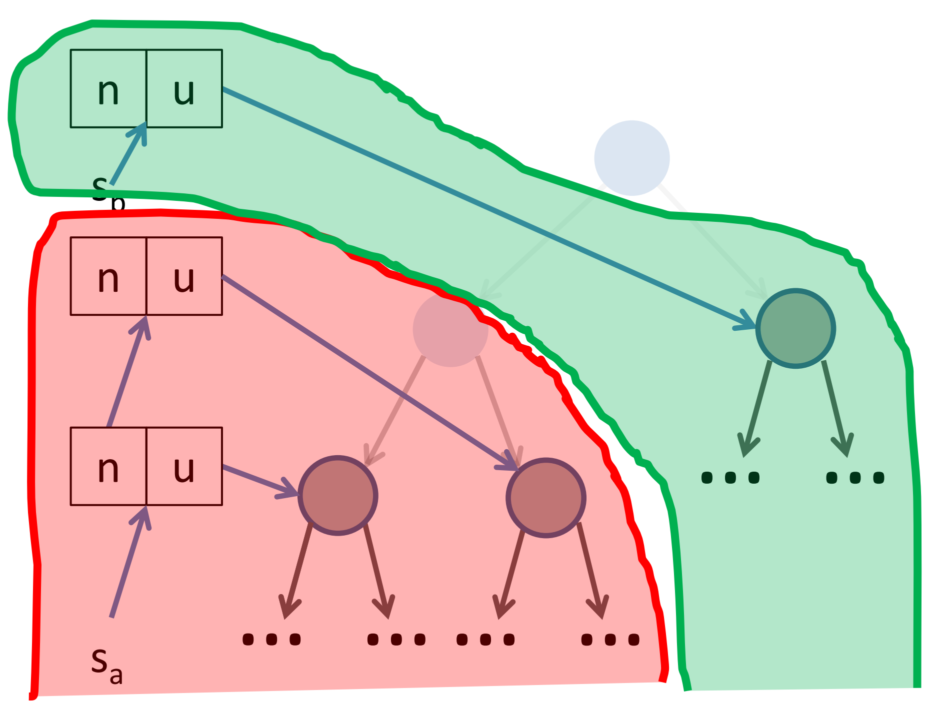
while s_a!=0 **do**
  ... loop body (access left sub-tree)
end while

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)

Imperial College London

Motivating example



... preamble (accessing root node)

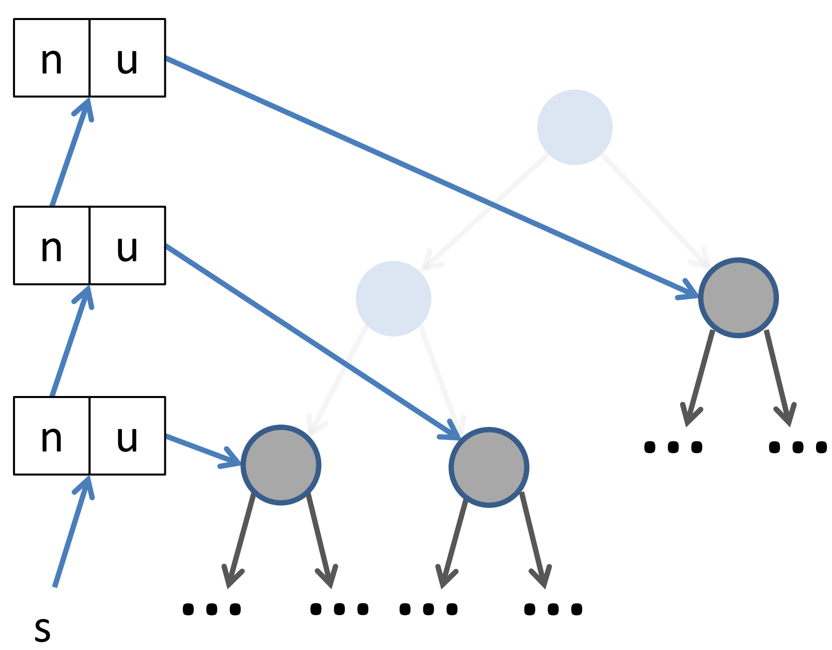
while s_a!=0 **do**
  ... loop body (access left sub-tree)
end while

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

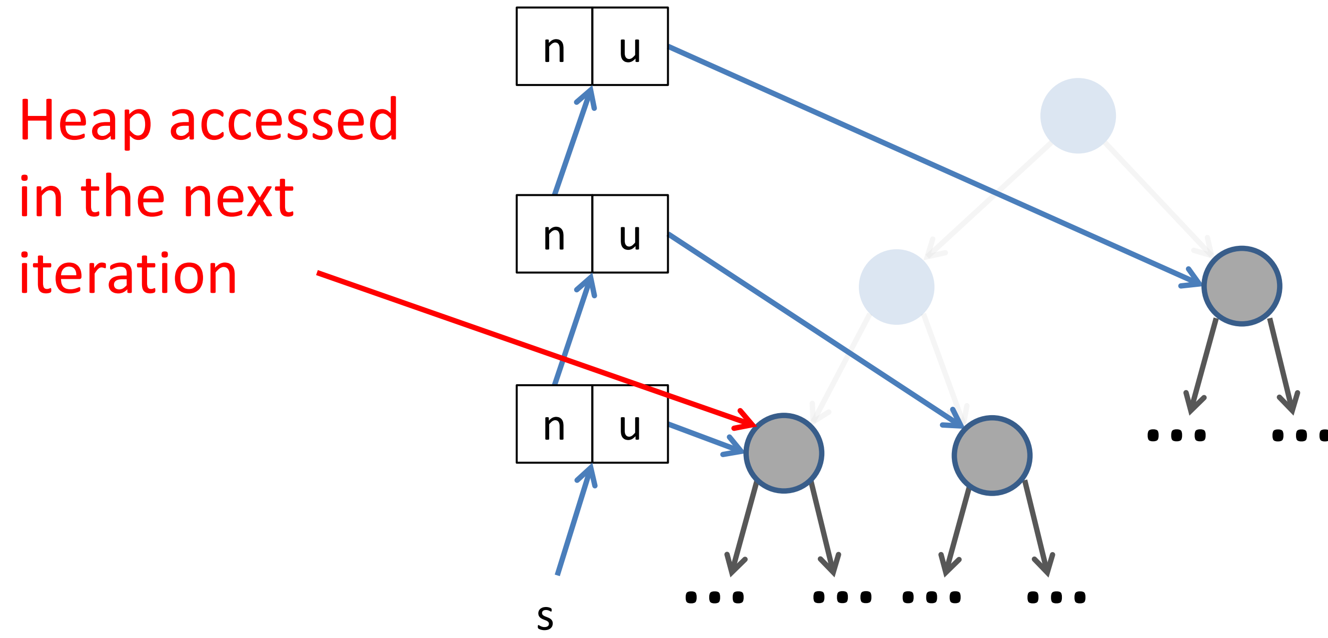
- Partition linked list and tree
- Will the red loop ever access data in the green partition? No!
- Parallelization is legal (does not violate data dependencies)
- Why is it hard for a tool to figure this out?

Imperial College London

The problem



The problem



Imperial College London

The problem

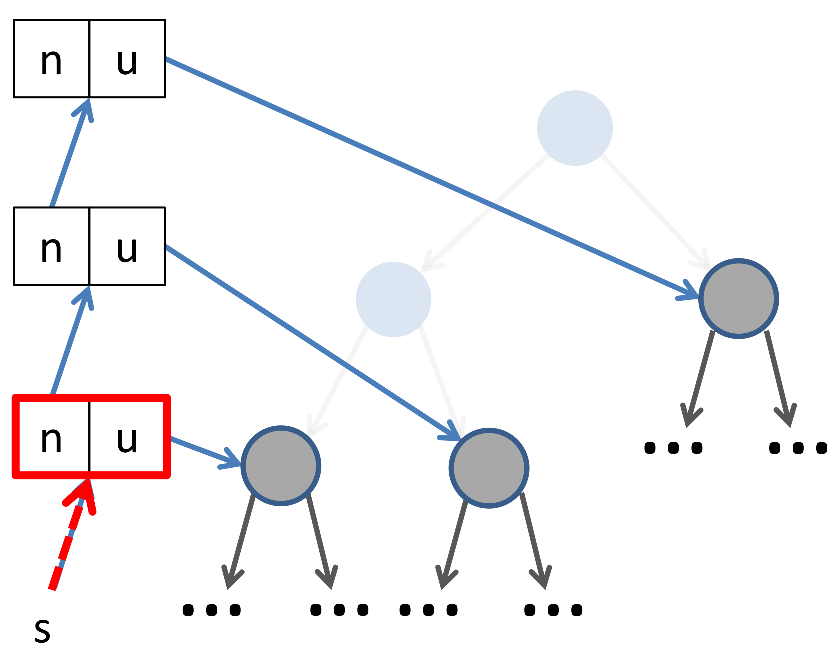
Imperial College London

The problem

- Do these iterations access the same memory cell?

Imperial College London

The problem

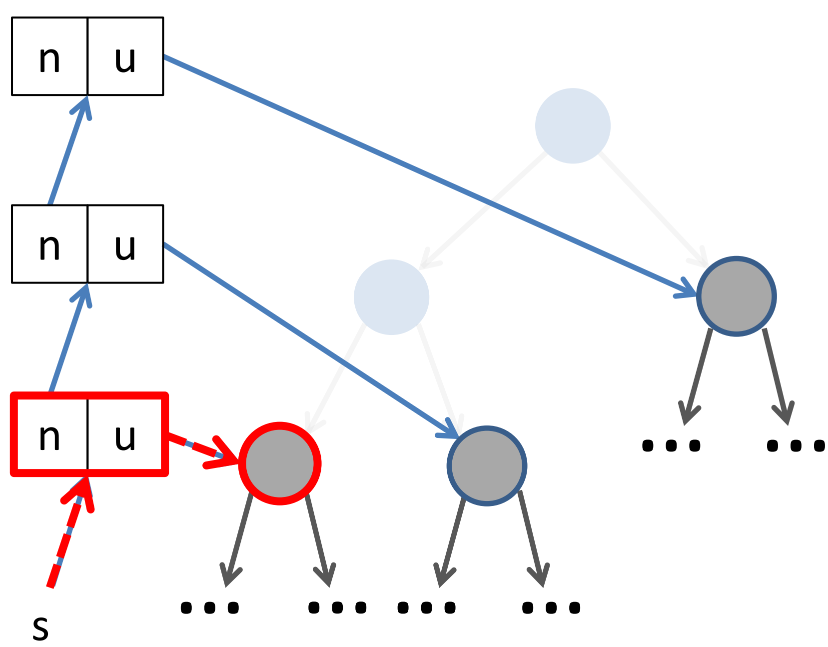


- Do these iterations access the same memory cell?

heap[s]

Imperial College London

The problem

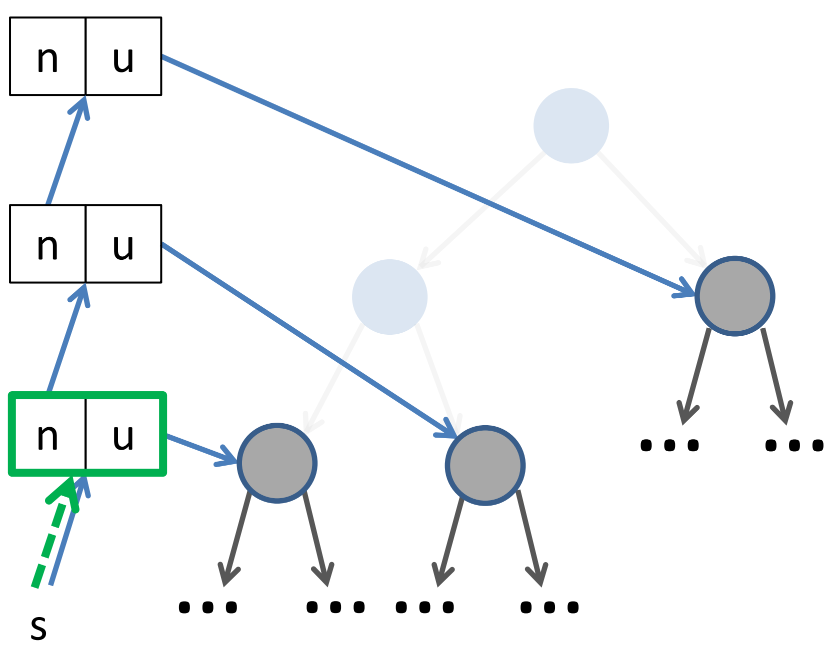


- Do these iterations access the same memory cell?

  heap[heap[s].u]

Imperial College London

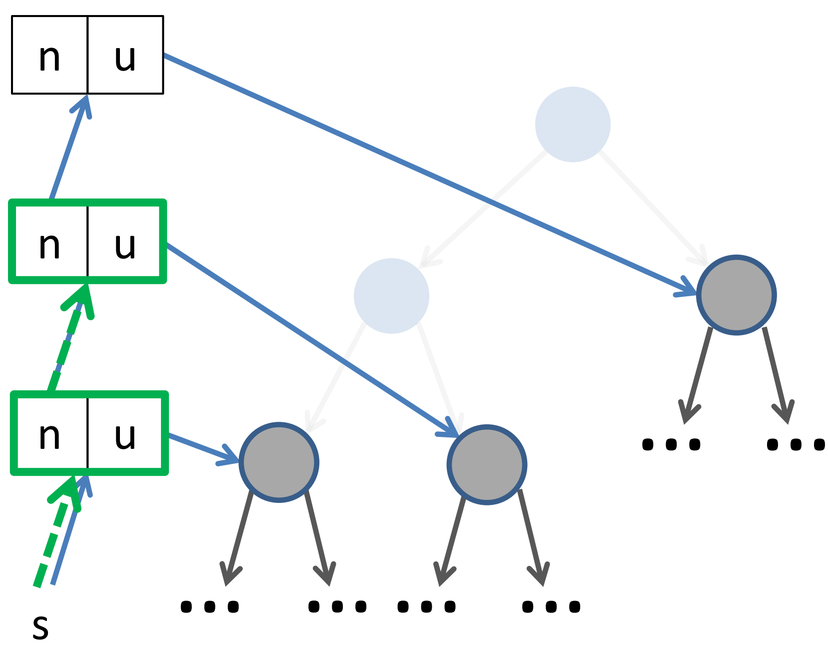
The problem



- Do these iterations access the same memory cell?

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

The problem

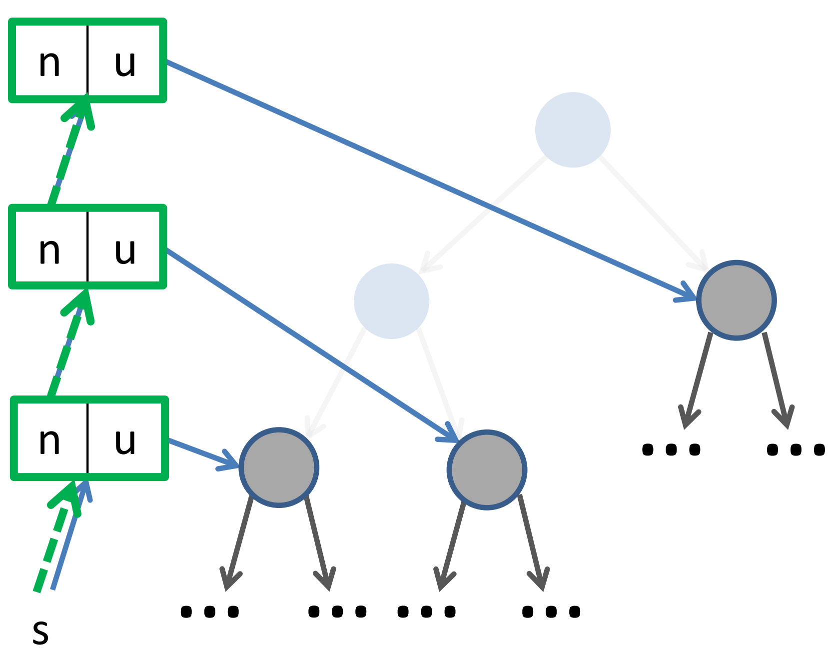


- Do these iterations access the same memory cell?

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

Imperial College London

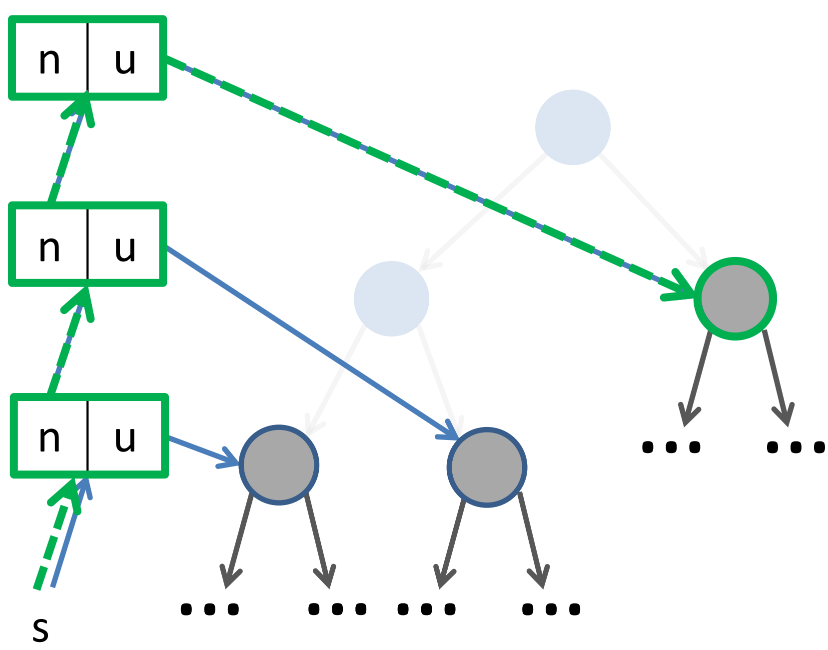
The problem



- Do these iterations access the same memory cell?

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

The problem

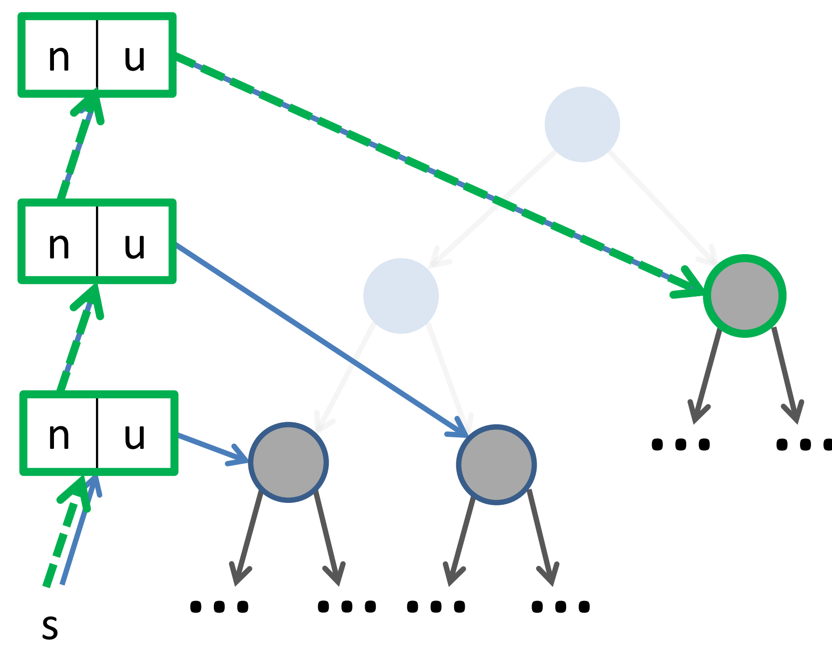


- Do these iterations access the same memory cell?

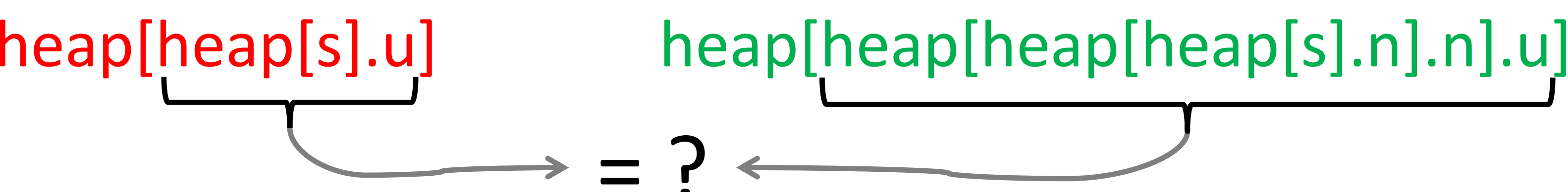
heap[heap[s].u] heap[heap[heap[heap[s].n].n].u]

Imperial College London

The problem

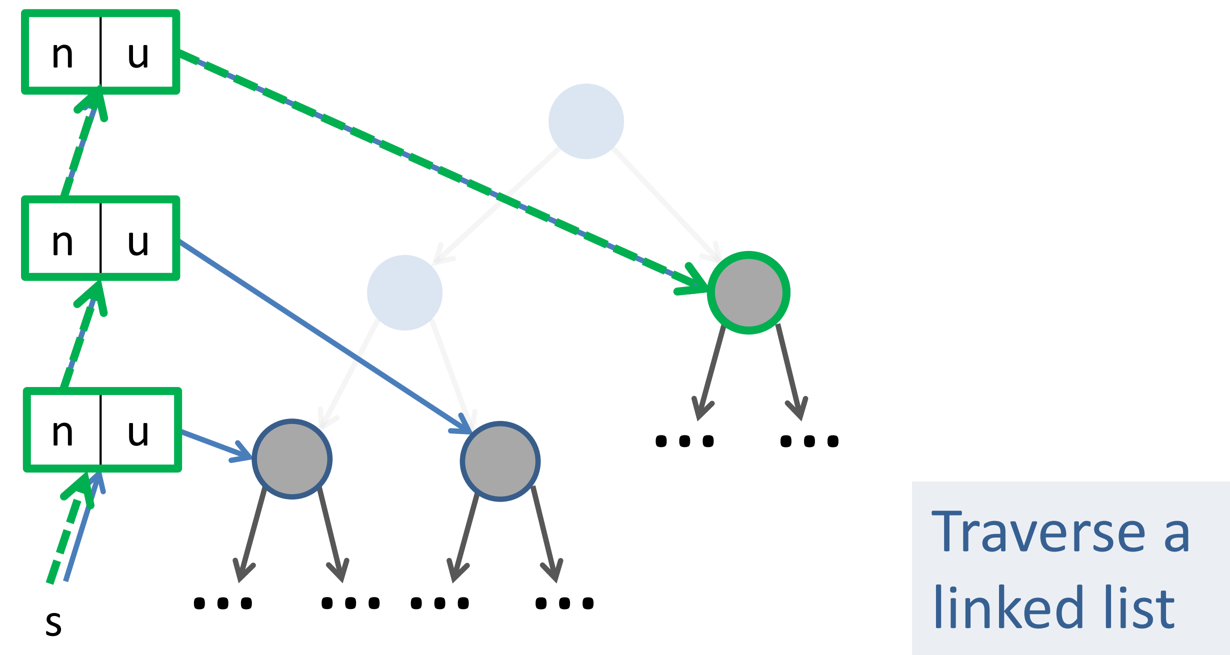


- Do these iterations access the same memory cell?

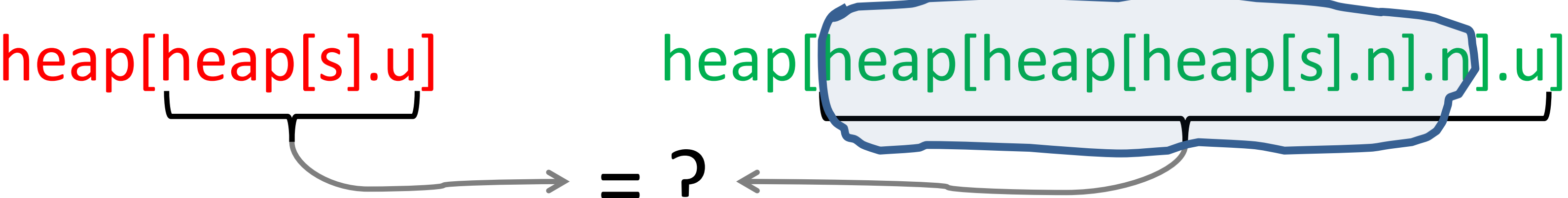


Imperial College London

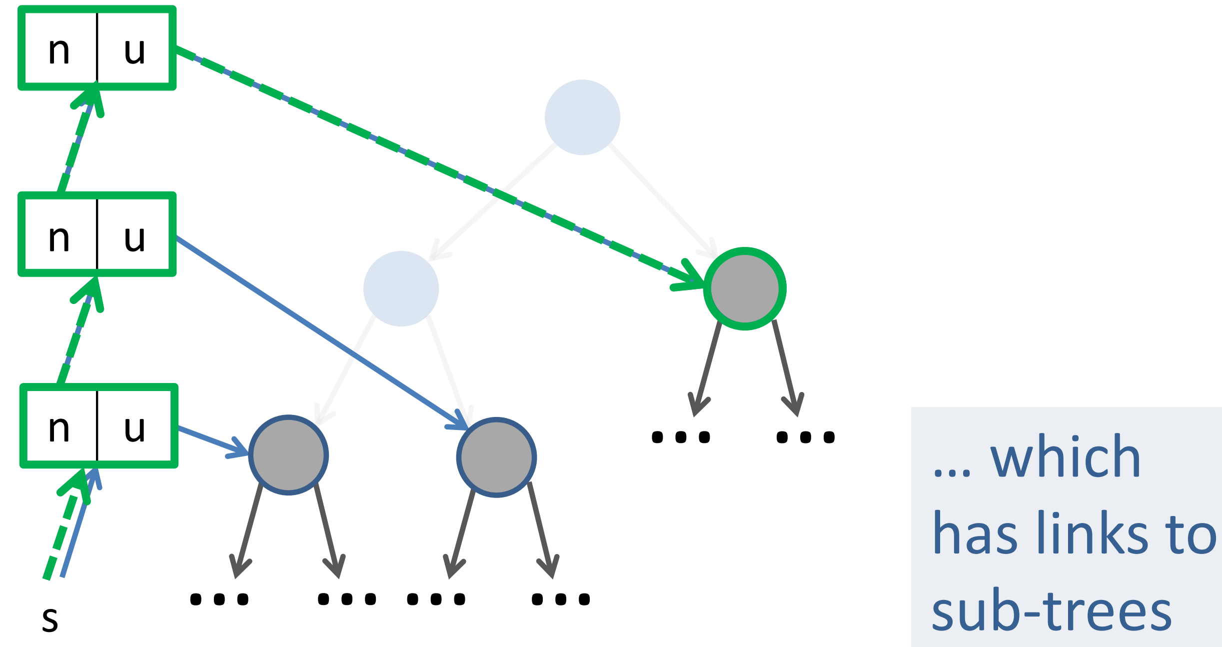
The problem



- Do these iterations access the same memory cell?



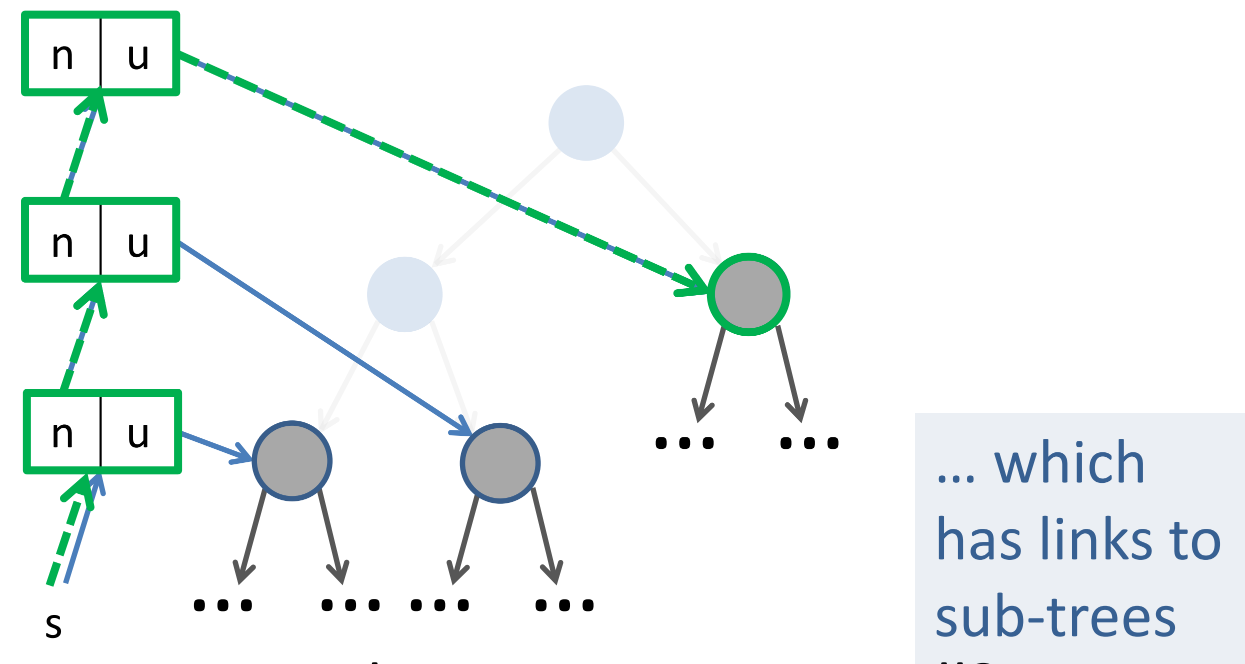
The problem



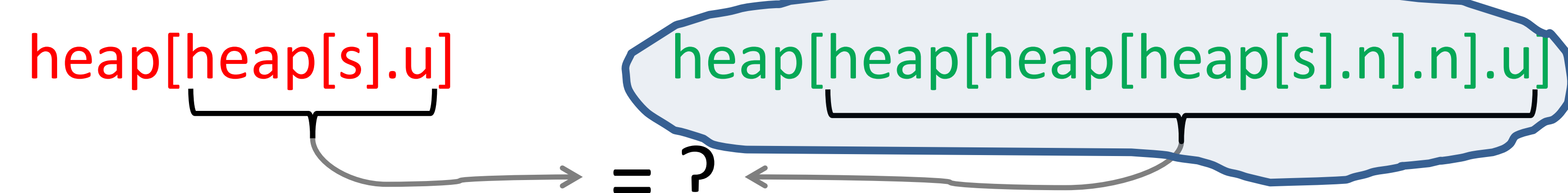
- Do these iterations access the same memory cell?

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

The problem



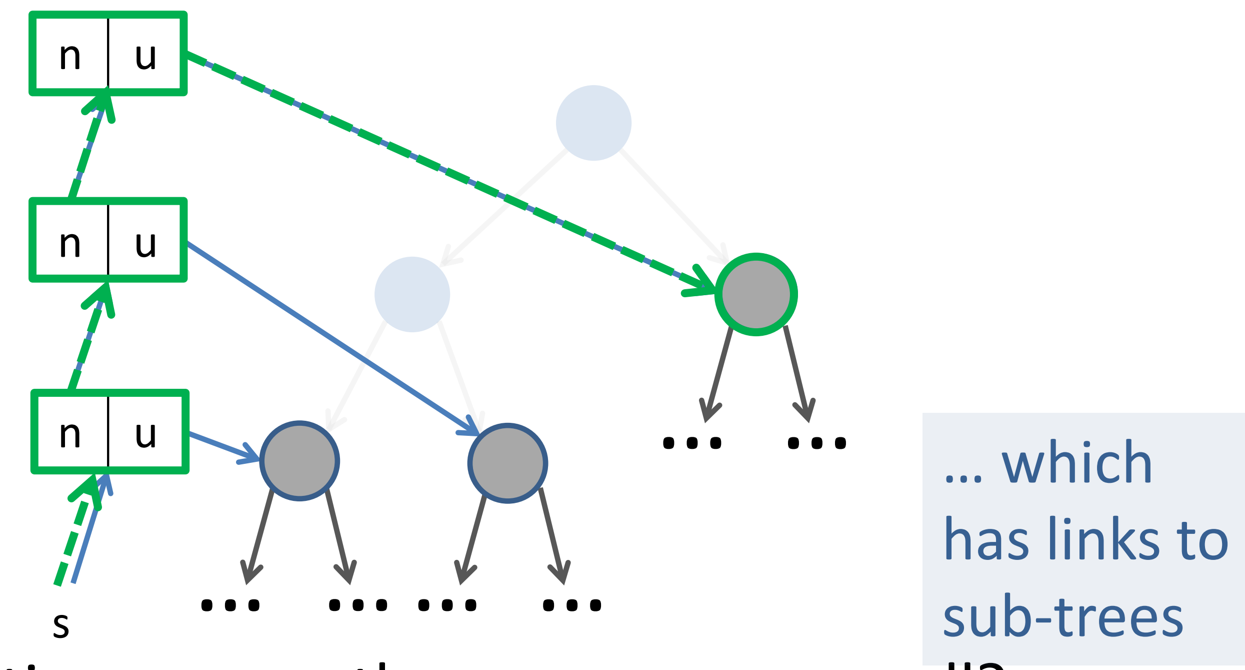
- Do these iterations access the same memory cell?



- Need to reason about structure, heap layout and disjointness

Imperial College London

The problem



- 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

Imperial College London

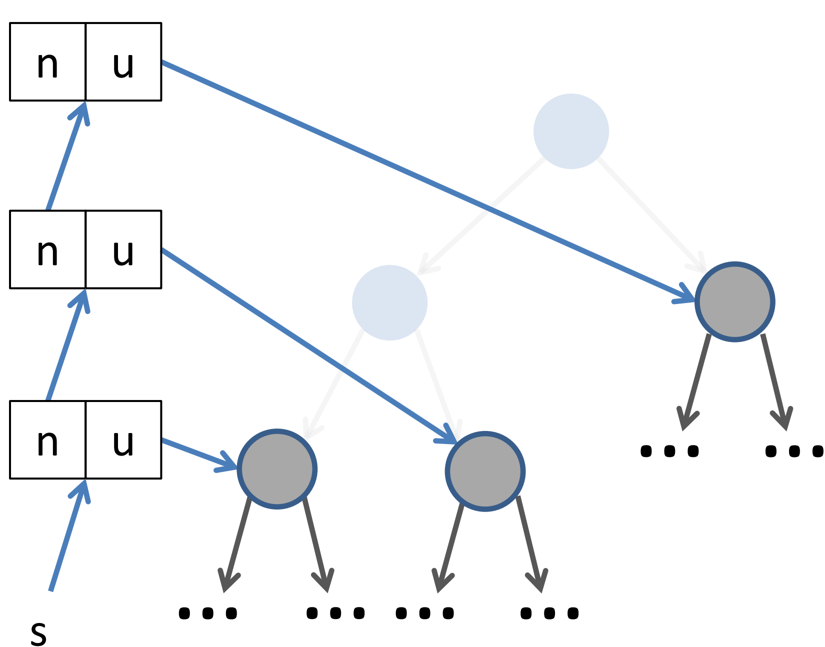
Outline

- Case study: High-level synthesis of dynamic data structures
- Challenge
- Motivating example
- **Leveraging recent advances in software verification**
- Implementation and results
- Outlook

Imperial College London

Classical first order logic

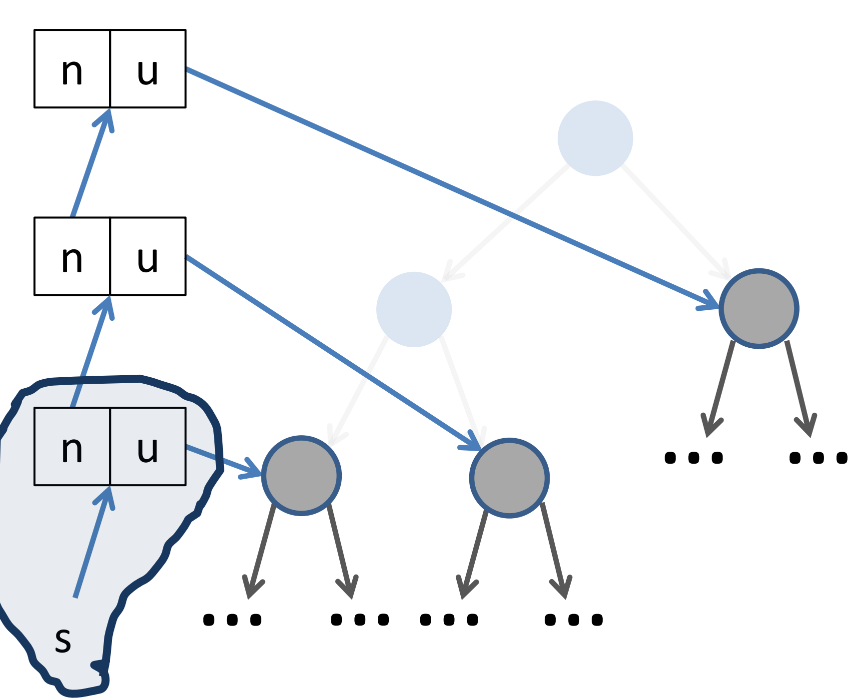
Describe heap layout with formulae



Imperial College London

Classical first order logic

Describe heap layout with formulae

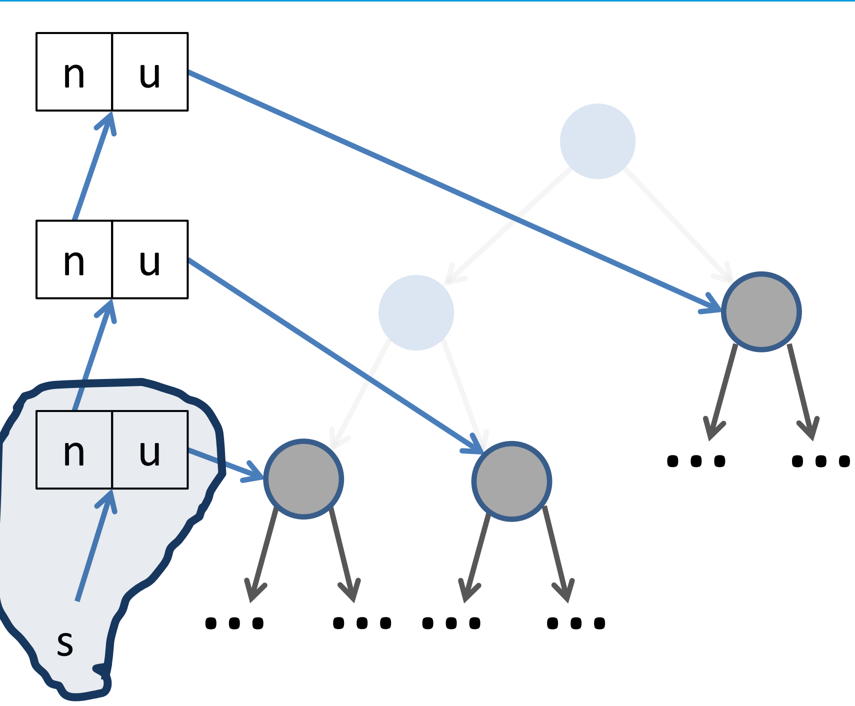


$$s \to [u\colon u'_1, n\colon s'_1]$$

Imperial College London

Classical first order logic

Describe heap layout with formulae



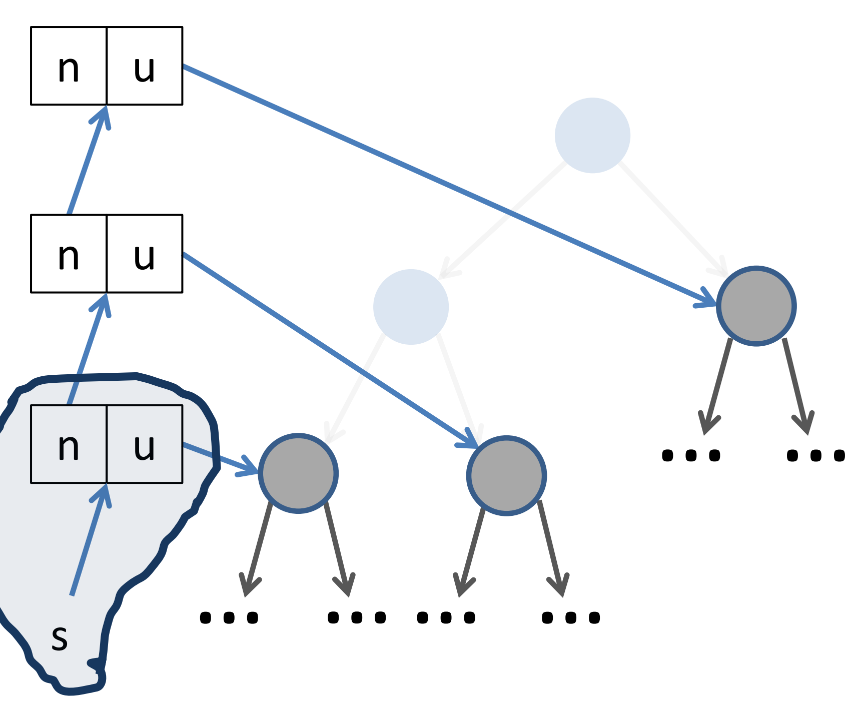
$s \to [u{:}\, u_1', n{:}\, s_1']$

"s points to

Imperial College London

Classical first order logic

Describe heap layout with formulae



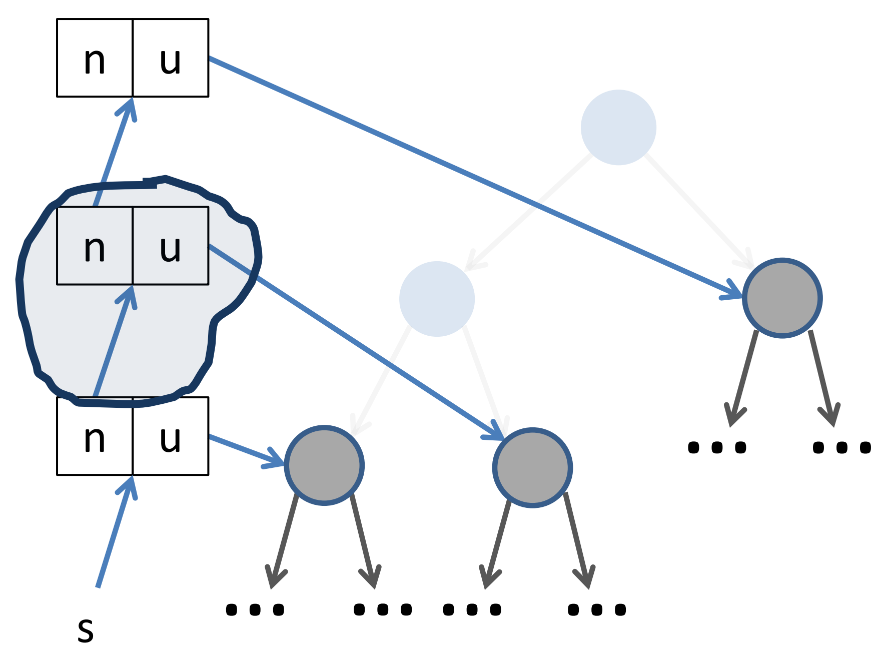
$s \rightarrow [u{:}\,u_1', n{:}\,s_1']$

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

Imperial College London

Classical first order logic

Describe heap layout with formulae

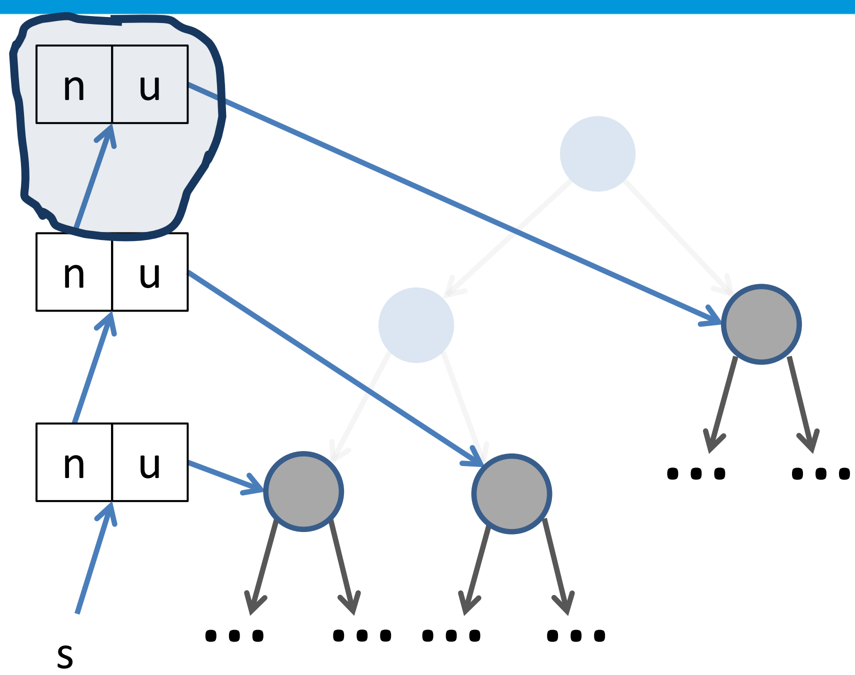


$$s \rightarrow [u\colon u_1', n\colon s_1'] \ \wedge \ s_1' \rightarrow [u\colon u_2', n\colon s_2']$$

Imperial College London

Classical first order logic

Describe heap layout with formulae

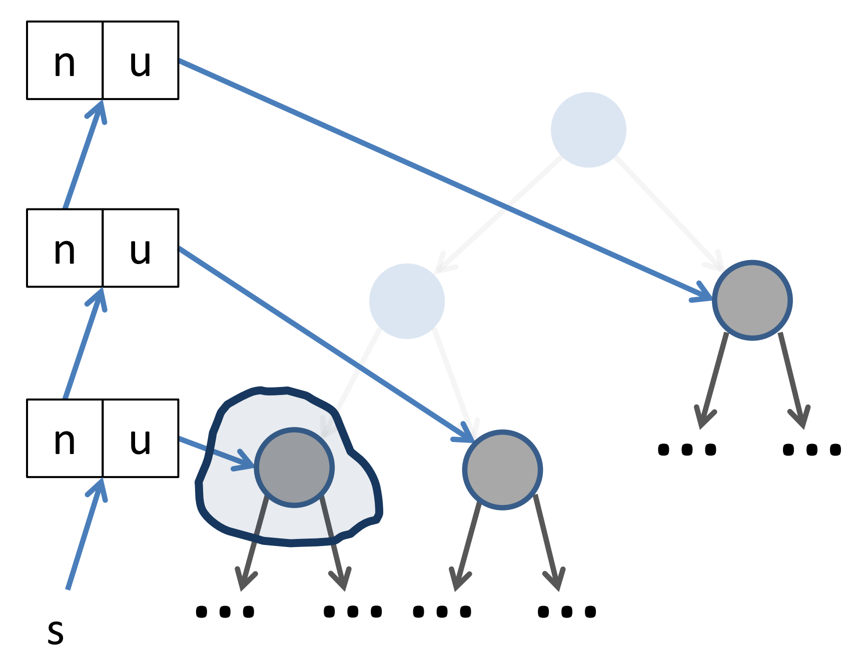


$$s \to [u{:}\, u_1', n{:}\, s_1'] \;\wedge\; s_1' \to [u{:}\, u_2', n{:}\, s_2'] \;\wedge\; s_2' \to [u{:}\, u_3', n{:}\, 0]$$

Imperial College London

Classical first order logic

Describe heap layout with formulae

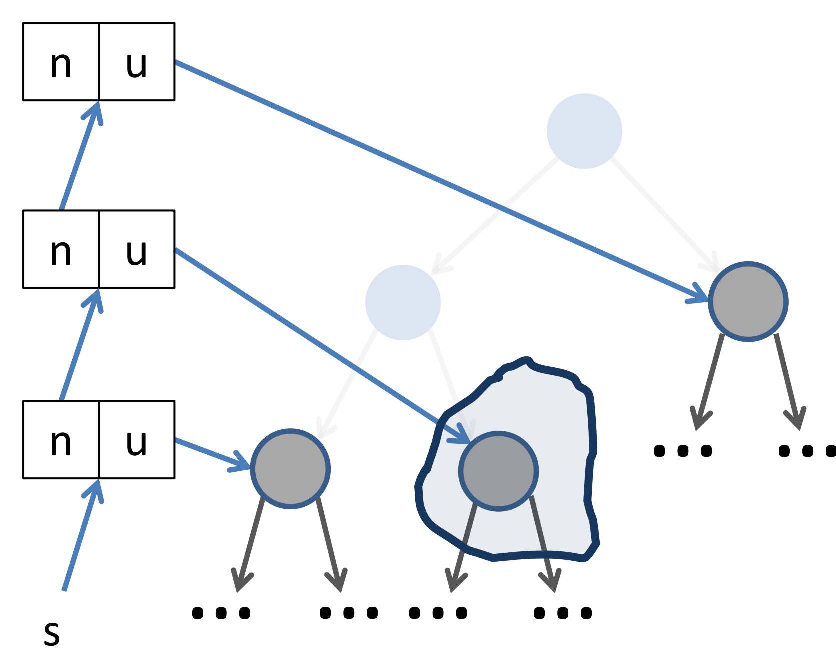


$$s \rightarrow [u\colon u_1', n\colon s_1'] \;\wedge\; s_1' \rightarrow [u\colon u_2', n\colon s_2'] \;\wedge\; s_2' \rightarrow [u\colon u_3', n\colon 0]$$

$$\wedge\; u_1' \rightarrow [l\colon u_4', r\colon u_5']$$

Classical first order logic

Describe heap layout with formulae

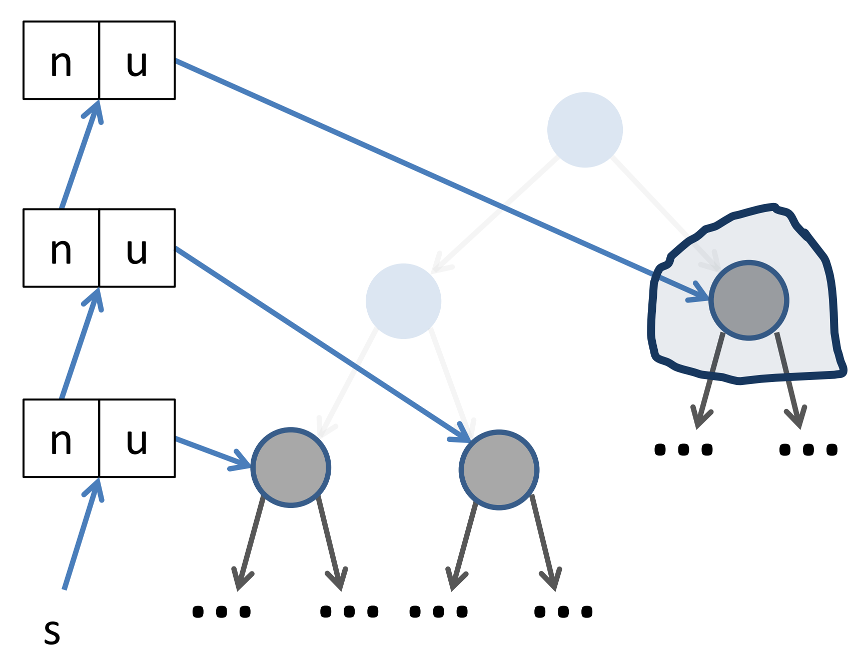


$$s \rightarrow [u{:}\, u_1', n{:}\, s_1'] \;\wedge\; s_1' \rightarrow [u{:}\, u_2', n{:}\, s_2'] \;\wedge\; s_2' \rightarrow [u{:}\, u_3', n{:}\, 0]$$

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

Classical first order logic

Describe heap layout with formulae

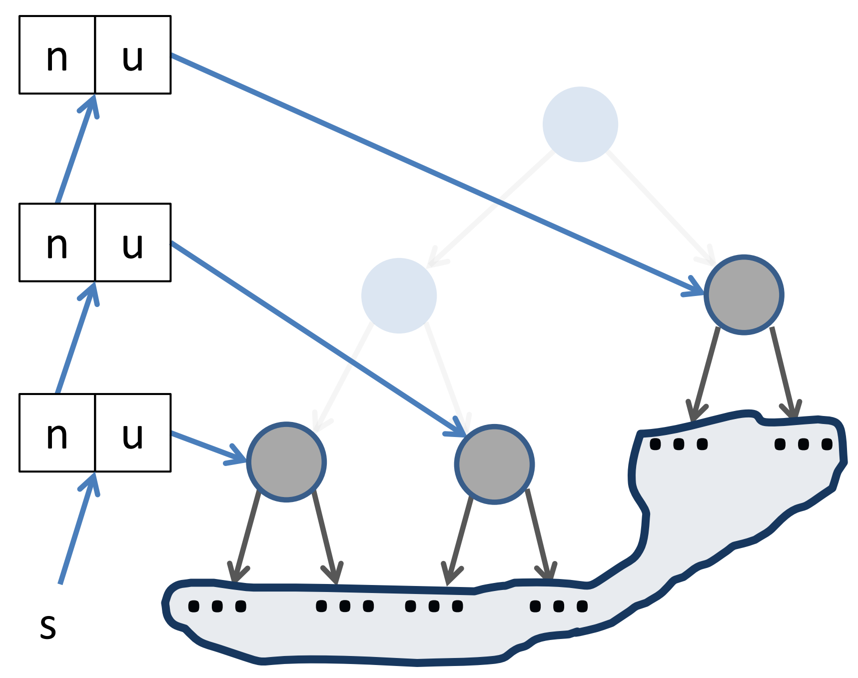


$$s \to [u{:}\, u_1', n{:}\, s_1'] \;\wedge\; s_1' \to [u{:}\, u_2', n{:}\, s_2'] \;\wedge\; s_2' \to [u{:}\, u_3', n{:}\, 0]$$
$$\wedge\; u_1' \to [l{:}\, u_4', r{:}\, u_5'] \;\wedge\; u_3' \to [l{:}\, u_8', r{:}\, u_9'] \;\wedge\; u_2' \to [l{:}\, u_6', r{:}\, u_7']$$

Imperial College London

Classical first order logic

Describe heap layout with formulae



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

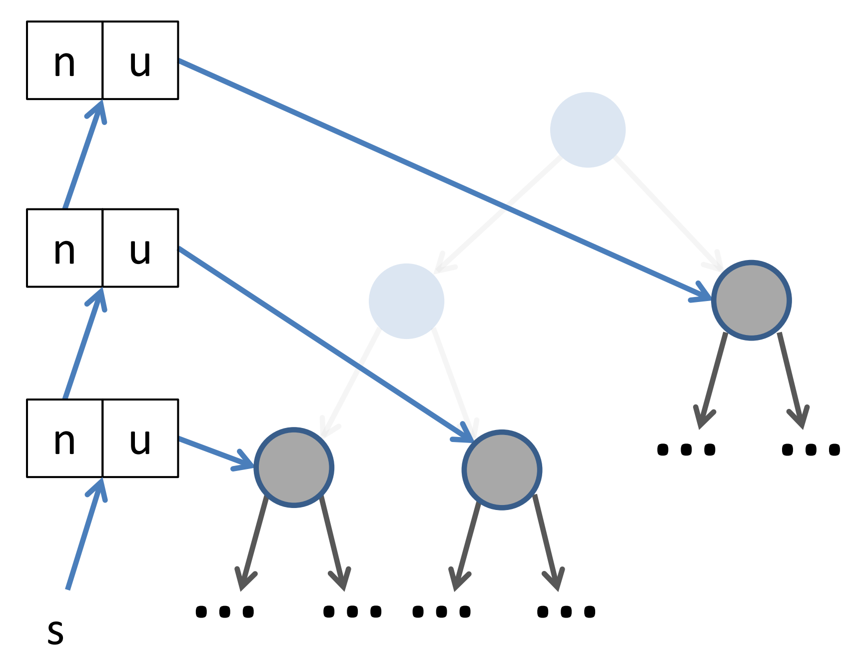
$$\wedge\; u_1' \rightarrow [l{:}\, u_4', r{:}\, u_5'] \;\wedge\; u_3' \rightarrow [l{:}\, u_8', r{:}\, u_9'] \;\wedge\; u_2' \rightarrow [l{:}\, u_6', r{:}\, u_7']$$

$$\wedge\; \ldots$$

Imperial College London

Classical first order logic

Describe heap layout with formulae



Conjunction '∧' does not rule out aliasing!

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

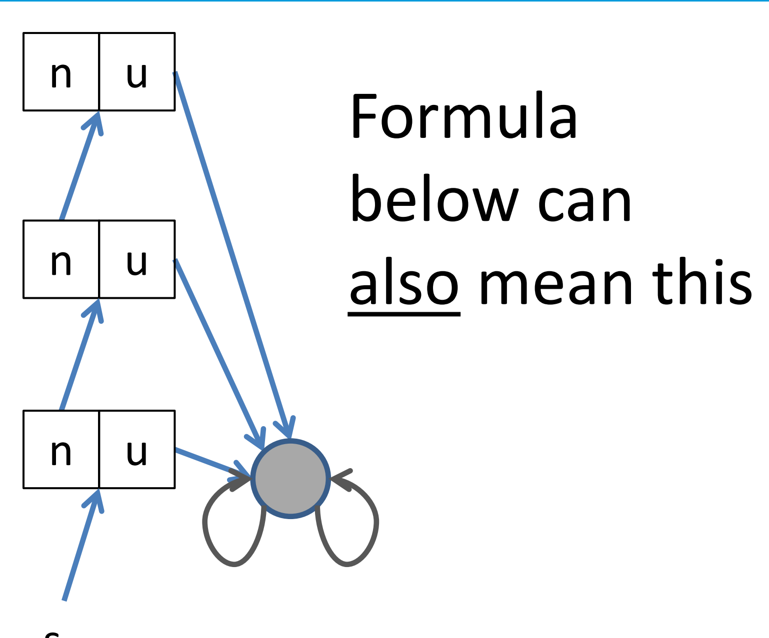
$\wedge\, u_1' \rightarrow [l\!: u_4', r\!: u_5'] \wedge u_3' \rightarrow [l\!: u_8', r\!: u_9'] \wedge u_2' \rightarrow [l\!: u_6', r\!: u_7']$

$\wedge \ldots$

Imperial College London

Classical first order logic

Describe heap layout with formulae



Formula below can also mean this

Conjunction '∧' does not rule out aliasing!

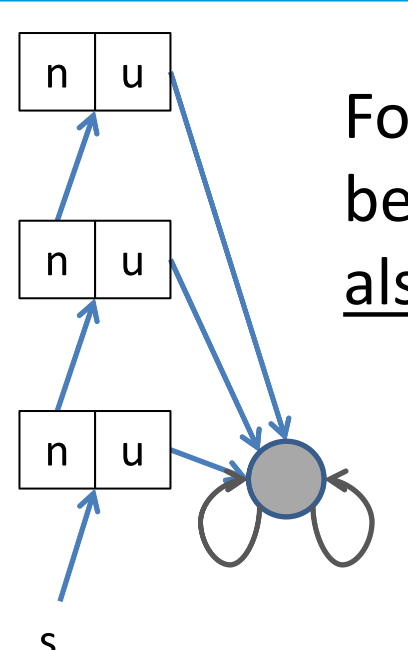
$$s \rightarrow [u{:}\, u_1', n{:}\, s_1'] \;\wedge\; s_1' \rightarrow [u{:}\, u_2', n{:}\, s_2'] \;\wedge\; s_2' \rightarrow [u{:}\, u_3', n{:}\, 0]$$

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

$\wedge$ ...

Classical first order logic

Describe heap layout with formulae



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

Conjunction

All u-pointers alias
$u_1' = u_2' = u_3' = u_4' =$
$u_5' = u_6' = u_7' = u_8' = u_9' = \cdots$

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

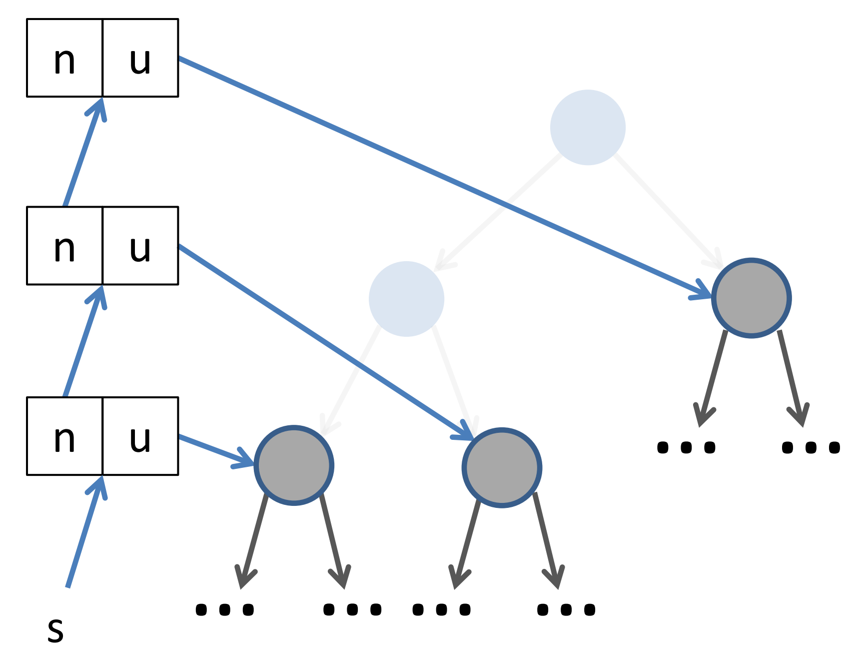
$$\wedge\; u_1' \to [l{:}\, u_4', r{:}\, u_5'] \;\wedge\; u_3' \to [l{:}\, u_8', r{:}\, u_9'] \;\wedge\; u_2' \to [l{:}\, u_6', r{:}\, u_7']$$

$$\wedge\; \ldots$$

Imperial College London

Classical first order logic

Describe heap layout with formulae



Conjunction '∧' does not rule out aliasing!

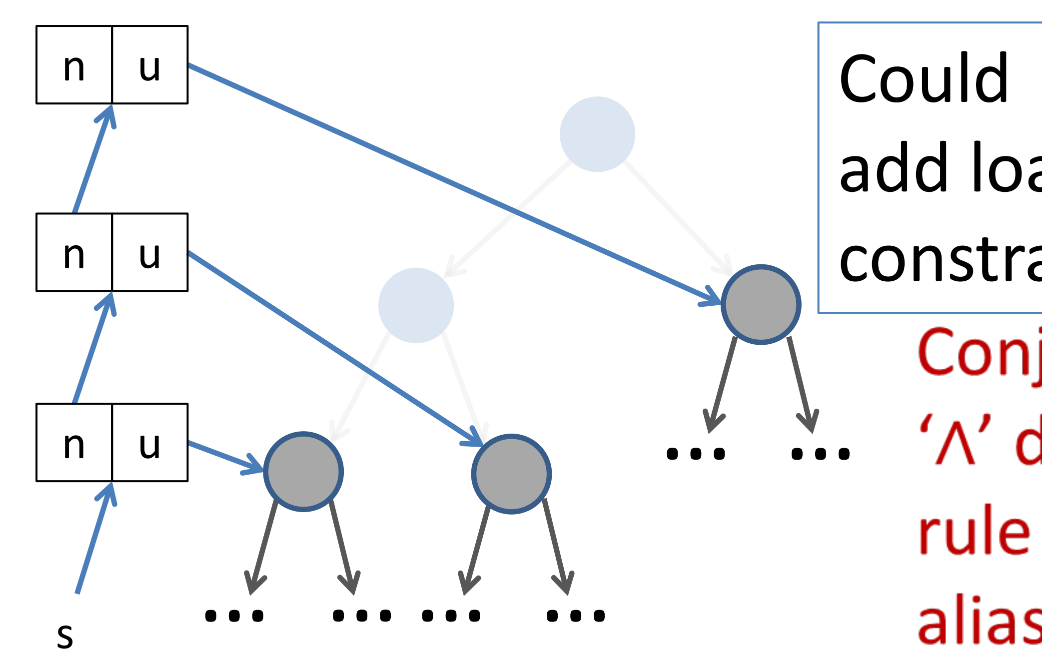
$$s \rightarrow [u{:}\, u_1', n{:}\, s_1'] \ \wedge\ s_1' \rightarrow [u{:}\, u_2', n{:}\, s_2'] \ \wedge\ s_2' \rightarrow [u{:}\, u_3', n{:}\, 0]$$

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

$$\wedge\ \ldots$$

Classical first order logic

Describe heap layout with formulae



Could add loads of constraints

Conjunction '∧' does not rule out aliasing!

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

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

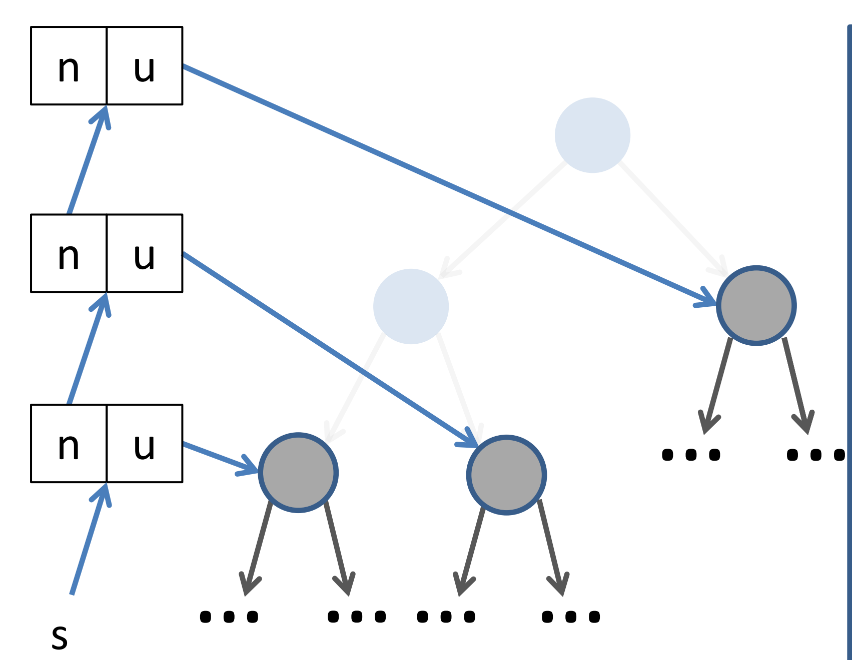
$$\wedge\; \ldots \wedge u_1' \neq u_2' \wedge u_1 \neq u_3' \wedge u_1' \neq u_4' \wedge u_1' \neq u_5' \wedge u_1' \neq u_6' \wedge u_1' \neq u_7'$$

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

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

Separation logic

Describe heap layout with formulae



O'Hearn, Reynolds, Ishtiaq, Yang:

Separating conjunction '*'

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

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

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

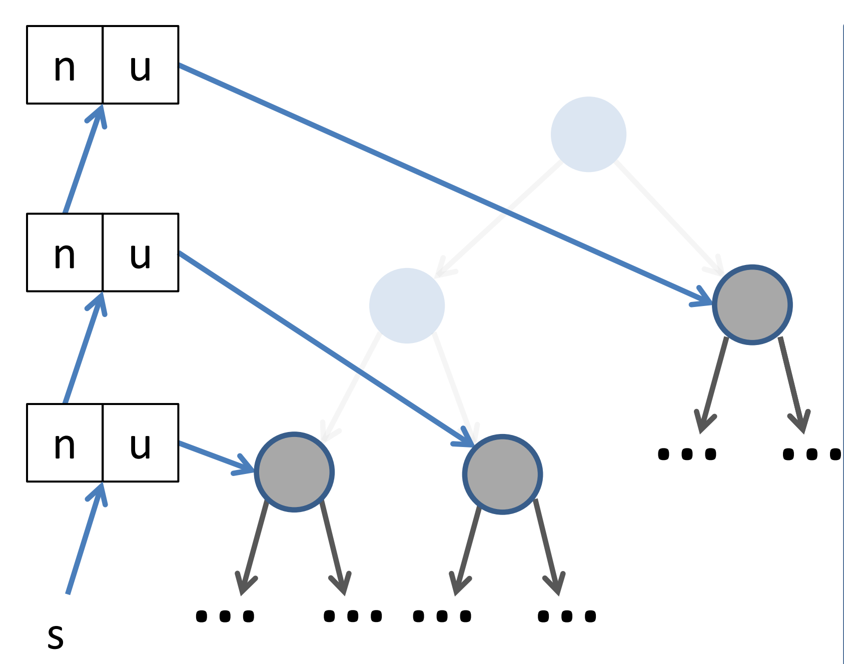
$$\wedge u_1' \neq u_8' \wedge u_1 \neq u_9' \wedge u_3' \neq u_4' \wedge u_3' \neq u_5' \wedge u_3' \neq u_6' \wedge u_3' \neq u_7'$$

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

Imperial College London

Separation logic

Describe heap layout with formulae



O'Hearn, Reynolds, Ishtiaq, Yang:

Separating conjunction '$*$' rules out aliasing!

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

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

$$\wedge\; \ldots \wedge u'_1 \neq u'_2 \wedge u_1 \neq u'_3 \wedge u'_1 \neq u'_4 \wedge u'_1 \neq u'_5 \wedge u'_1 \neq u'_6 \wedge u'_1 \neq u'_7$$

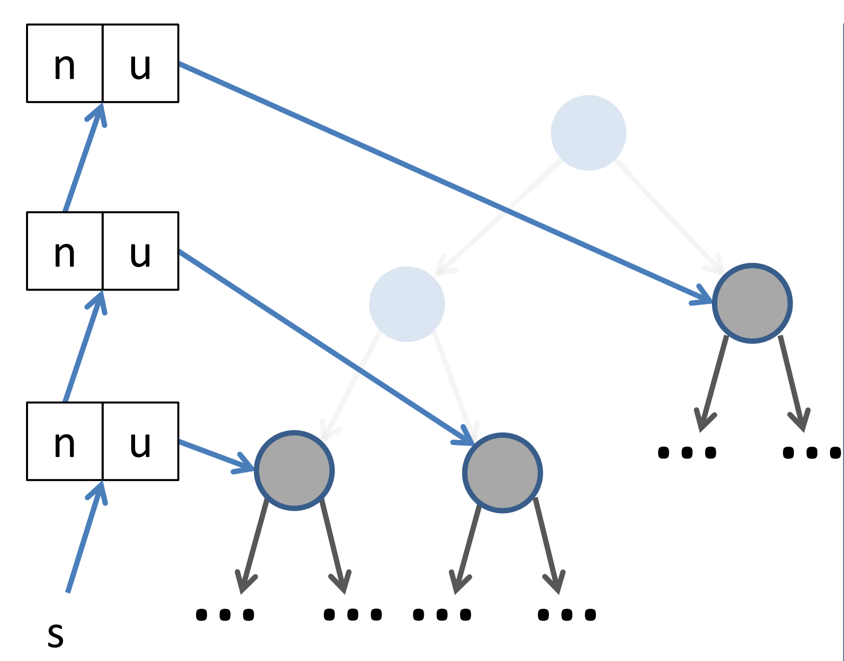
$$\wedge\, u'_1 \neq u'_8 \wedge u_1 \neq u'_9 \wedge u'_3 \neq u'_4 \wedge u'_3 \neq u'_5 \wedge u'_3 \neq u'_6 \wedge u'_3 \neq u'_7$$

$$\wedge\, u'_3 \neq u'_8 \wedge u_2 \neq u'_3 \wedge u'_2 \neq u'_4 \wedge u'_2 \neq u'_5 \wedge u'_2 \neq u'_6 \wedge \ldots$$

Imperial College London

Separation logic

Describe heap layout with formulae



O'Hearn, Reynolds, Ishtiaq, Yang:

Separating conjunction '∗' rules out aliasing!

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

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

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

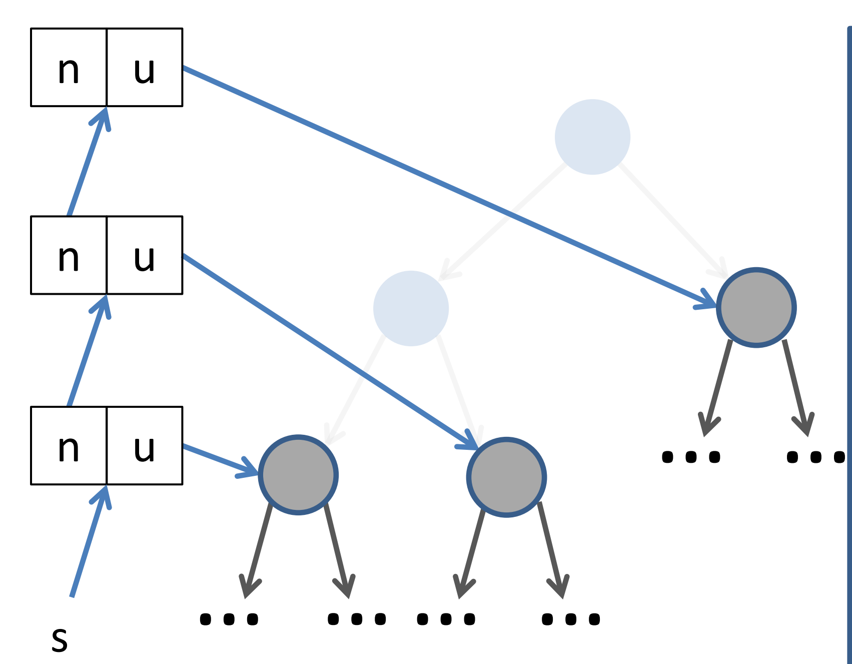
$$\wedge\, u_1' \neq u_8' \wedge u_1 \neq u_9' \wedge u_3' \neq u_4' \wedge u_3' \neq u_5' \wedge u_3' \neq u_6' \wedge u_3' \neq u_7'$$

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

Imperial College London

Separation logic

Describe heap layout with formulae



O'Hearn, Reynolds, Ishtiaq, Yang:

Separating conjunction '*' rules out aliasing!

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

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

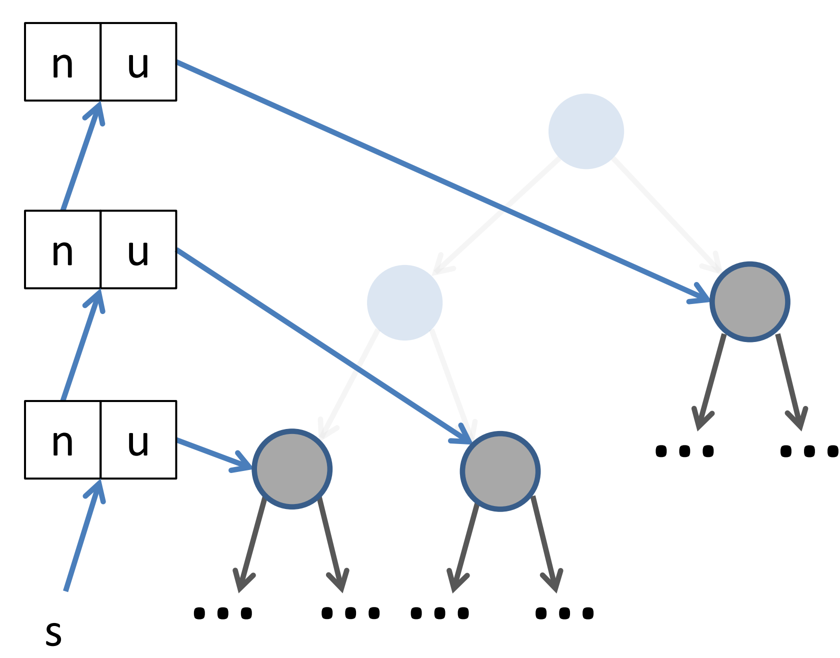
$$\wedge \;\ldots \wedge u_1' \neq u_2' \wedge u_1 \neq u_3' \wedge u_1' \neq u_4' \wedge u_1' \neq u_5' \wedge u_1' \neq u_6' \wedge u_1' \neq u_7'$$

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

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

Separation logic

Describe heap layout with formulae



O'Hearn, Reynolds, Ishtiaq, Yang:

Separating conjunction '*' rules out aliasing!

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

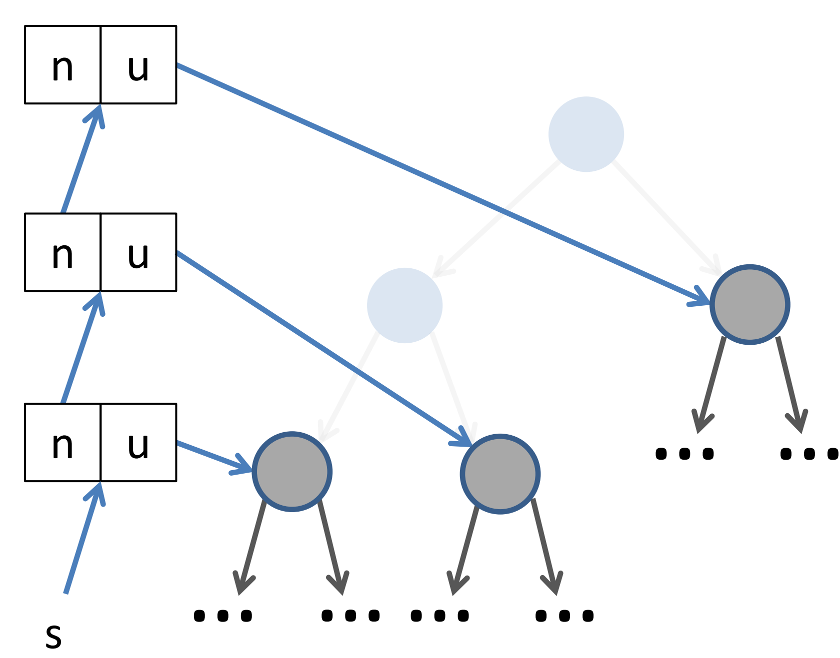
$$*\; u_1' \rightarrow [l{:}\, u_4', r{:}\, u_5'] \;*\; u_3' \rightarrow [l{:}\, u_8', r{:}\, u_9'] \;*\; u_2' \rightarrow [l{:}\, u_6', r{:}\, u_7']$$

- Tractable heap analysis – very popular in SW verification

Imperial College London

Separation logic

Describe heap layout with formulae



O'Hearn, Reynolds, Ishtiaq, Yang:

Separating conjunction '*' rules out aliasing!

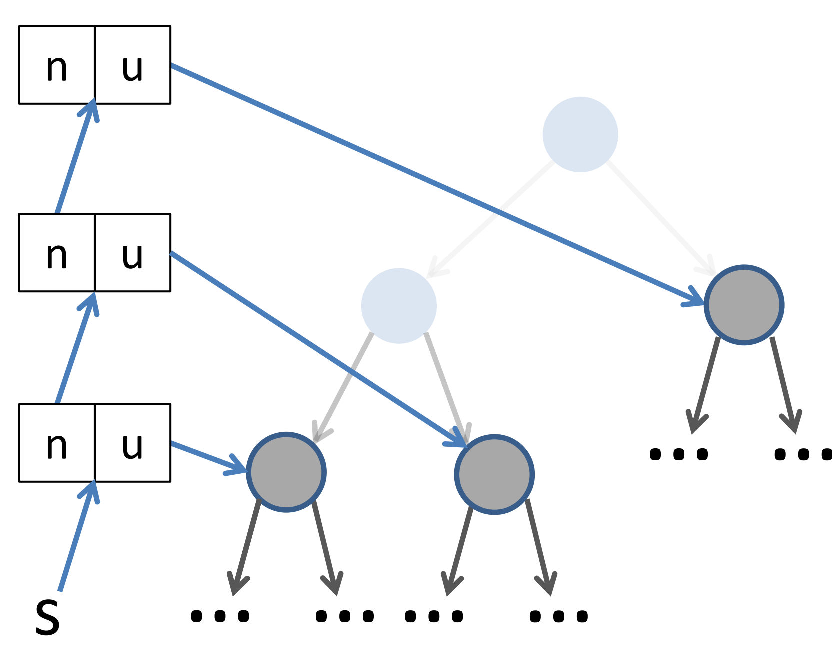
$$s \to [u\colon u_1', n\colon s_1'] \;*\; s_1' \to [u\colon u_2', n\colon s_2'] \;*\; s_2' \to [u\colon u_3', n\colon 0]$$

$$*\; u_1' \to [l\colon u_4', r\colon u_5'] \;*\; u_3' \to [l\colon u_8', r\colon u_9'] \;*\; u_2' \to [l\colon u_6', r\colon u_7']$$

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

Imperial College London

Back to our partitioning task

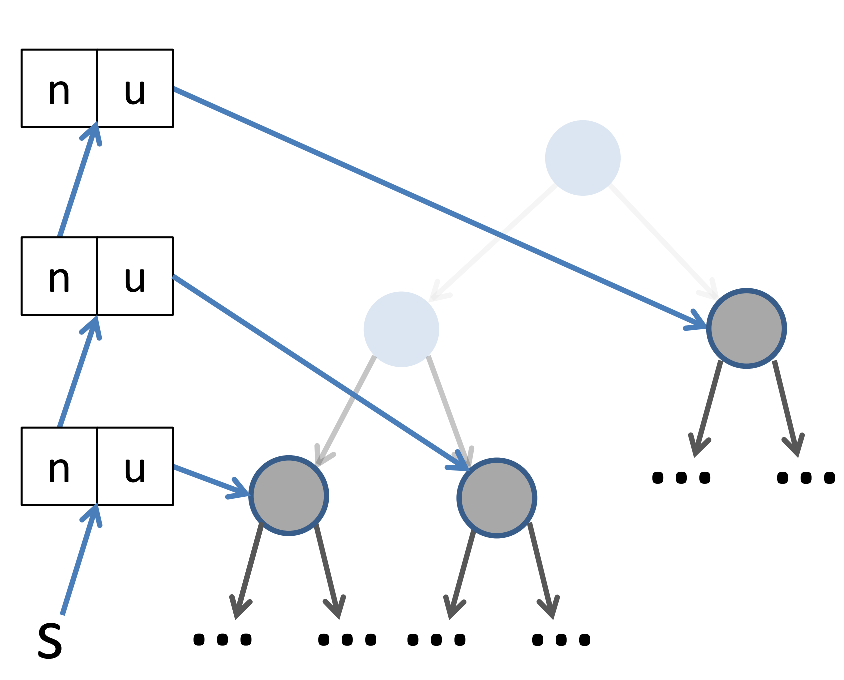


$$s \rightarrow [u{:}\,u_1', n{:}\,s_1'] \;*\; s_1' \rightarrow [u{:}\,u_2', n{:}\,s_2'] \;*\; s_2' \rightarrow [u{:}\,u_3', n{:}\,0]$$

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

Imperial College London

Back to our partitioning task

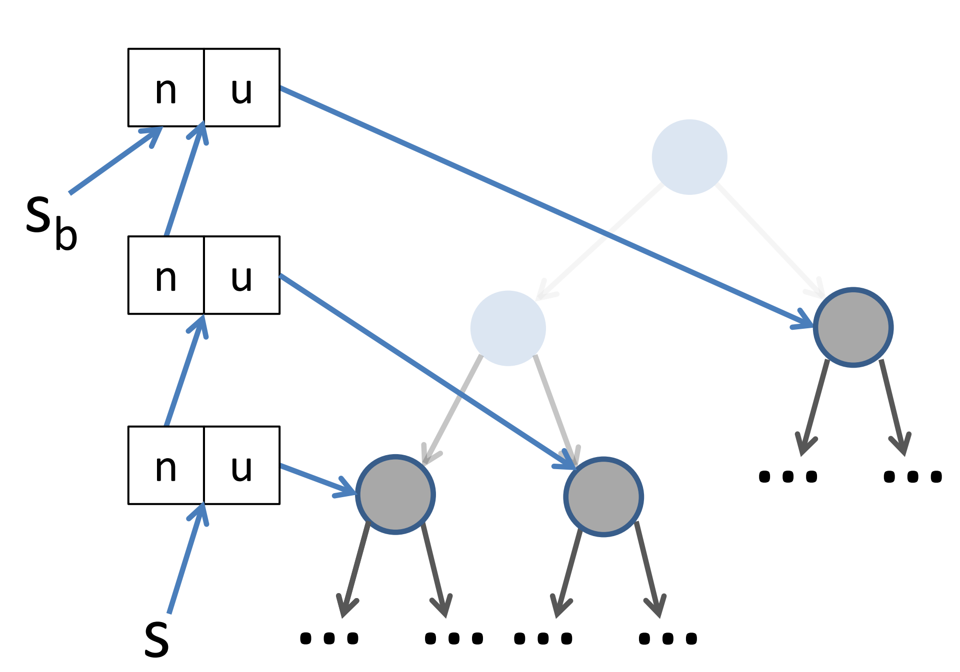


- Partitioning the heap = partitioning the formula describing it

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

Imperial College London

Back to our partitioning task

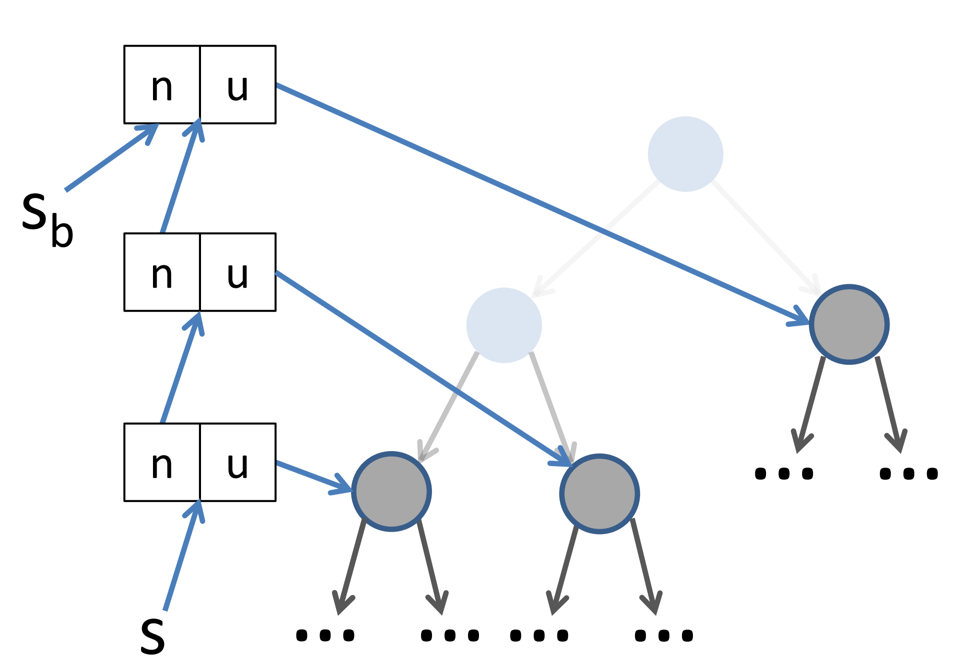


- Partitioning the heap = partitioning the formula describing it
- Add a second 'hook' into the data structure

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

Imperial College London

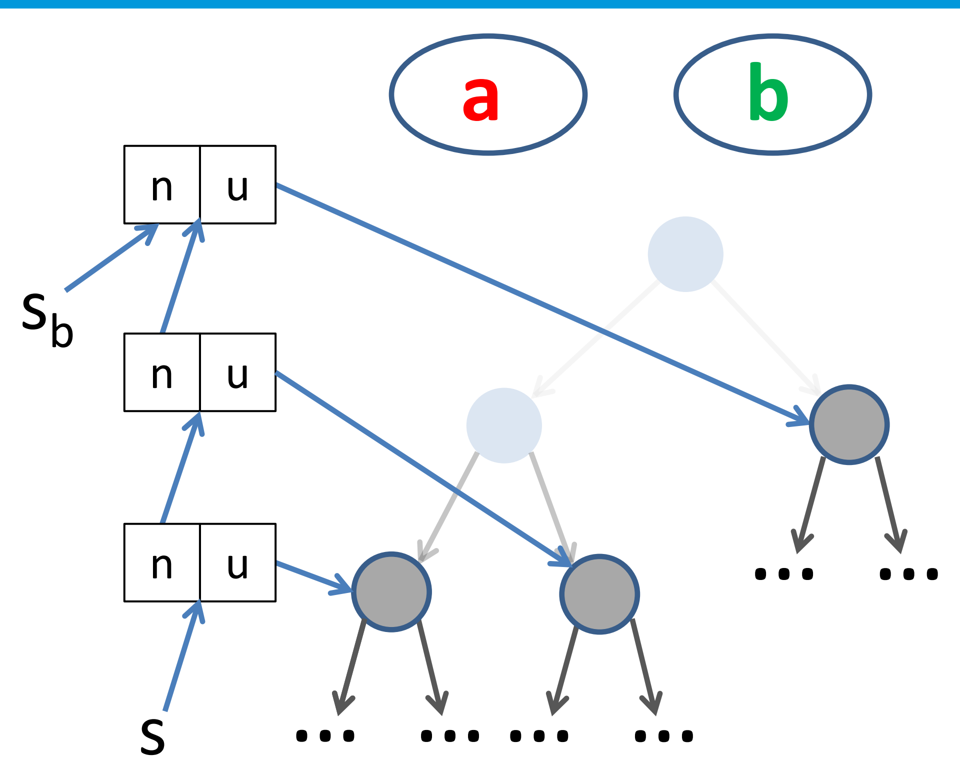
Back to our partitioning task



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

$$s \to [u\colon u_1', n\colon s_1'] \ * \ s_1' \to [u\colon u_2', n\colon s_2'] \ * \ s_2' \to [u\colon u_3', n\colon 0]$$
$$* \ u_1' \to [l\colon u_4', r\colon u_5'] \ * \ u_3' \to [l\colon u_8', r\colon u_9'] \ * \ u_2' \to [l\colon u_6', r\colon u_7']$$

Back to our partitioning task



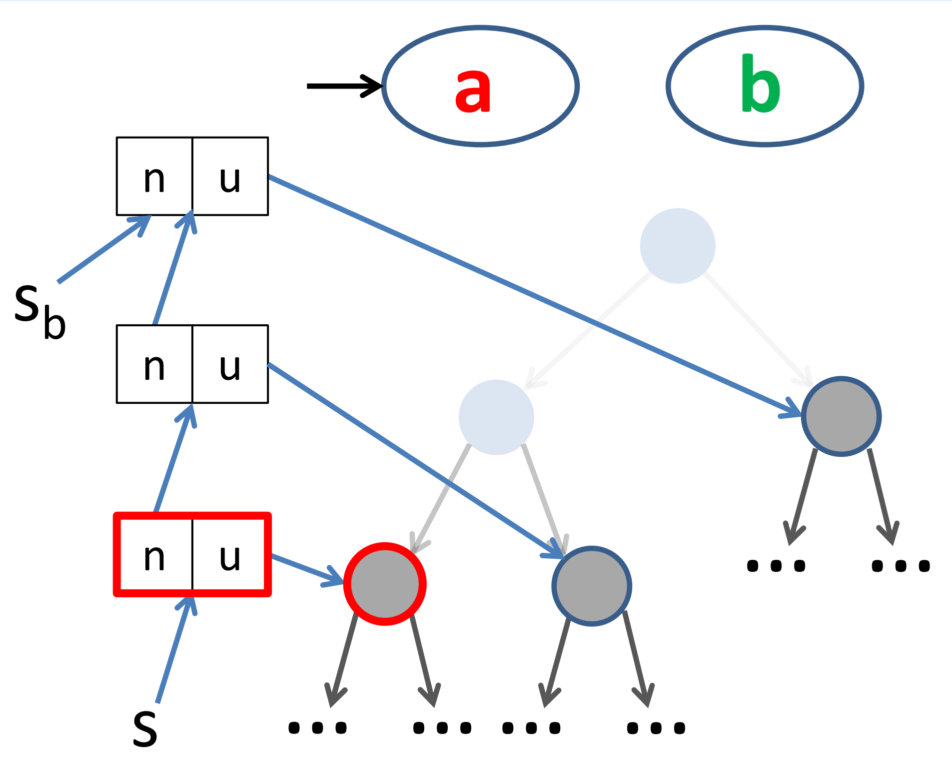
- Partitioning the heap = partitioning the formula describing it
- Add a second 'hook' into the data structure
- 'Symbolically' step through loop iterations

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

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

Imperial College London

Back to our partitioning task



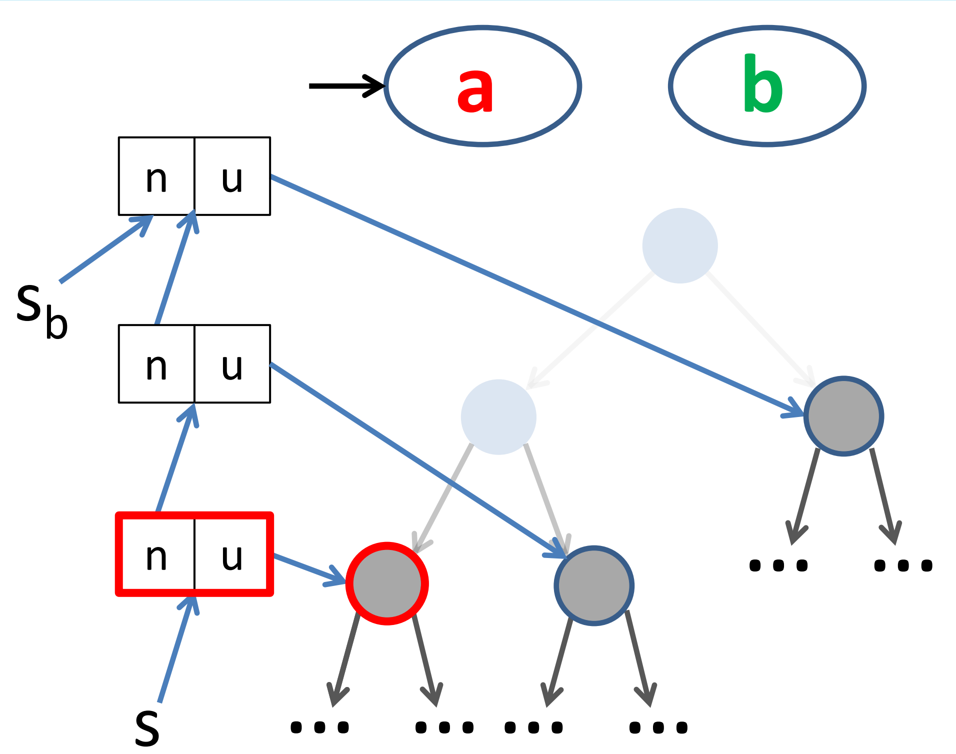
- Partitioning the heap = partitioning the formula describing it
- Add a second 'hook' into the data structure
- 'Symbolically' step through loop iterations

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

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

Imperial College London

Back to our partitioning task



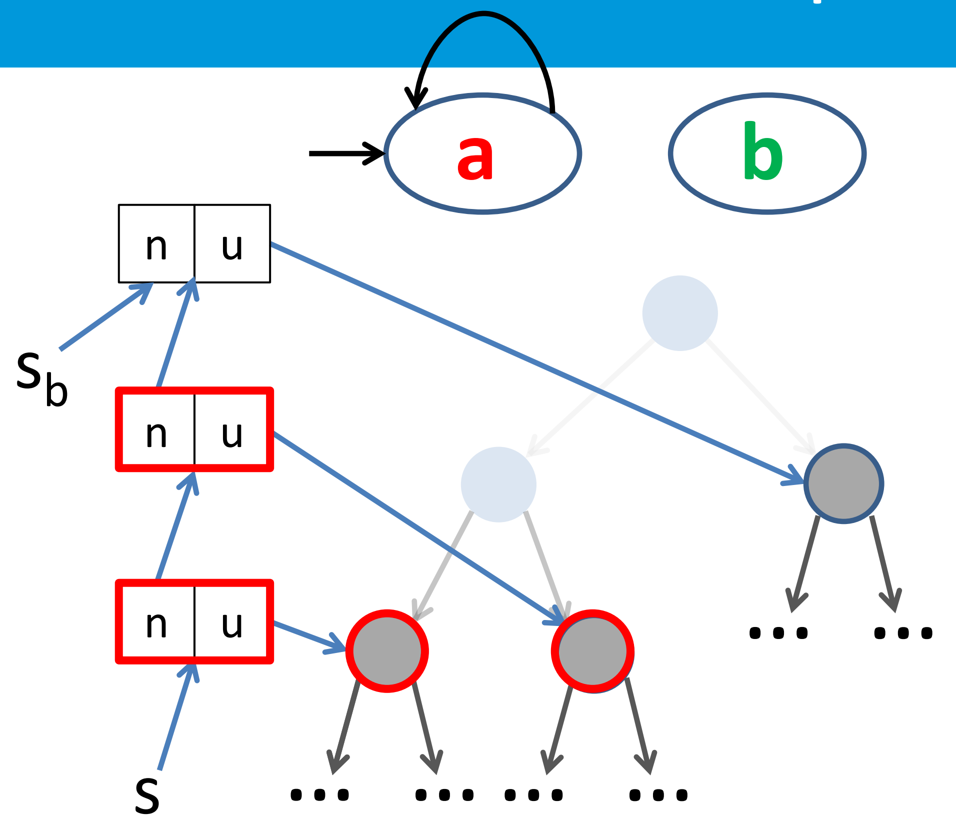
- Partitioning the heap = partitioning the formula describing it
- Add a second 'hook' into the data structure
- 'Symbolically' step through loop iterations
- Attach labels to heaplets

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

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

Imperial College London

Back to our partitioning task



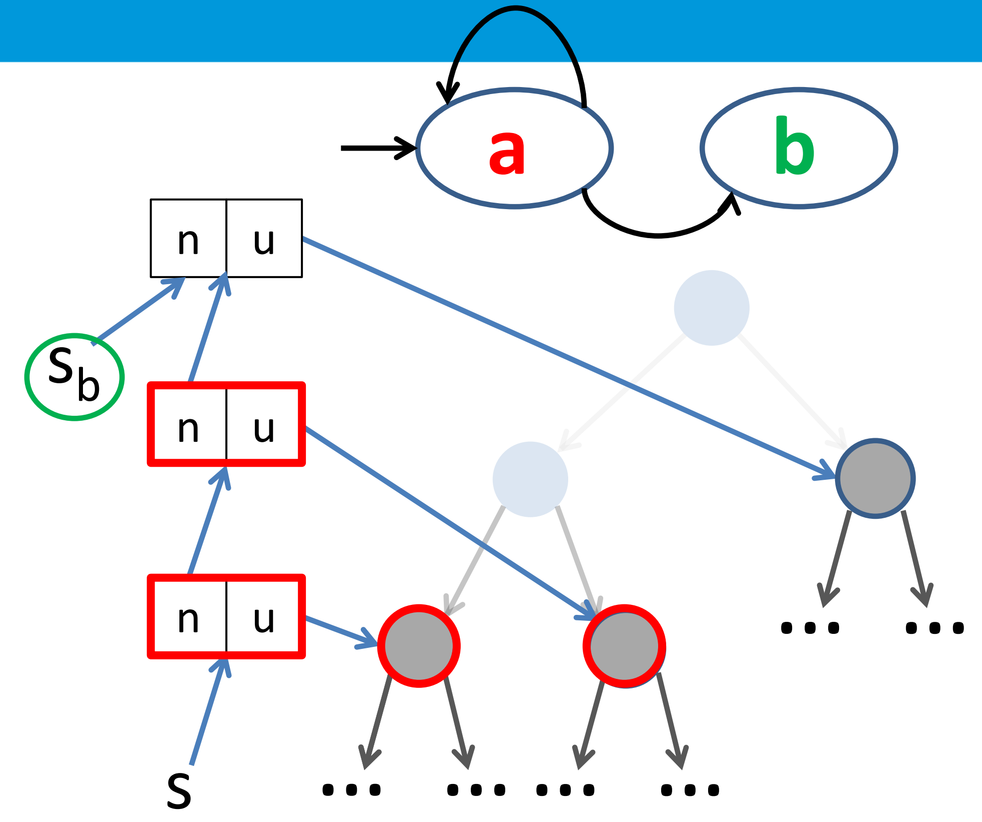
- Partitioning the heap = partitioning the formula describing it
- Add a second 'hook' into the data structure
- 'Symbolically' step through loop iterations
- Attach labels to heaplets

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

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

Imperial College London

Back to our partitioning task



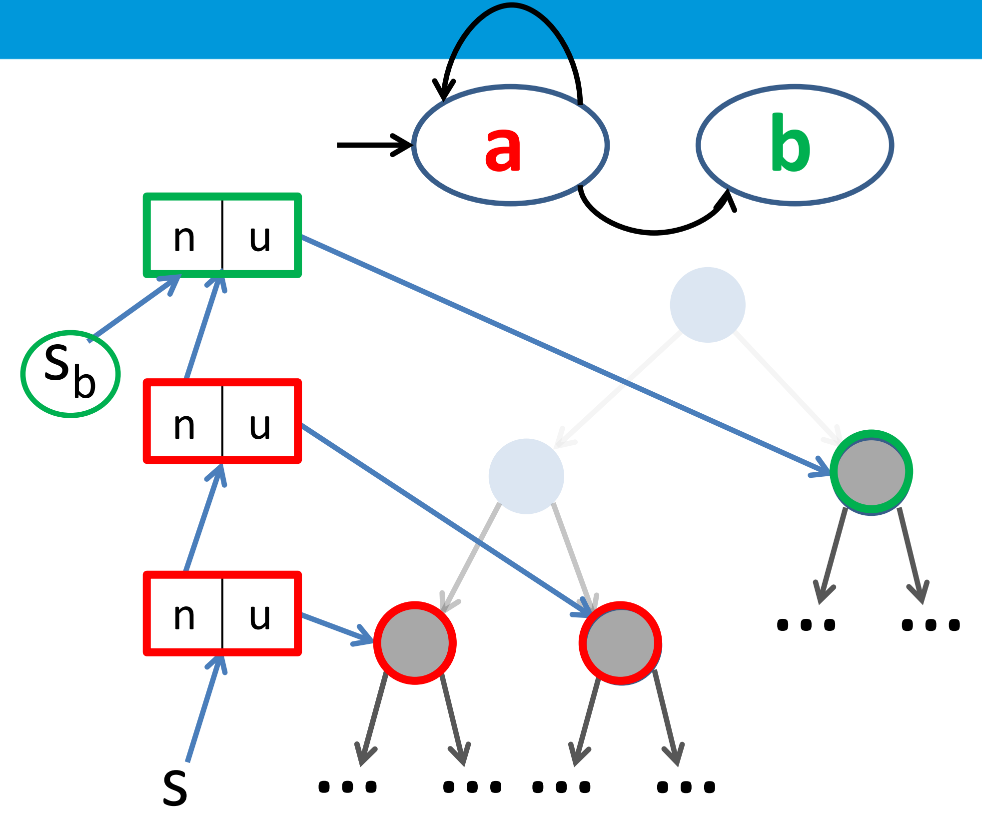
- Partitioning the heap = partitioning the formula describing it
- Add a second ‘hook’ into the data structure
- ‘Symbolically’ step through loop iterations
- Attach labels to heaplets

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

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

Imperial College London

Back to our partitioning task



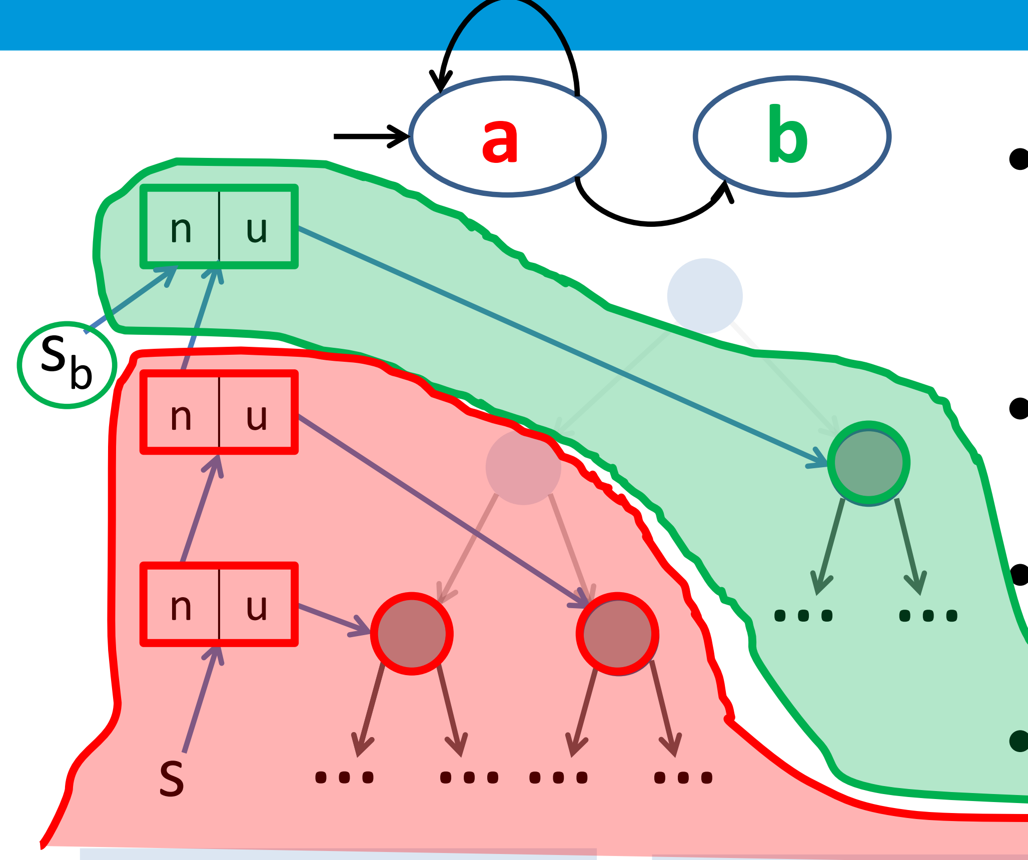
- Partitioning the heap = partitioning the formula describing it
- Add a second ‘hook’ into the data structure
- ‘Symbolically’ step through loop iterations
- Attach labels to heaplets

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

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

Imperial College London

Back to our partitioning task

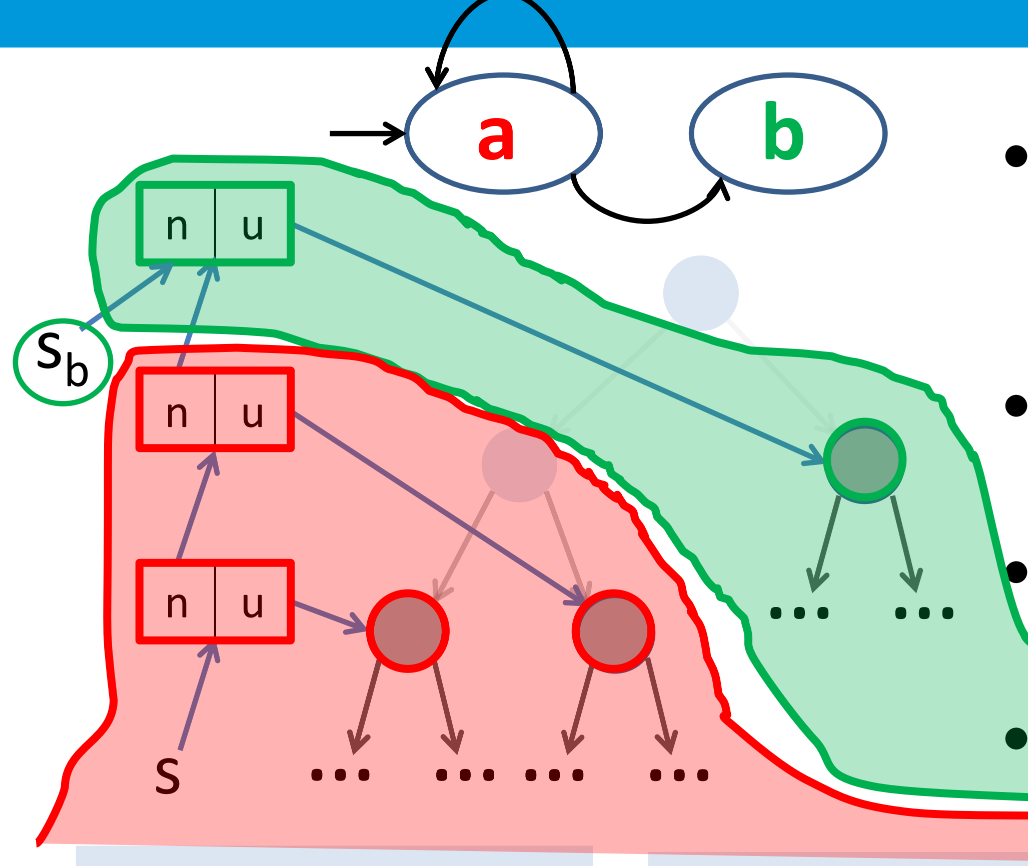


- Partitioning the heap = partitioning the formula describing it
- Add a second 'hook' into the data structure
- 'Symbolically' step through loop iterations
- Attach labels to heaplets

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

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

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 …

Imperial College London

Symbolic execution

- Capture the semantics of a program in terms of heap access
- Specify semantics of program commands
- Examples:

$\{x = y'\}\ \boldsymbol{x := 3}\ \{x = 3\}$ assignment

$\{x \to y'\}\ \boldsymbol{[x] := 3}\ \{x \to 3\}$ heap assignment

$\{emp\}\ \boldsymbol{new(x)}\ \{x \to y'\}$ allocation

Symbolic execution

Propagate formulae through CFG:

$\{emp\}$

l1: new(x);

$\{x \rightarrow y'\}$

l2: [x] = 3;

$\{x \rightarrow 3\}$

l3: …

Loops

$\{pre_0\}$

l1: while *b* do

$\{pre_0 \wedge b\}$

l1: new(x);

l2: [x] = 3;

l3: …

$\{post_0\}$

l4: end while

Imperial College London

Loops

- Can’t unroll ALL iterations during analysis
  data-dependent loop condition *b*

$\{pre_0\}$

l1: while *b* do

$\{pre_1 \wedge b\}$

l1: new(x);

l2: [x] = 3;

l3: …

$\{post_1\}$

l4: end while

Loops

- Can't unroll ALL iterations during analysis
  data-dependent loop condition *b*
- Instead:
  Symb. execute iterations until a fix-point can be established (Magill 2006)

$$post_i \Rightarrow post_{i-1}$$

- Theorem prover to decide

$\{pre_0\}$

```
l1:    while b do
```

$\{pre_1 \wedge b\}$

```
l1:        new(x);
l2:        [x] = 3;
l3:        …
```

$\{post_1\}$

```
l4:    end while
```

Imperial College London

Outline

- Case study: High-level synthesis of dynamic data structures
- Challenge
- Motivating example
- Leveraging recent advances in software verification
- **Implementation and results**
- Outlook

Imperial College London

Implementation

Automated source-to-source compiler
LLNL ROSE Compiler Infrastructure (C++)

Parse into AST

Unparse AST

C/C++ code → Analysis interface → Substitute dynamic memory allocation → Heap splitting / Loop splitting → C/C++ code → Vivado HLS → RTL impl.

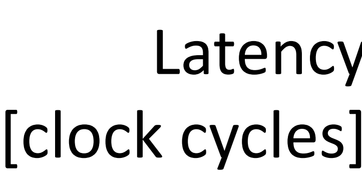
Analysis interface → Heap analysis / Theorem proving → Heap splitting / Loop splitting

Automated proof engine (OCaml)

Imperial College London

Case Studies (I)

Tree-based *K*-means clustering



x 10^5

12
10
8
6
4
2
0

no parallelization,
only ensuring synthesizability

parallelization p=2

parallelization p=4

0 2000 4000 6000 8000 10000

Slices

Imperial College London

Case Studies (II)

		P	Slices	Clock	Cycles
1	**Merger (linked lists)**				
	Baseline (no par.)	1	574	9.0 ns	21167k
	Autom. Parallelization	4	965	8.7 ns	5483k
2	**Tree deletion (tree, linked list)**				
	Baseline (no par.)	1	1521	5.2 ns	901k
	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
	Autom. Parallelization	2	5618	7.0 ns	606k

Imperial College London

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	

Imperial College London

Case Studies (II)

	P	Slices	Clock	Cycles
1 Merger (linked lists)				
Baseline (no par.)	1	574	9.0 ns	21167k
Autom. Parallelization	4	965	8.7 ns	5483k
2 Tree deletion (tree, linked list)				
Baseline (no par.)	1	1521	5.2 ns	901k
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
Autom. Parallelization	2	5618	7.0 ns	606k

x4

x2

x2

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

	P	Slices	Clock	Cycles
Hand-optimized HLS	2	5492	5.5 ns	165k

Imperial College London

Case Studies (II)

	P	Slices	Clock	Cycles
1 Merger (linked lists)				
Baseline (no par.)	1	574	9.0 ns	21167k
Autom. Parallelization	4	965	8.7 ns	5483k
2 Tree deletion (tree, linked list)				
Baseline (no par.)	1	1521	5.2 ns	901k
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
Autom. Parallelization	2	5618	7.0 ns	606k
Hand-optimized HLS	2	5492	5.5 ns	165k

Merger: x4

Tree deletion: x2

K-means: x2

Manual loop flattening, pipelining, custom bit widths, data streaming directives, data… x3.6

Imperial College London

Outline

- Case study: High-level synthesis of dynamic data structures
- Challenge
- Motivating example
- Leveraging recent advances in software verification
- Implementation and results
- **Outlook**

Imperial College London

What else?

1. Generating application-specific multi-cache architectures
2. Loop transformations for pipelining
3. Worst-case/ average case bounds on heap usage

Generate multi-cache arch

Imperial College London

Generate multi-cache arch

Off-chip memory

arb

cache1

cache2

FU1

FU2

FPGA

Imperial College London

Generate multi-cache arch

Off-chip memory

arb

Cache coherence

cache1

cache2

FU1

FU2

FPGA

Communication-free parallelism:

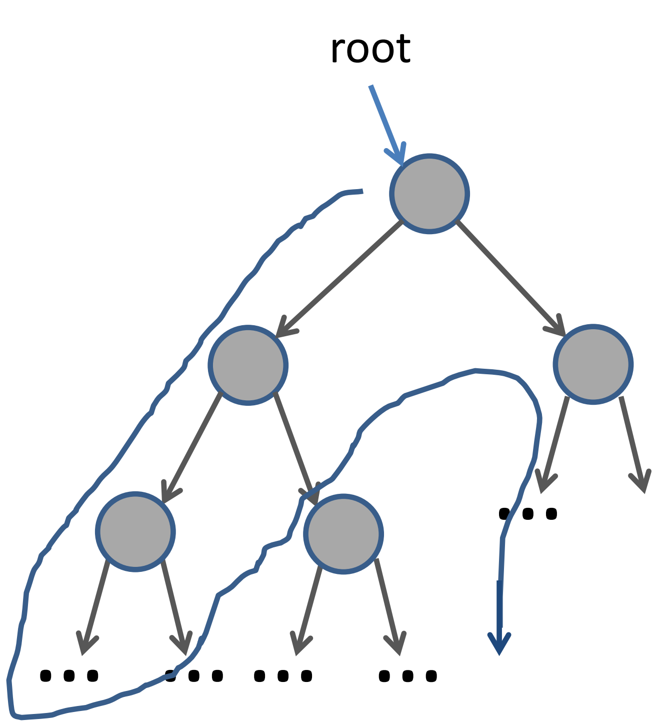
Caches are truly private

Imperial College London

What else?

1. Generating application-specific multi-cache architectures
2. Loop transformations for pipelining
3. Worst-case/ average case bounds on heap usage

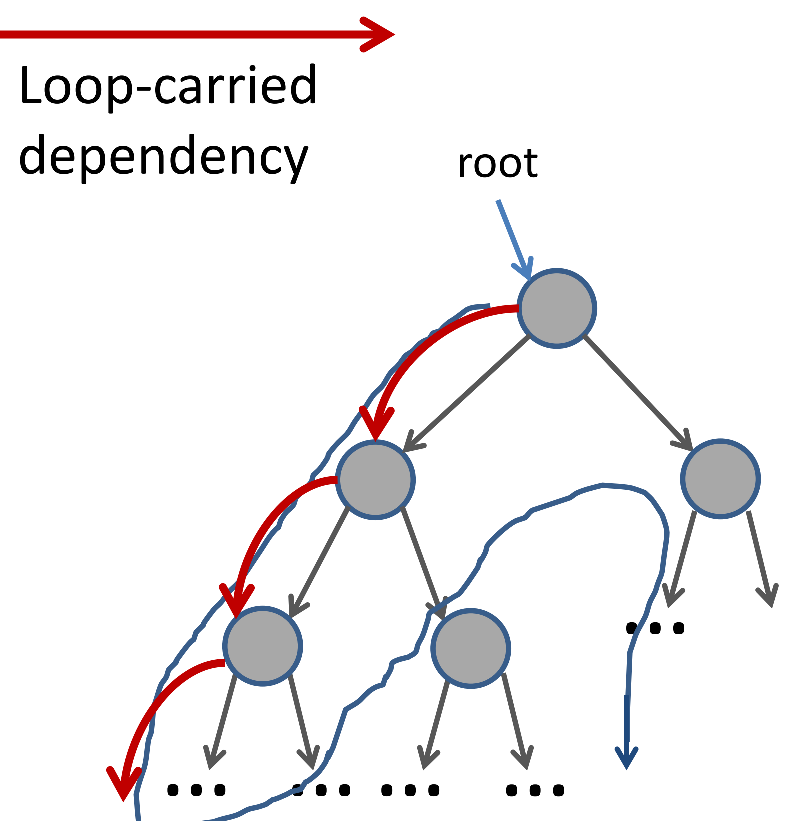
Pipelining




```
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
```

Change loop schedule to increase distance between dependent iterations

Pipelining




```
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
```

Imperial College London

Pipelining

Loop-carried dependency

root

...

... ...

```
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
```

Change loop schedule to increase distance between dependent iterations

Imperial College London

What else?

1. Generating application-specific multi-cache architectures
2. Loop transformations for pipelining
3. Worst-case/ average case bounds on heap usage

Imperial College London

Bounds on heap usage

Profiling the tree-based clustering app

N=16384, K=128, σ=0.2

x 22

Run-time [node-centre pairs x100k]

Bound on memory size [number of centre sets]

Imperial College London

Bounds on heap usage

Profiling the tree-based clustering app

N=16384, K=128, σ=0.2

Run-time [node-centre pairs x100k]

Bound on memory size [number of centre sets]

x 22

512 BRAMs

Cook et al., FMCAD 2009?

Imperial College London

Bounds on heap usage

Profiling the tree-based clustering app

N=16384, K=128, σ=0.2

Run-time [node-centre pairs x100k]

0, 5, 10, 15, 20

10^0, 10^1, 10^2, 10^3, 10^4, 10^5

Bound on memory size [number of centre sets]

x 22

8 BRAMs

How?

512 BRAMs

Cook et al., FMCAD 2009?

Imperial College London

Bounds on heap usage

Profiling the tree-based clustering app

Imperial College London

Conclusion

- Limited HLS support for heap-manipulating programs
- Static analysis of heap-manipulating programs
  - Leveraging recent advances in separation logic
  - Distribute heap across on-chip memory banks
  - Loop parallelization
- STeffiHLS tool implementation
  - Automated heap analyzer
  - Source-to-source transformations (synthesizability and parallelization)
  - Successful parallelization using standard HLS tool
- Future work
  - Generating application-specific multi-cache architectures
  - Loop transformations for pipelining
  - Compute worst-case/average-case bounds on heap usage

Thank you for listening.

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