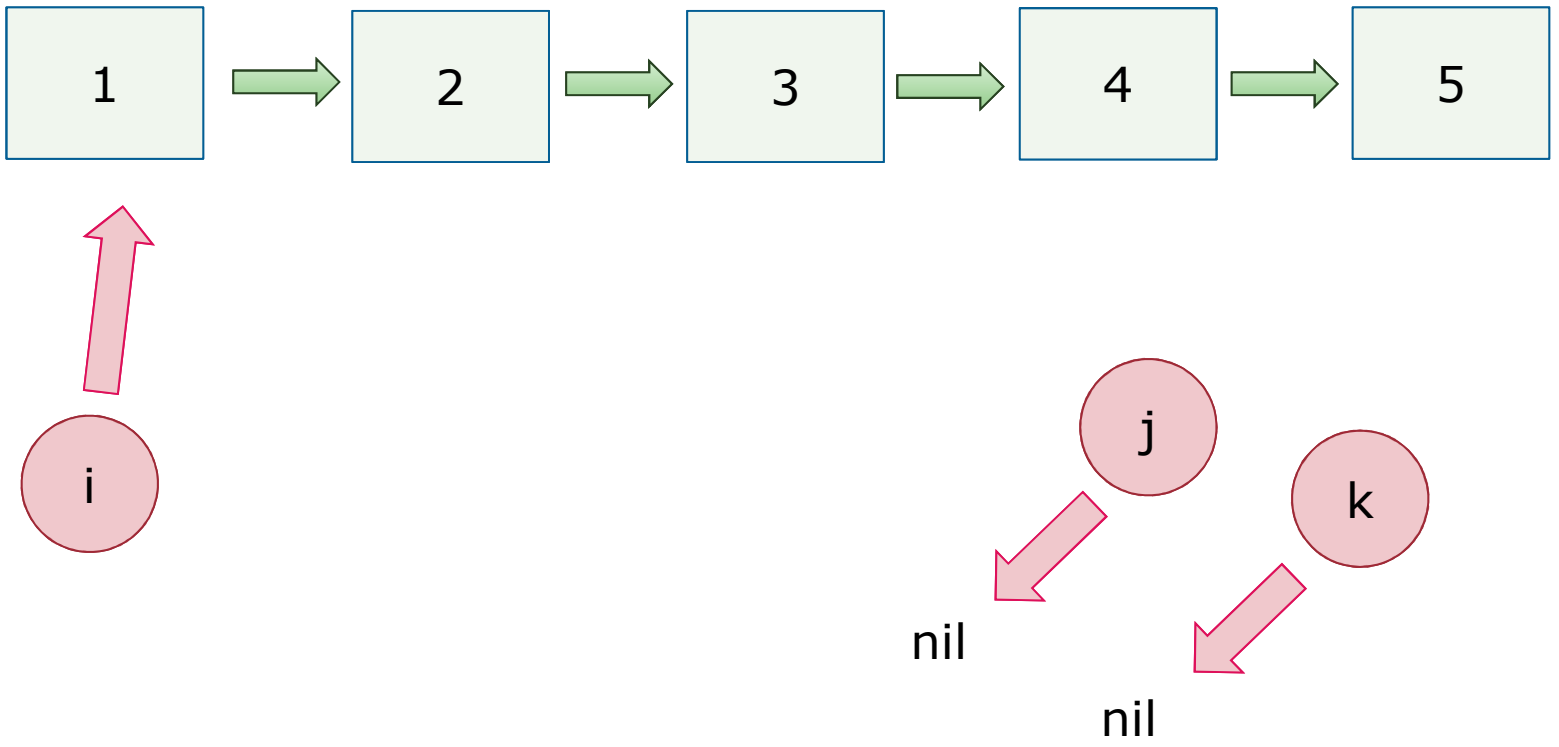
Separation as Abstraction

Nisansala Yatapanage
with Cliff Jones and Andrius Velykis
Newcastle University

- Issues of separation are well handled by Concurrent Separation Logic.
- (at least) Some examples can be clearly developed using layers of abstraction.
- Two examples: Reynolds' in-place list reversal algorithm and concurrent DOM trees.

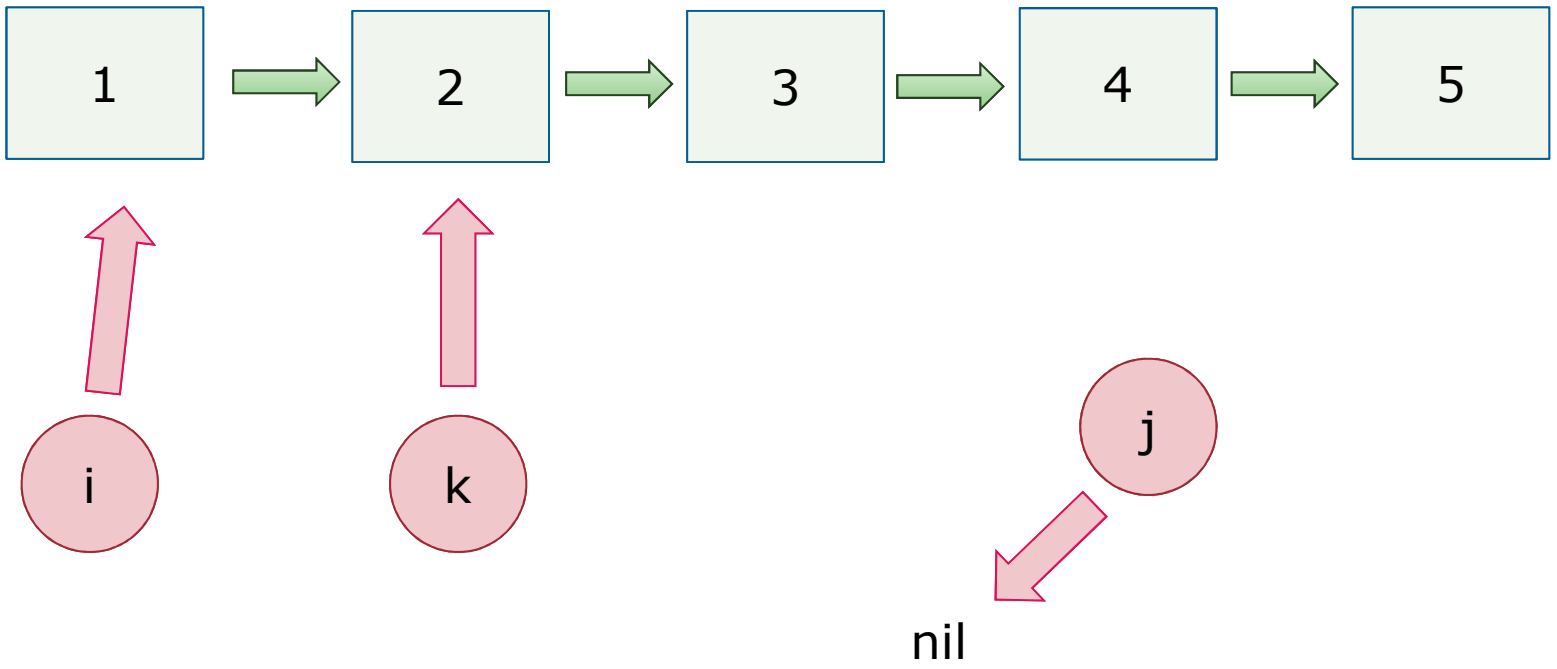
In-place List Reversal

Originally presented by John Reynolds



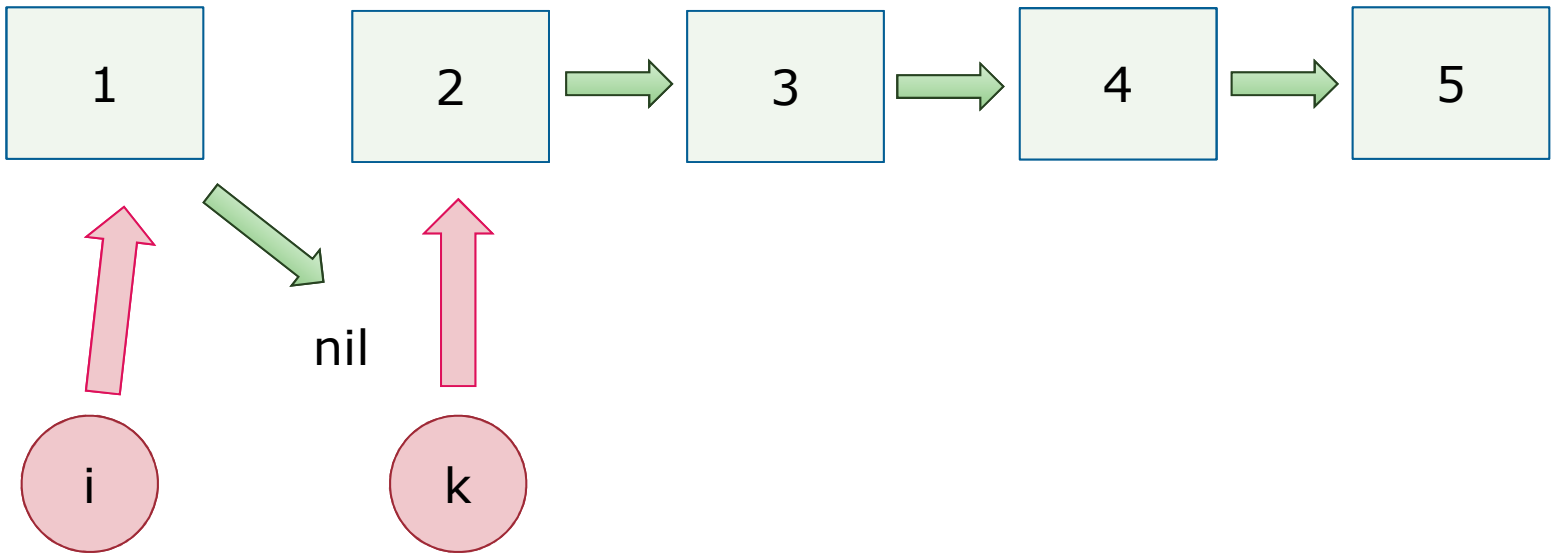
In-place List Reversal

Originally presented by John Reynolds



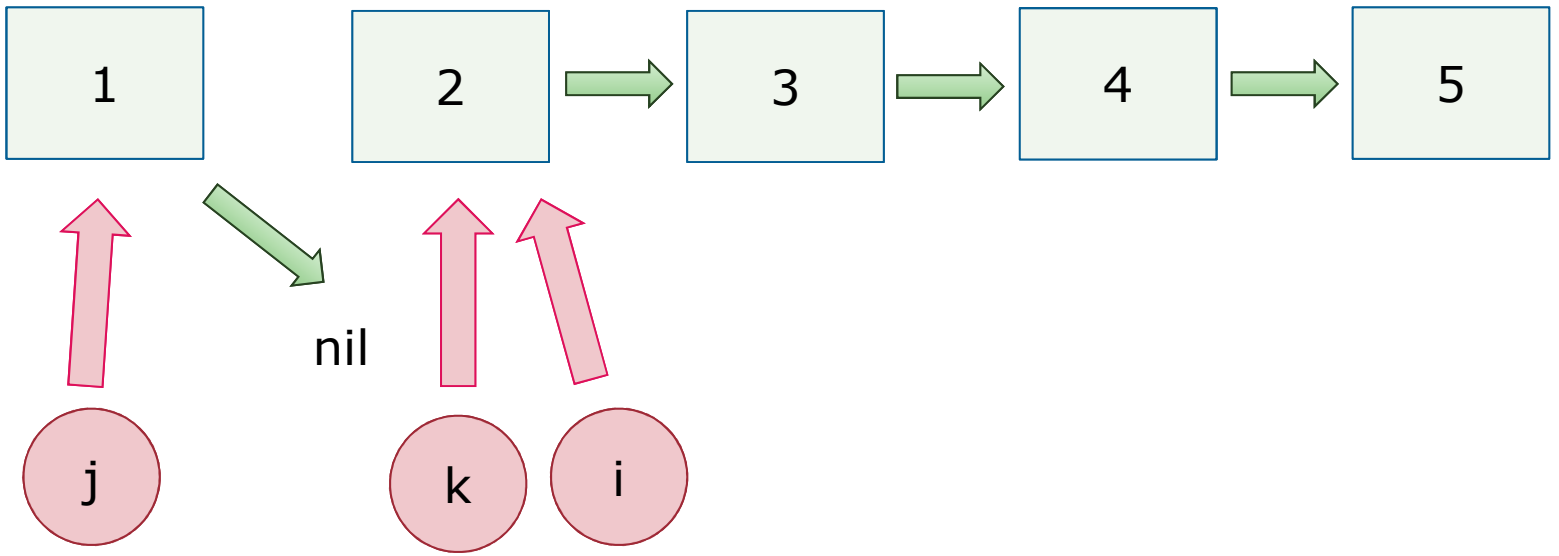
In-place List Reversal

Originally presented by John Reynolds



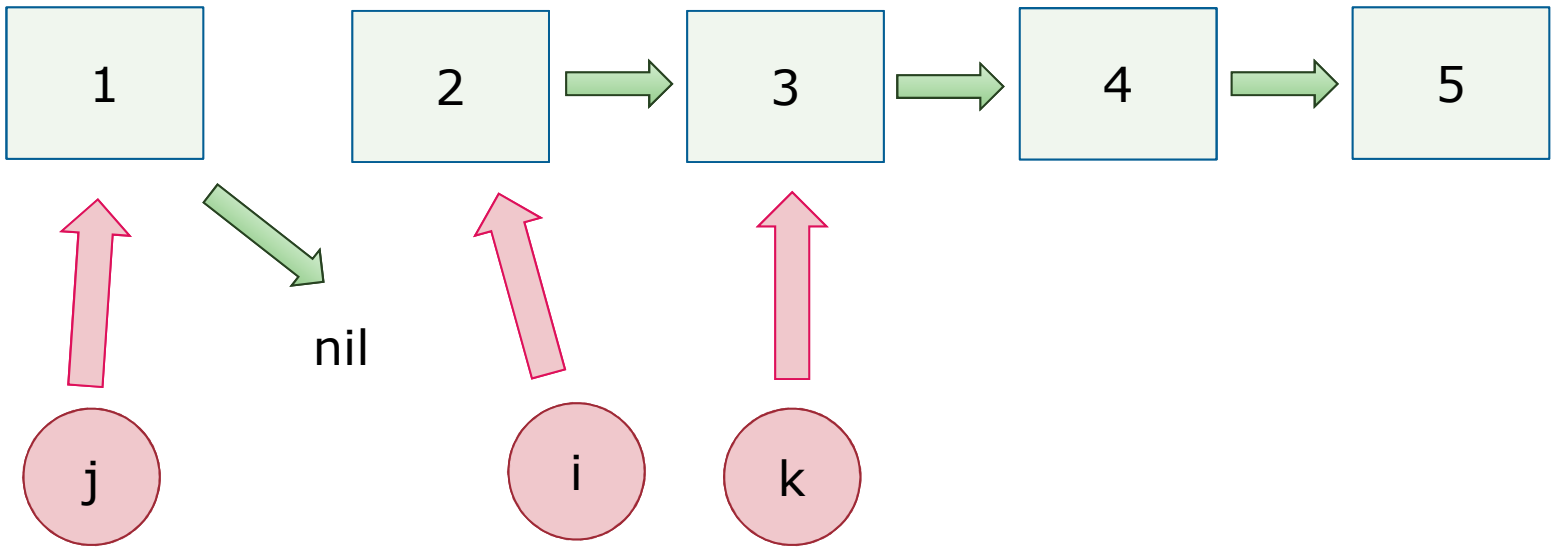
In-place List Reversal

Originally presented by John Reynolds



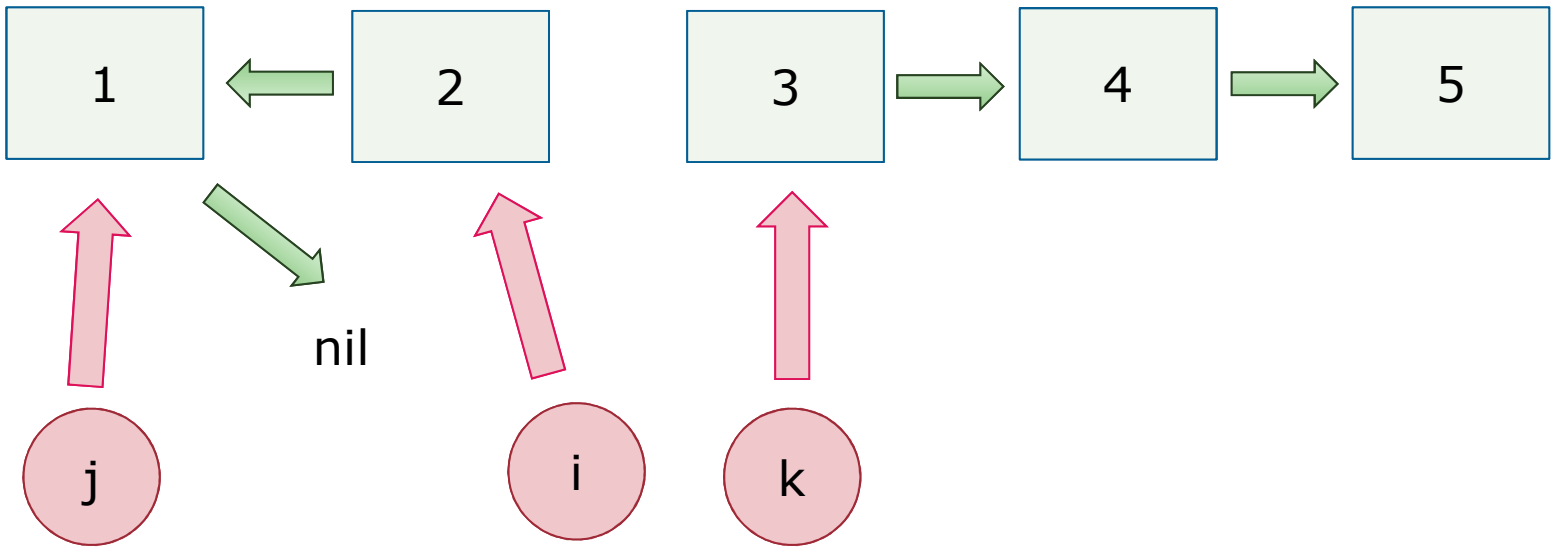
In-place List Reversal

Originally presented by John Reynolds



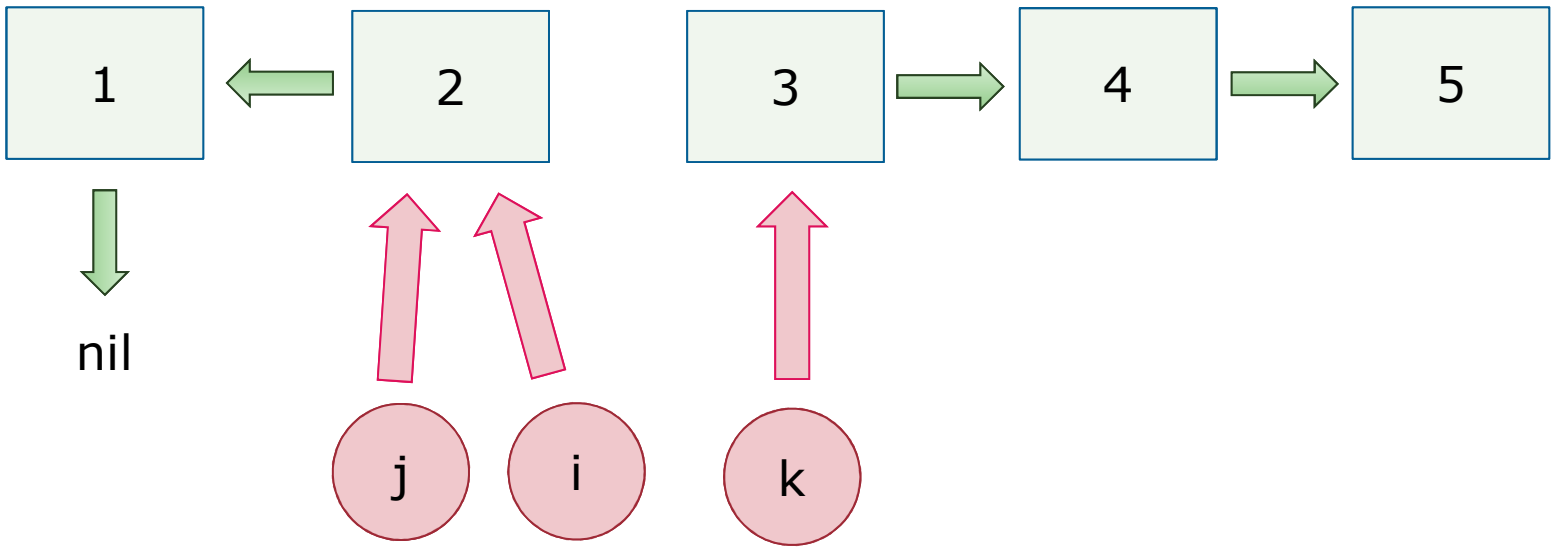
In-place List Reversal

Originally presented by John Reynolds



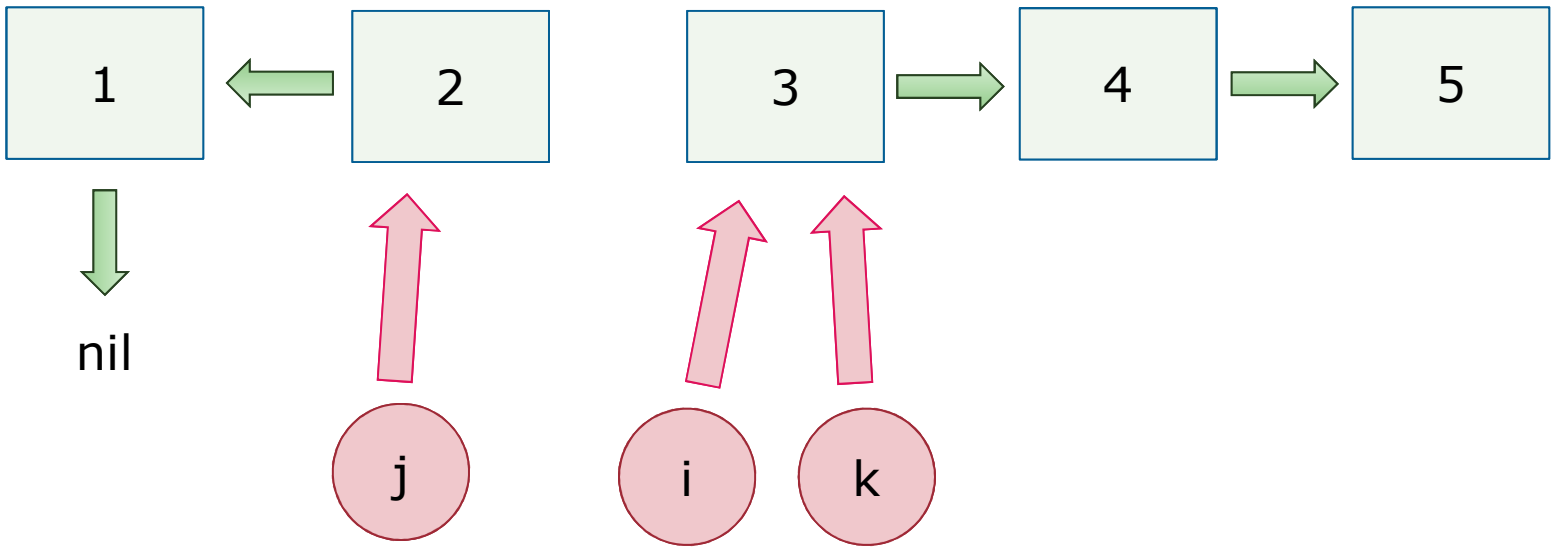
In-place List Reversal

Originally presented by John Reynolds



In-place List Reversal

Originally presented by John Reynolds



Abstract List Reversal

while s is not empty **do**
    r' = head of s joined to r
    s' = tail of s
end while

Postcondition: r’ = rev(s)

Separation

- In the abstract specification, r and s are assumed to be distinct.
- In normal data reification, the separation is preserved.

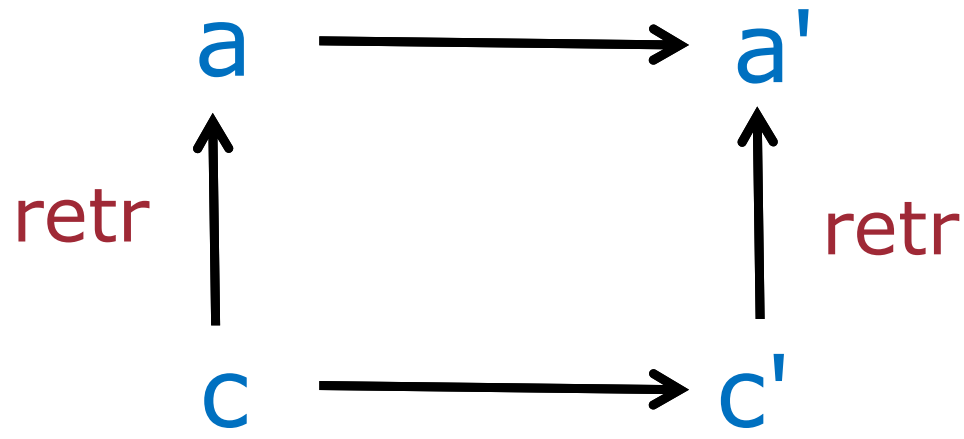
Separation

- In the abstract specification, r and s are assumed to be distinct.
- In normal data reification, the separation is preserved.
- For Reynold's algorithm, we are reifying r and s onto parts of the same vector.
- This leads to a new form of reification: preservation of separation.

Implementation

The separation has to be stated as a predicate in the invariant.

The implementation can be shown to satisfy the abstract specification.



Proof – retr

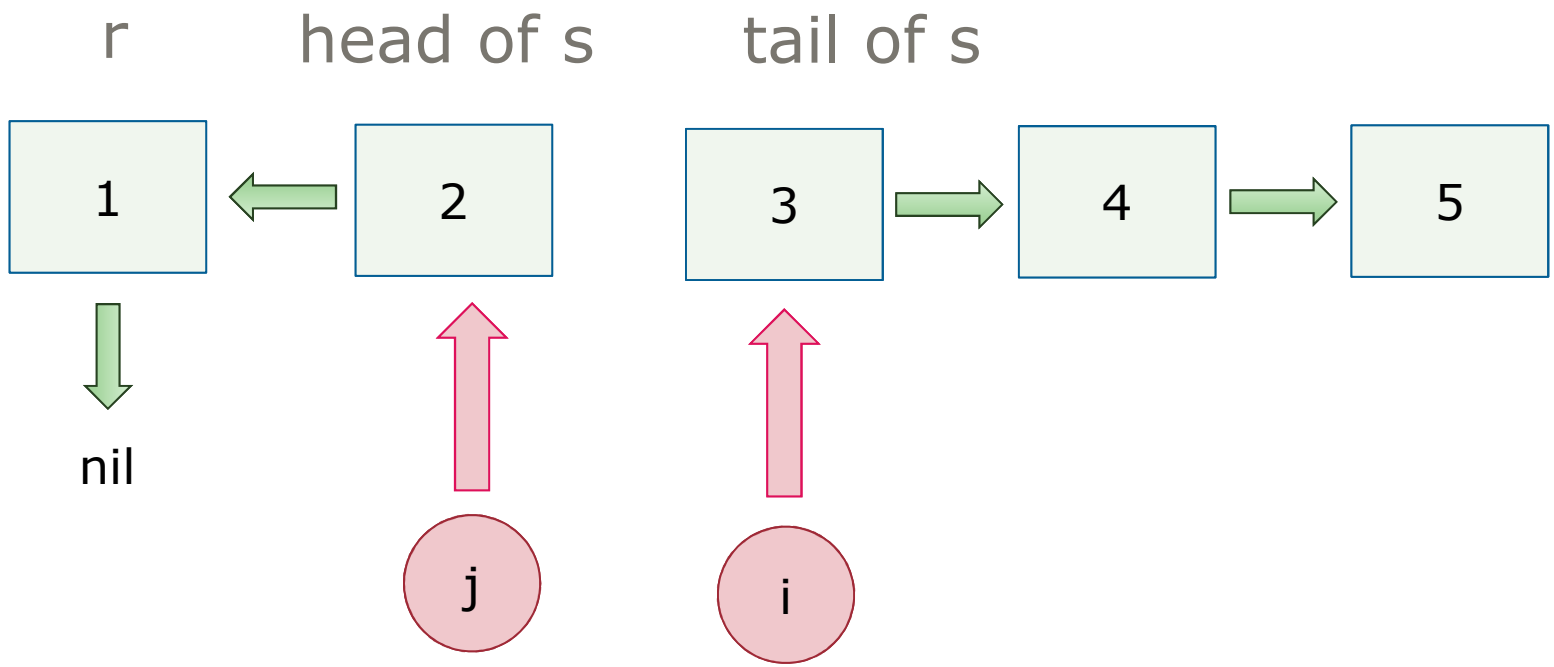
$$retr : \Sigma_r \rightarrow \Sigma_a$$

$$retr(mk\text{-}\Sigma_r(m, i, j)) \triangleq$$
$$\quad mk\text{-}\Sigma_a(gather(i, m), gather(j, m))$$

$$gather(n, m) \triangleq$$
$$\quad \textbf{if } n = nil$$
$$\quad \textbf{then } [\,]$$
$$\quad \textbf{else let } (v, p) = m(n) \textbf{ in}$$
$$\quad\quad [v] \frown gather(p, m)$$

Proof

At each step, recall: r' = head of s joined to r;
s' = tail of s



Concurrent DOM Trees

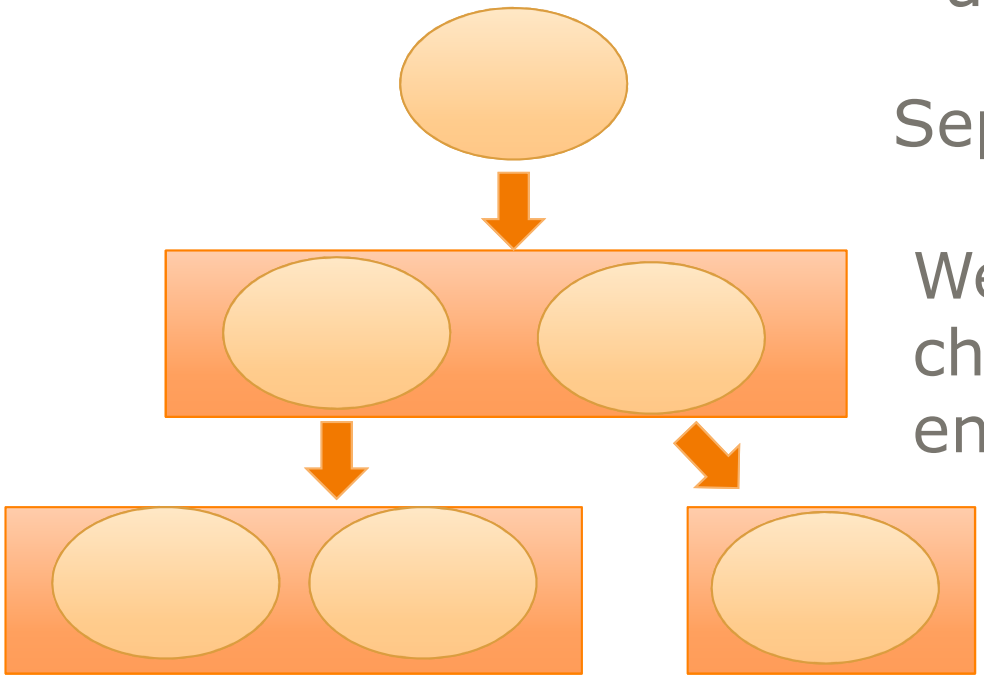
- Philippa Gardner & colleagues published proofs using separation logic of sequential algorithms for updating DOM trees.

Concurrent DOM Trees

- Philippa Gardner & colleagues published proofs using separation logic of sequential algorithms for updating DOM trees.
- Our first attempt at abstraction used recursive structures.
- This wasn’t suitable as we needed NodeID’s to match with DOM.

Concurrent DOM Trees

Abstract trees



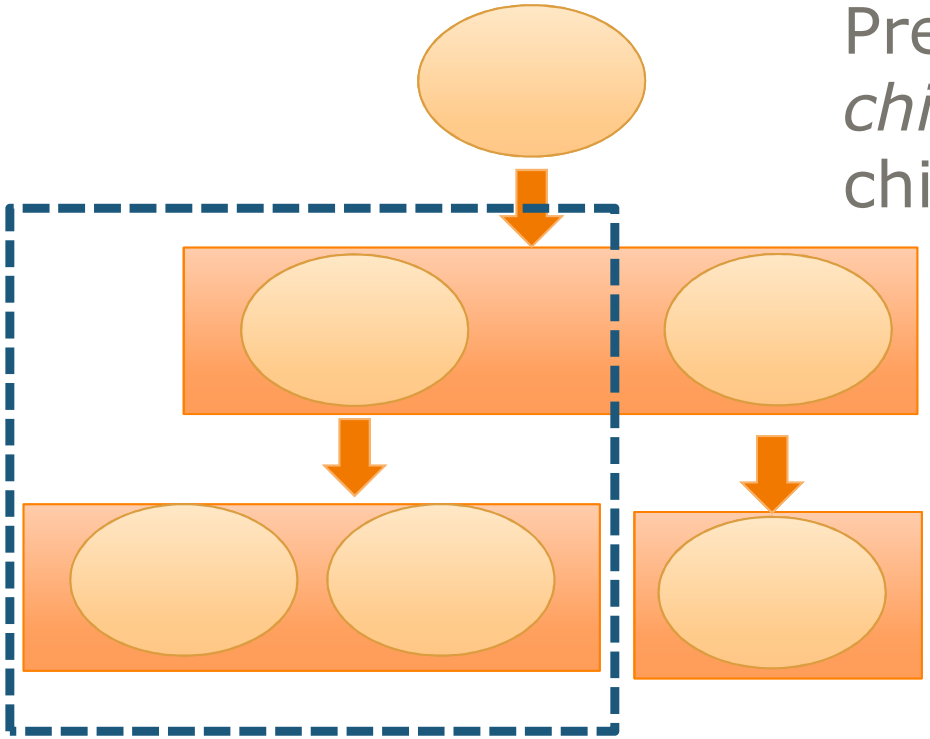
Each node has data and a list of children.

Separation is implied by:

Well-foundedness of the child to parent relation, ensuring no loops.

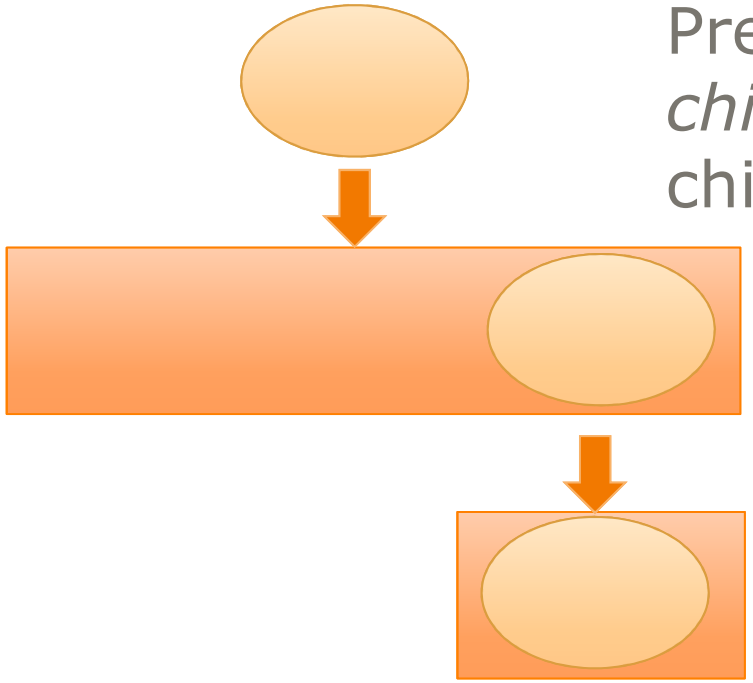
No child has more than one parent.

Abstract Sequential Remove



Pre: *parent* exists, *child* is one of *parent*'s children

Abstract Sequential Remove



Pre: *parent* exists, *child* is one of *parent*'s children

Post: The final state is the same except for that sub-tree removed from its parent.

From sequential to concurrent

- The sequential postcondition can state equalities about the whole system.

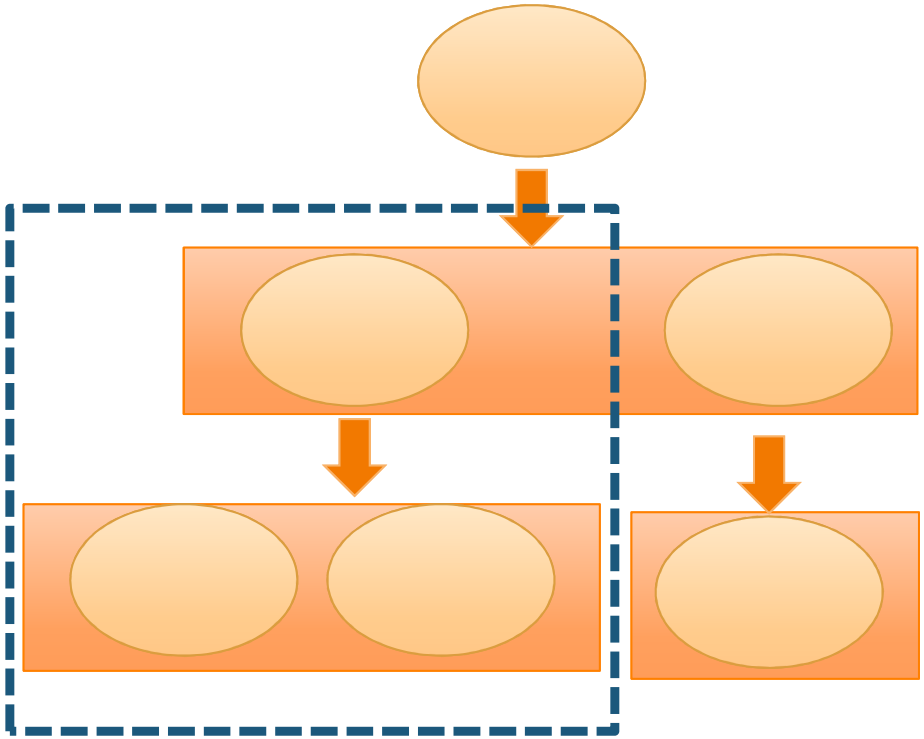
From sequential to concurrent

- The sequential postcondition can state equalities about the whole system.
- With concurrency, other processes may be operating simultaneously.

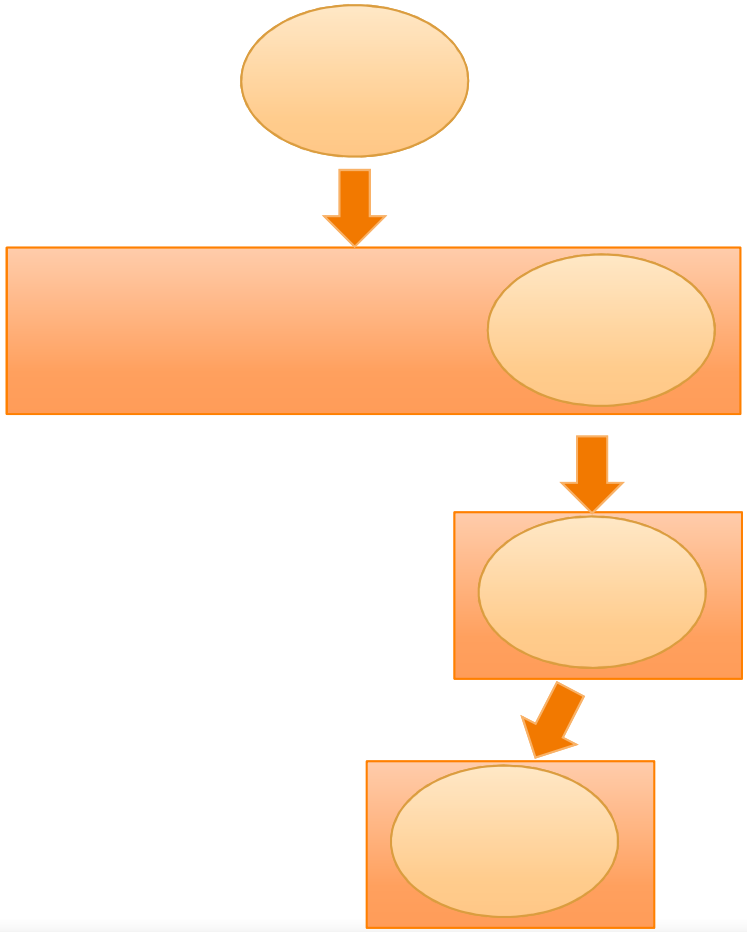
From sequential to concurrent

- The sequential postcondition can state equalities about the whole system.
- With concurrency, other processes may be operating simultaneously.
- The postcondition is weakened to form the concurrent postcondition and the guar.

Abstract Concurrent Remove

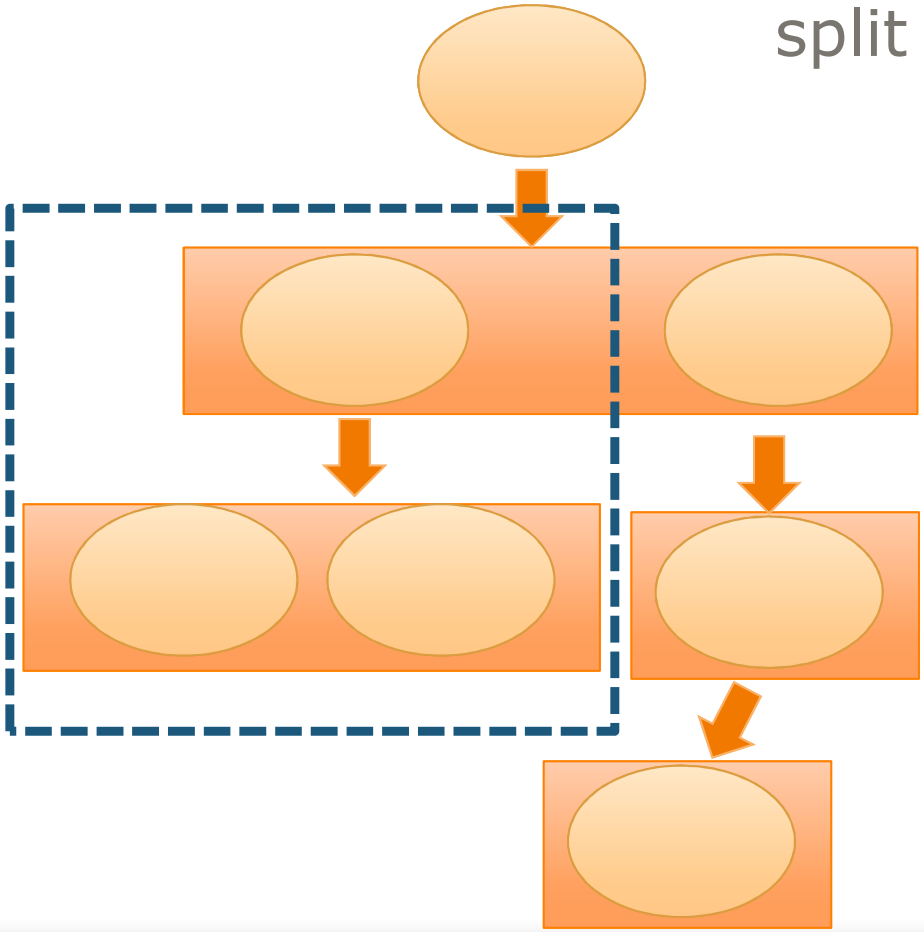


Abstract Concurrent Remove

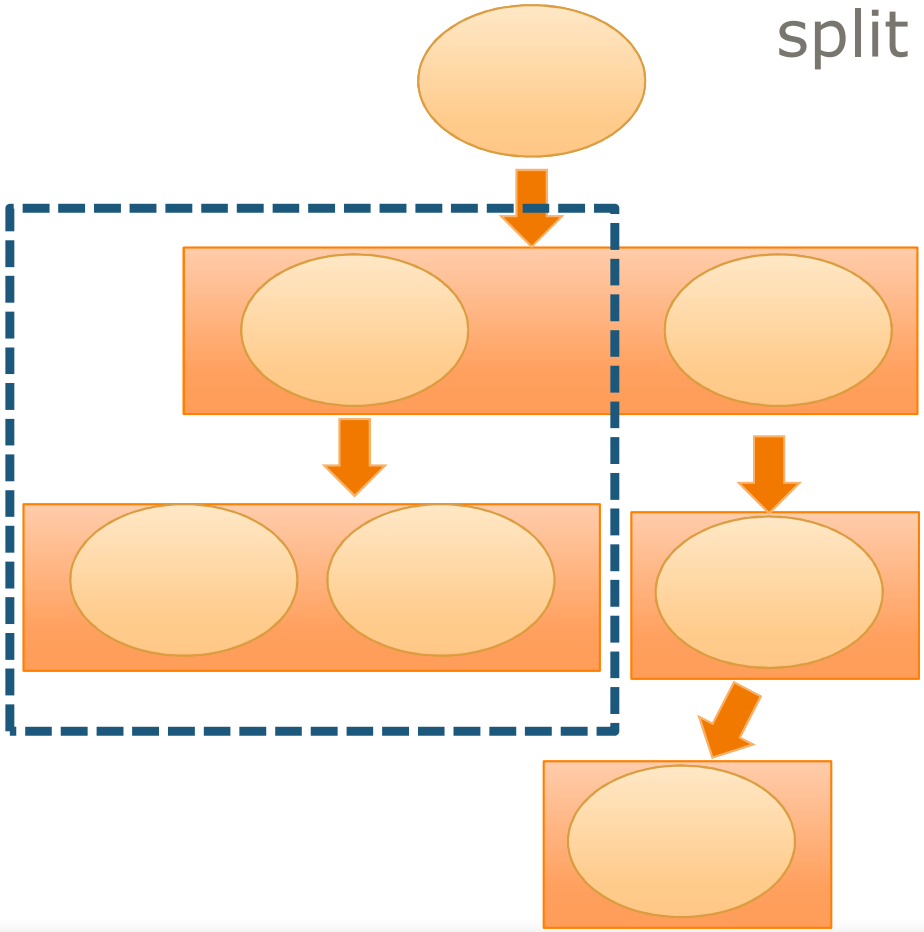


Abstract Concurrent Remove

The abstract post is now split into the post and guar:



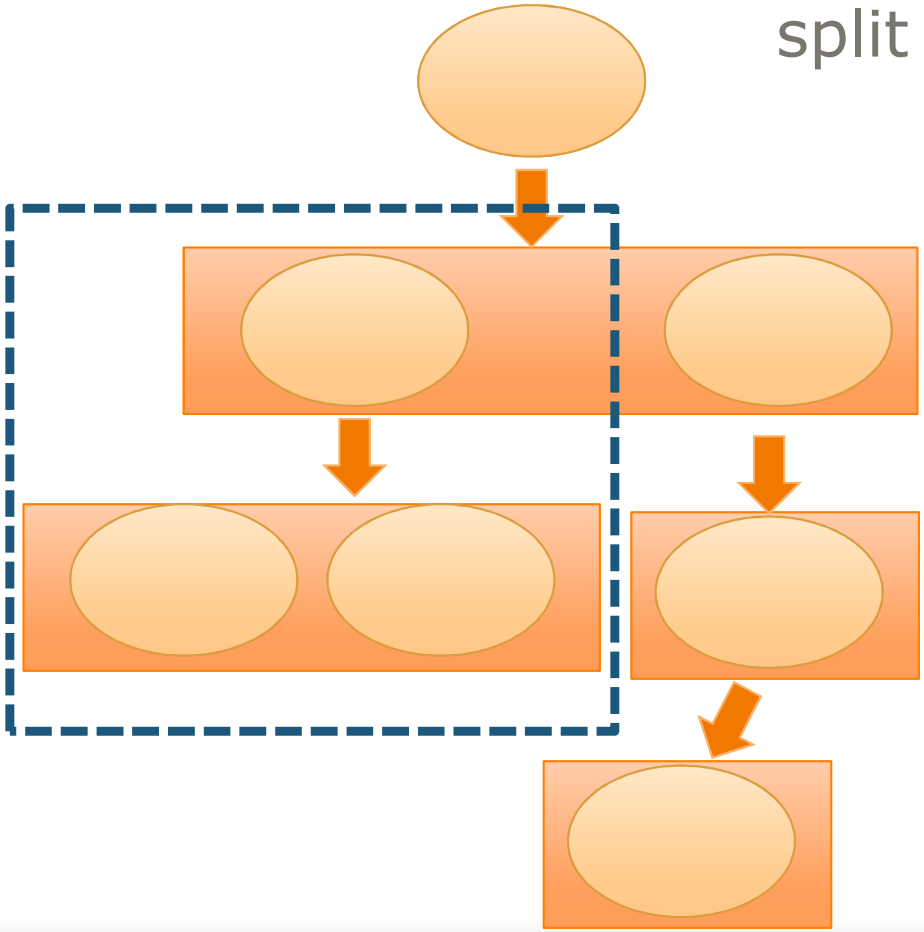
Abstract Concurrent Remove



The abstract post is now split into the post and guar:

post: *child* is removed from *parent*'s children list. (Nothing about the rest of the tree).

Abstract Concurrent Remove

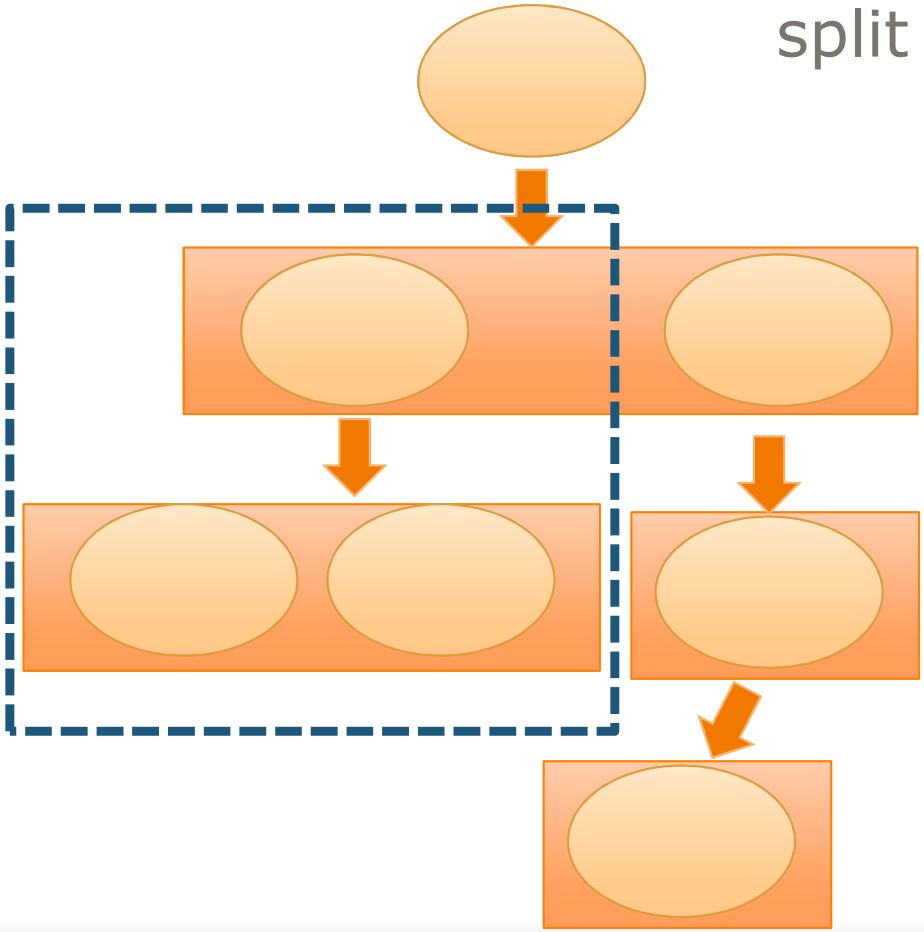


The abstract post is now split into the post and guar:

post: *child* is removed from *parent*'s children list. (Nothing about the rest of the tree).

guar: this process will only change *parent*'s children

Abstract Concurrent Remove



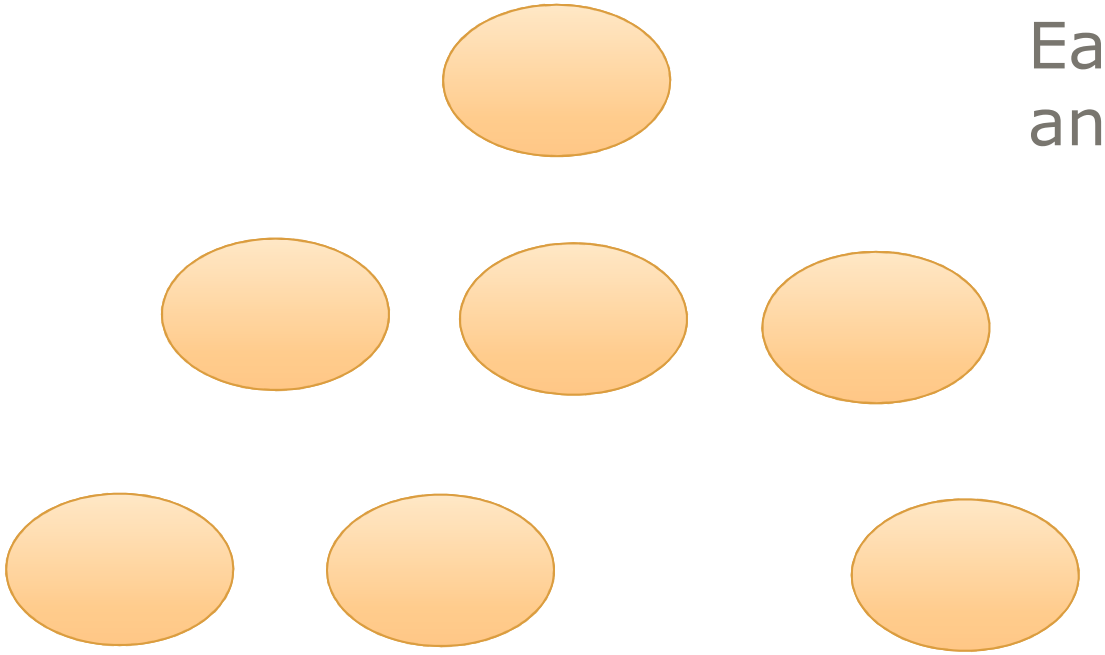
The abstract post is now split into the post and guar:

post: *child* is removed from *parent*'s children list. (Nothing about the rest of the tree).

guar: this process will only change *parent*'s children

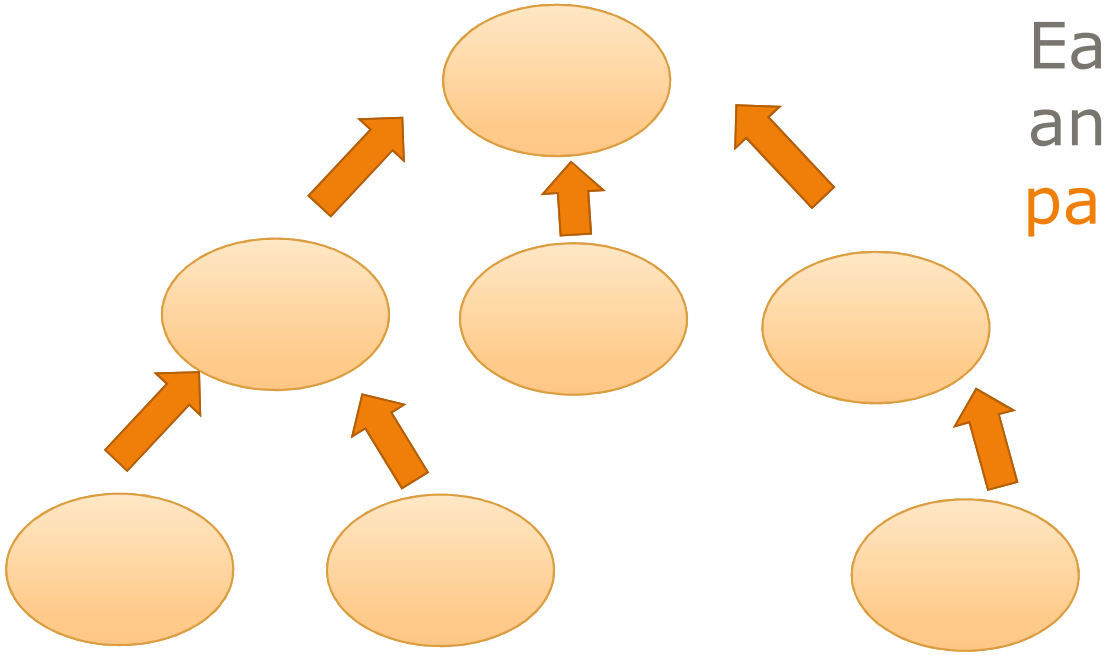
rely: *parent*'s children won't change

Data Reification



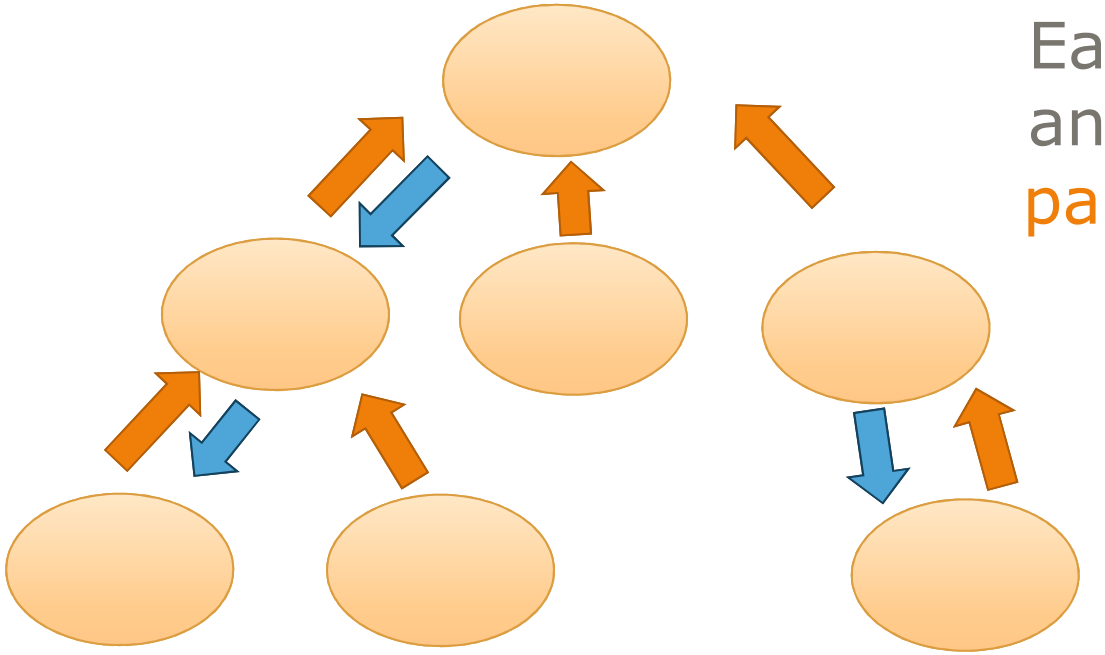
Each node has data and pointers to its

Data Reification



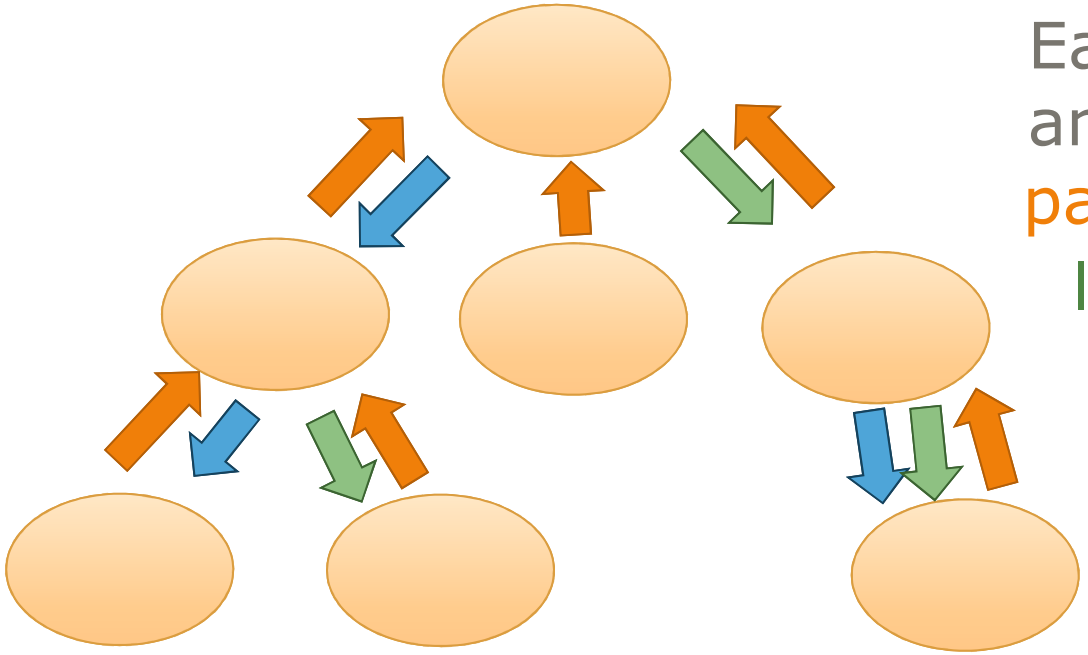
Each node has data and pointers to its parent,

Data Reification



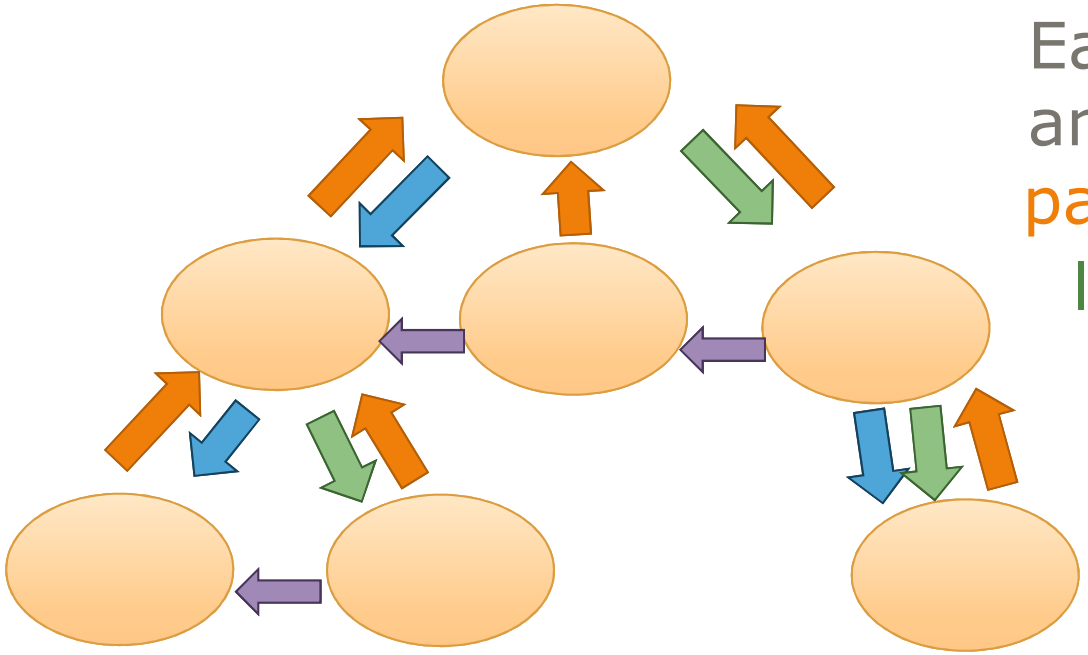
Each node has data and pointers to its parent, first child,

Data Reification



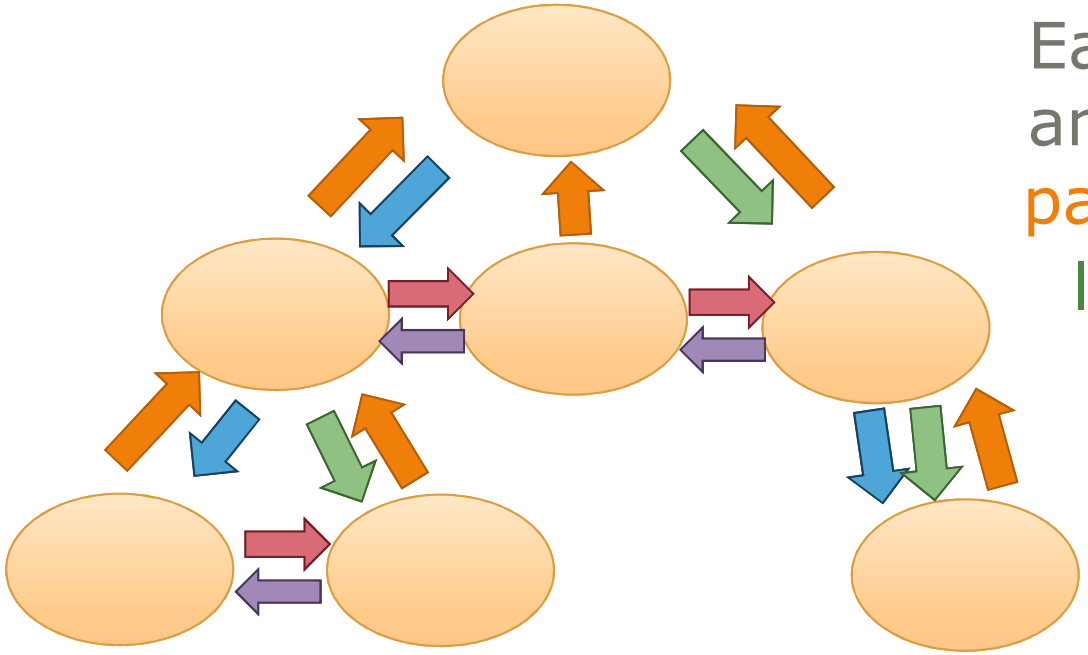
Each node has data and pointers to its parent, first child, last child,

Data Reification



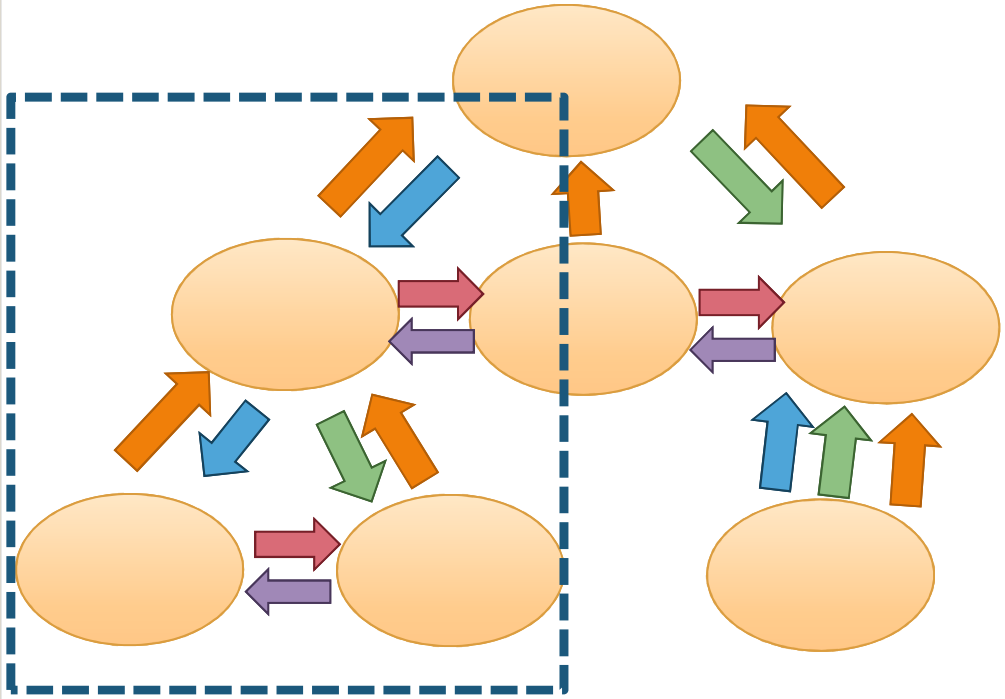
Each node has data and pointers to its parent, first child, last child, previous sibling

Data Reification



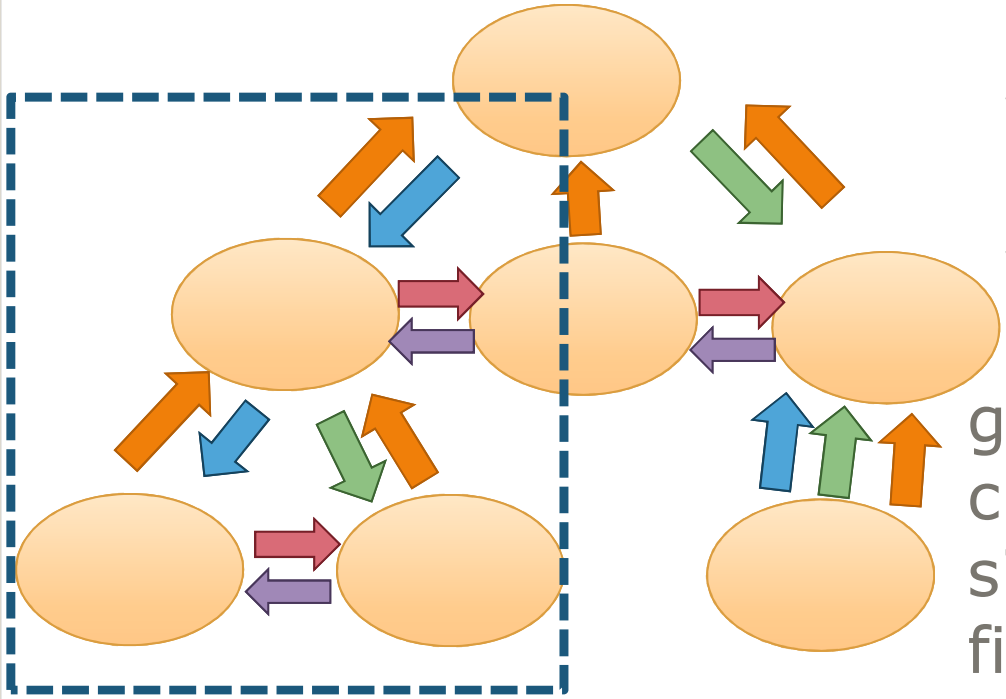
Each node has data and pointers to its parent, first child, last child, previous sibling and next sibling.

Reified Concurrent Remove



rely: *parent*'s children won't change, *child*'s parent & siblings won't change.

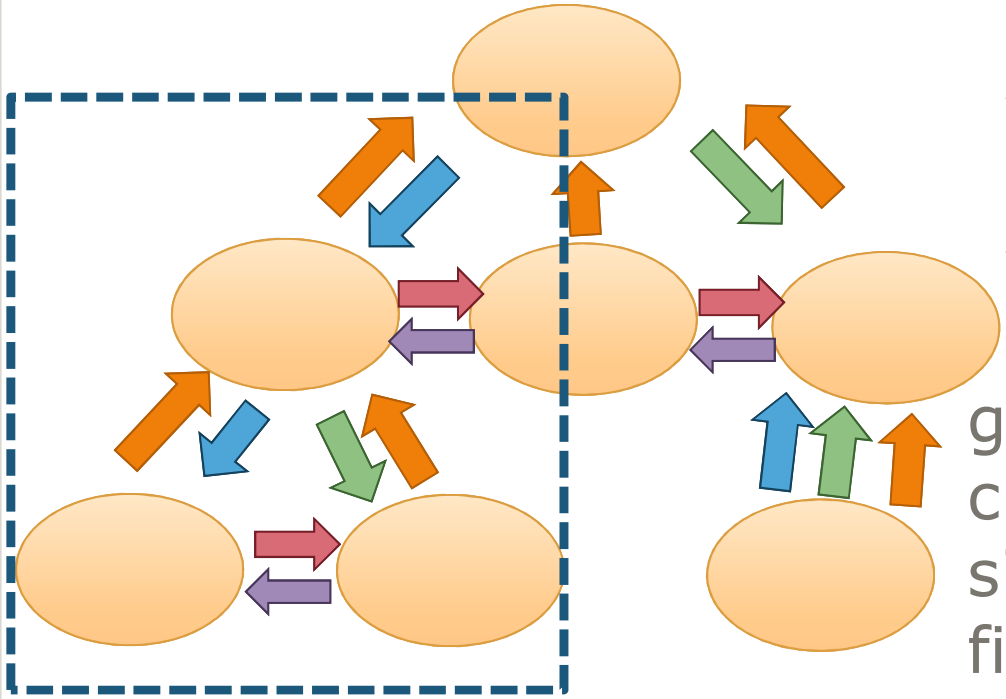
Reified Concurrent Remove



rely: *parent*'s children won't change, *child*'s parent & siblings won't change.

guar: This may only change *child*'s parent & sibling pointers, *parent*'s first/last child pointers.

Reified Concurrent Remove

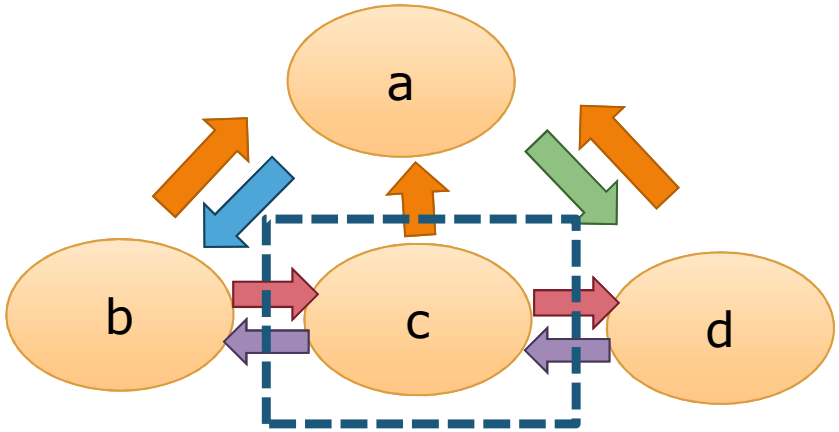


rely: *parent*'s children won't change, *child*'s parent & siblings won't change.

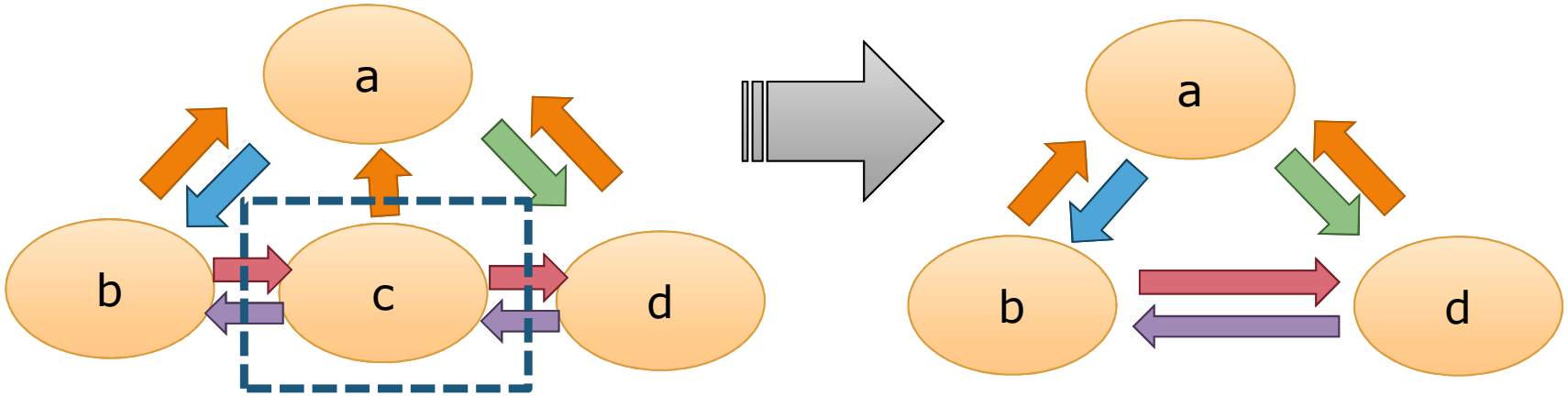
guar: This may only change *child*'s parent & sibling pointers, *parent*'s first/last child pointers.

post: *child*'s parent, prev & next siblings are nil and the sibling pointers are updated.

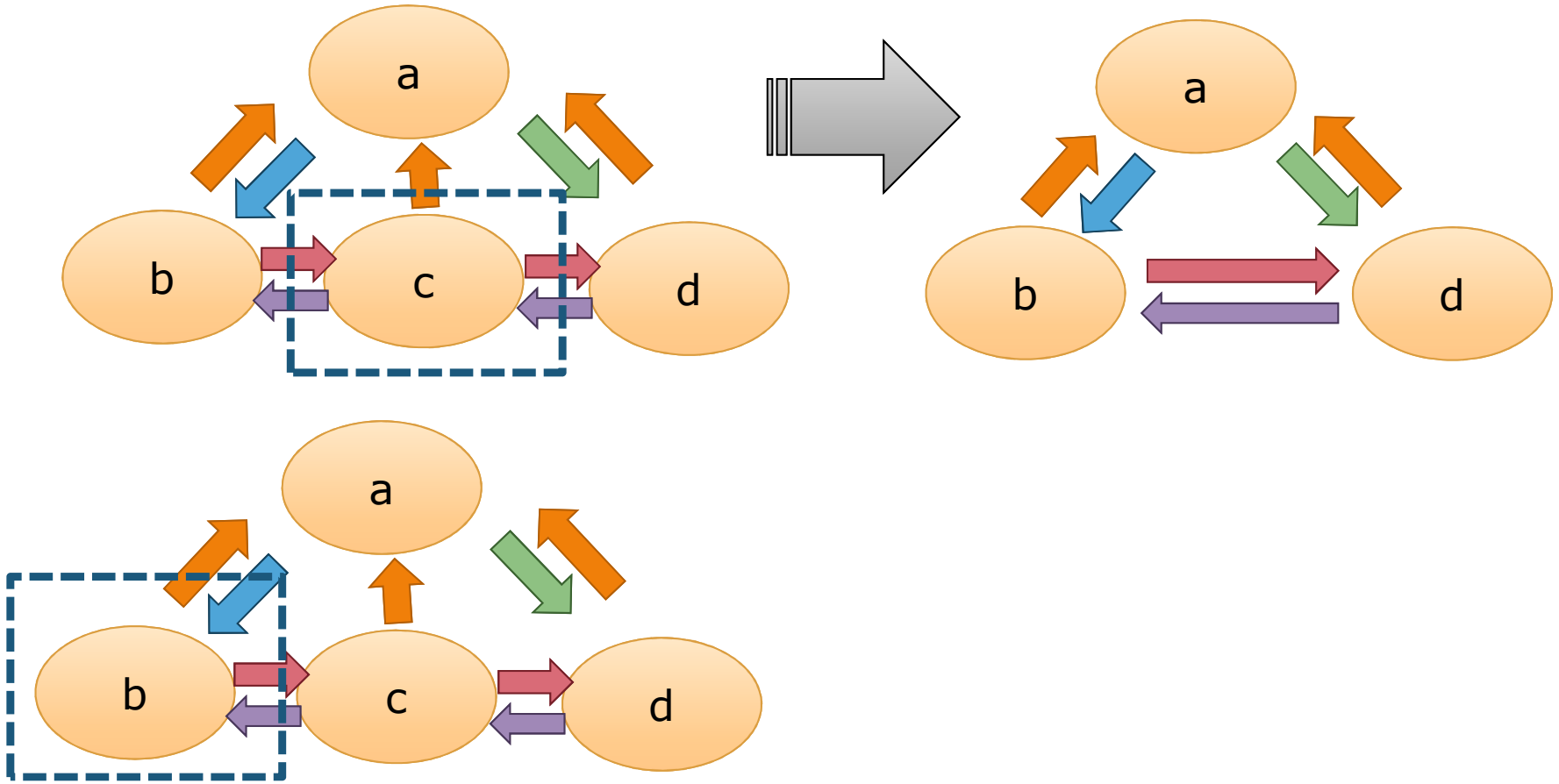
Updating siblings



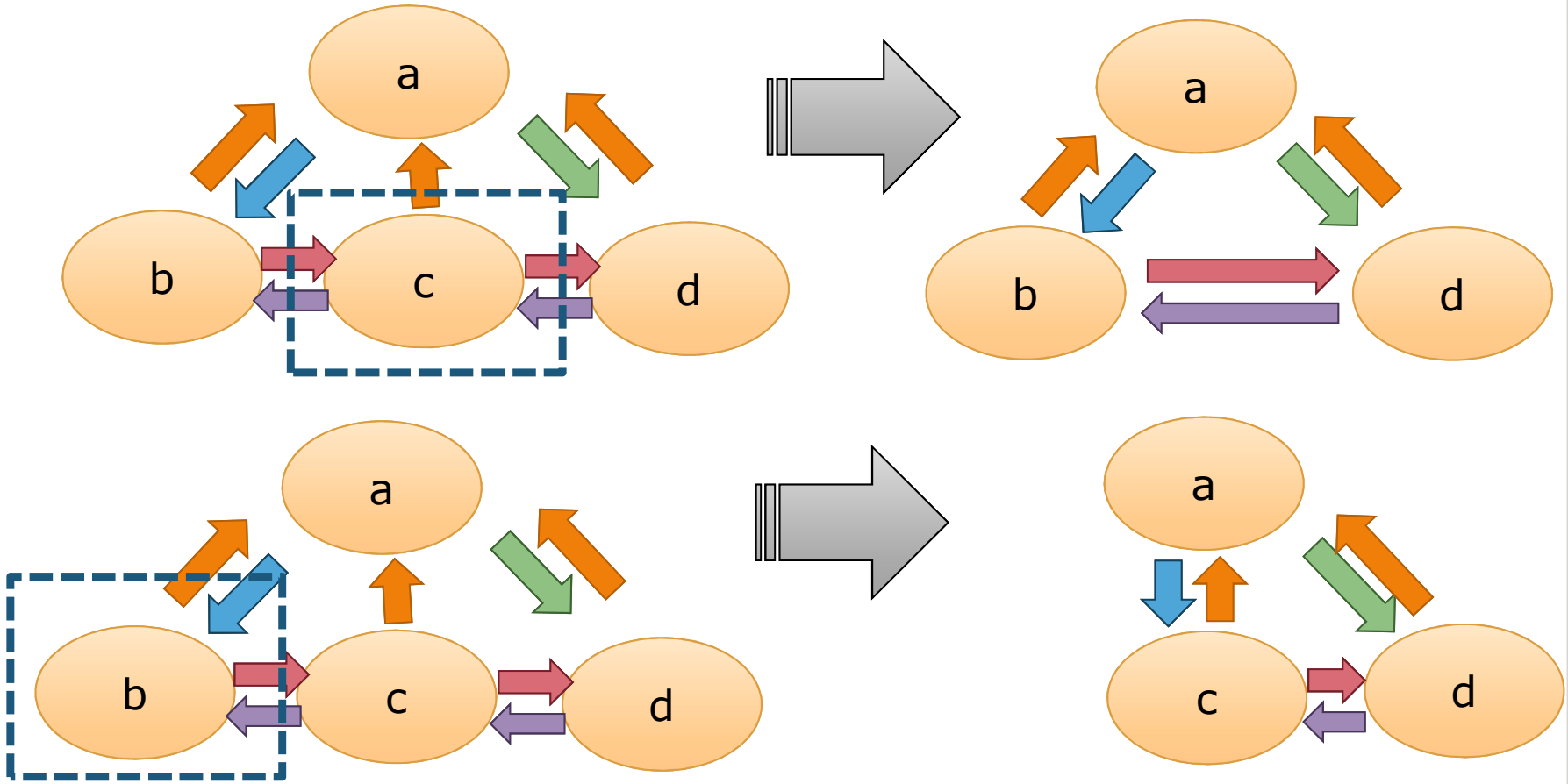
Updating siblings



Updating siblings



Updating siblings



Locking

- Rely/guar has been used on lock-free algorithms, e.g. 4-slot

Locking

- Rely/guar has been used on lock-free algorithms, e.g. 4-slot
- For this algorithm, the rely/guar conditions can be satisfied by locking the parent.

Locking

- Rely/guar has been used on lock-free algorithms, e.g. 4-slot
- For this algorithm, the rely/guar conditions can be satisfied by locking the parent.
- Finer-grained locking can be accomplished by using special sentinel nodes at the corners.

Conclusions

- Handling separation by reasoning with layers of abstraction seems promising.

Conclusions

- Handling separation by reasoning with layers of abstraction seems promising.
- For concurrency, the sequential postcondition can be weakened to form the concurrent postcondition and the guar condition.

Conclusions

- Handling separation by reasoning with layers of abstraction seems promising.
- For concurrency, the sequential postcondition can be weakened to form the concurrent postcondition and the guar condition.
- The process was demonstrated from abstract sequential trees to abstract concurrent trees to reified concurrent trees.