A Specification-Based Approach to Reasoning about Pointers

Gregory Kulczycki
Virginia Tech

Murali Sitaraman
Clemson University

Bruce W. Weide
The Ohio State University

Atanas Rountev
The Ohio State University

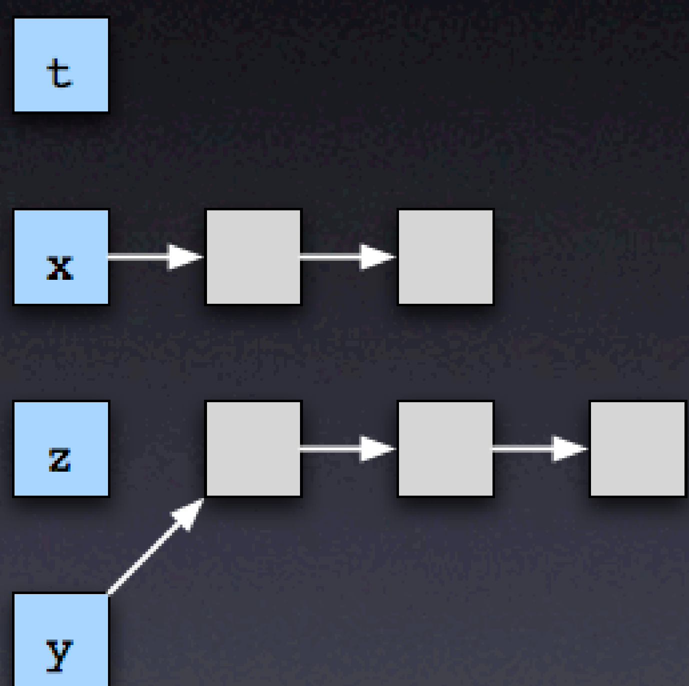
Motivation

Traditional static analysis techniques for pointers are limited in what they can prove about pointer programs.

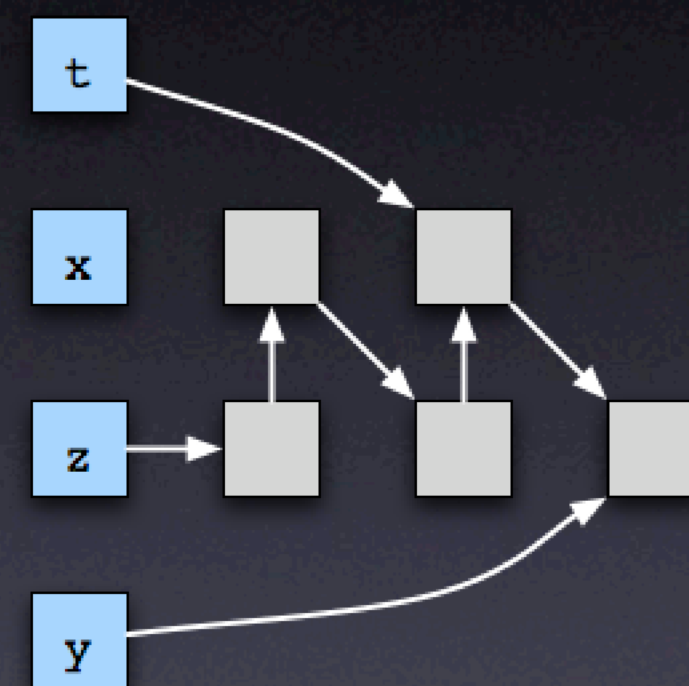
Example: Shape Analysis

```
typedef struct list {
  struct list *n;
  int data;
} List;

List *splice(List *x, List *y) {
  List *t = NULL;
  List *z = y;
  while (x != NULL) {
    t = x;
    x = t->n;
    t->n = y->n;
    y->n = t;
    y = y->n->n;
  }
  return z;
}
```

Example: Shape Analysis



Example state at the beginning of splice



Example state at the end of splice

Example: Shape Analysis

“Statically verify that, if the input lists x and y are disjoint and acyclic, then the list returned by splice is acyclic.”

Region-based shape analysis with tracked locations
Hackett and Rugina, POPL 2005

Example: Shape Analysis

"Statically verify that, if the input lists x and y are disjoint and acyclic, then the list returned by splice is acyclic."

Region-based shape analysis with tracked locations
Hackett and Rugina, POPL 2005

Example: Shape Analysis

Shape analysis helps with some properties:

- Is a memory location referenced by more than one other location?
- Is a location accessed through a dangling references?
- Are memory leaks present?

Example: Shape Analysis

Shape analysis helps with some properties:

- Is a memory location referenced by more than one other location?
- Is a location accessed through a dangling references?
- Are memory leaks present?

But it does not help with other properties:

- Does the splice operation do what it is supposed to do (i.e., does the operation interleave the elements of the incoming lists)?

Specification-Based Approach

- Provide a generic pointer component
- Include its full formal specification
- Give it a special implementation
- Special syntax is optional

Mathematical Model

Concept Location_Linking_Template (**type** Info);

Defines Location: **Set**;
Defines Void: Location;
Var Target: Location → Location;
Var Contents: Location → Info;
Var Is_Taken: Location → **B**;

Initialization ensures ∀q: Location, ¬Is Taken(q);

Type Position **is modeled by** Location;
  exemplar p;
  Initialization ensures p = Void;

...

Mathematical Model

Concept Location_Linking_Template (**type** Info);

Defines Location: **Set**;
Defines Void: Location;
Var Target: Location $\rightarrow$ Location;
Var Contents: Location $\rightarrow$ Info;
Var Is_Taken: Location $\rightarrow$ **B**;

Definitions

Initialization ensures $\forall$q: Location, $\neg$Is Taken(q);

Type Position **is modeled by** Location;
    exemplar p;
    Initialization ensures p = Void;

...

Exported Type

Mathematical Model

Concept Location_Linking_Template (**type** Info);

Defines Location: **Set**;
Defines Void: Location;
Var Target: Location → Location;
Var Contents: Location → Info;
Var Is_Taken: Location → **B**;

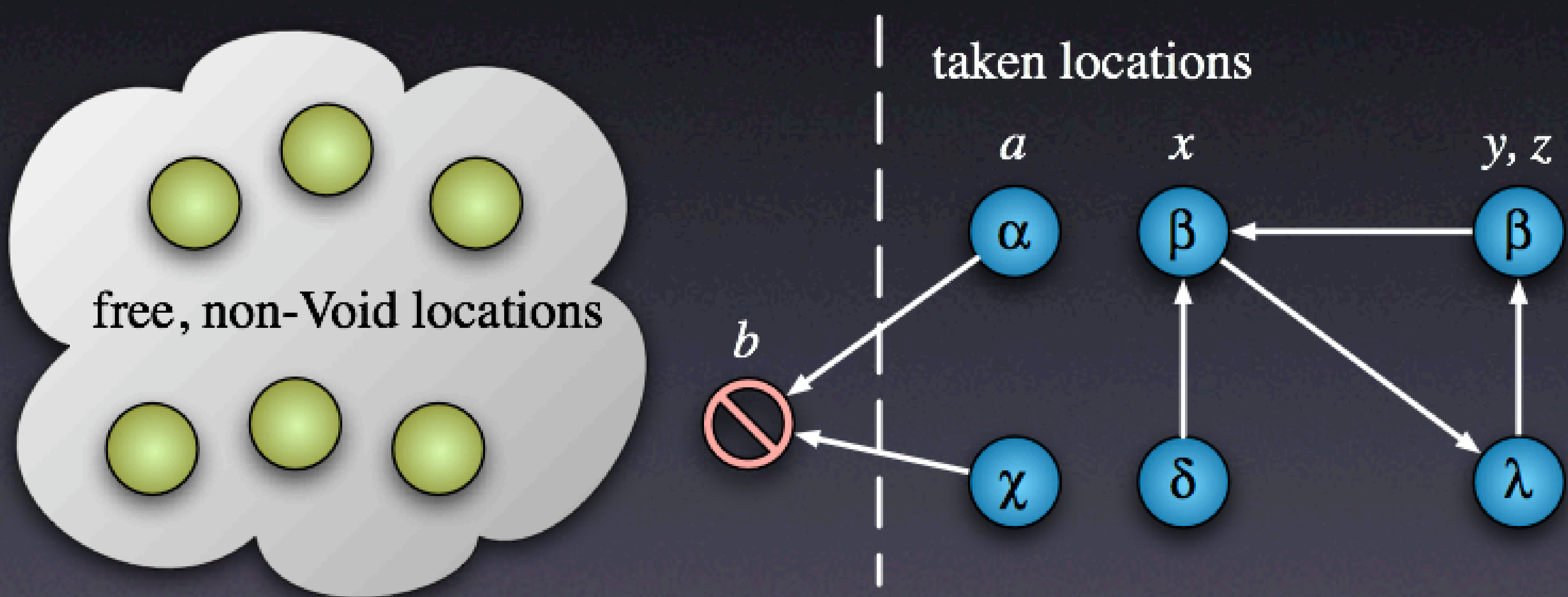
Conceptual Variables

Initialization ensures ∀q: Location, ¬Is Taken(q);

Type Position **is modeled by** Location;
  exemplar p;
  Initialization ensures p = Void;

...

A System of Linked Locations



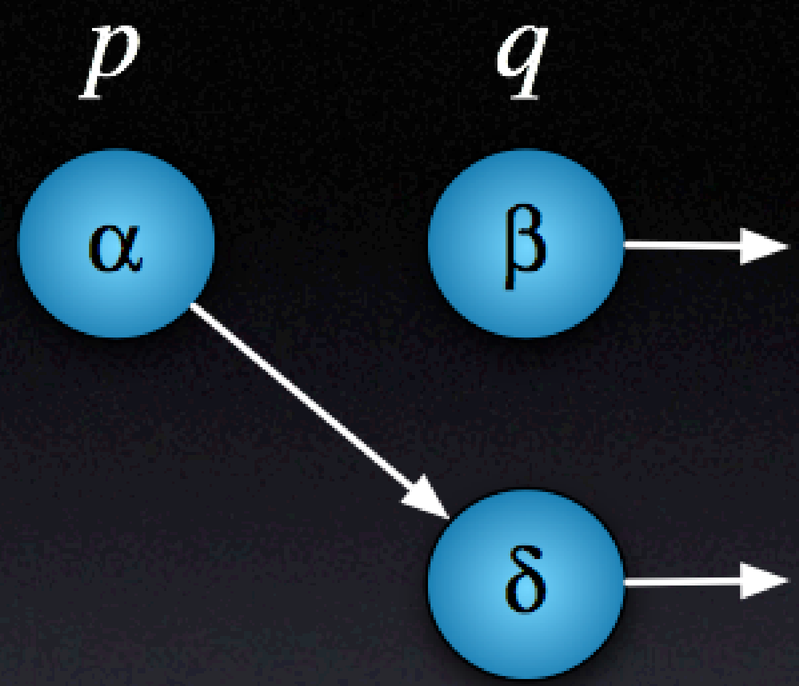
Operation Signatures

```
Concept Location_Linking_Template (type Info);
    Type Position;
    Operation Take_New_Location (updates p: Position);
    Operation Abandon_Location (clears: p: Position);
    Operation Relocate (updates p: Position;
                        preserves q: Position);
    Operation Follow_Link (updates p: Position);
    Operation Redirect_Link (preserves p: Position;
                             preserves q: Position);
    ...
end Location_Linking_Template;
```

Redirect Link

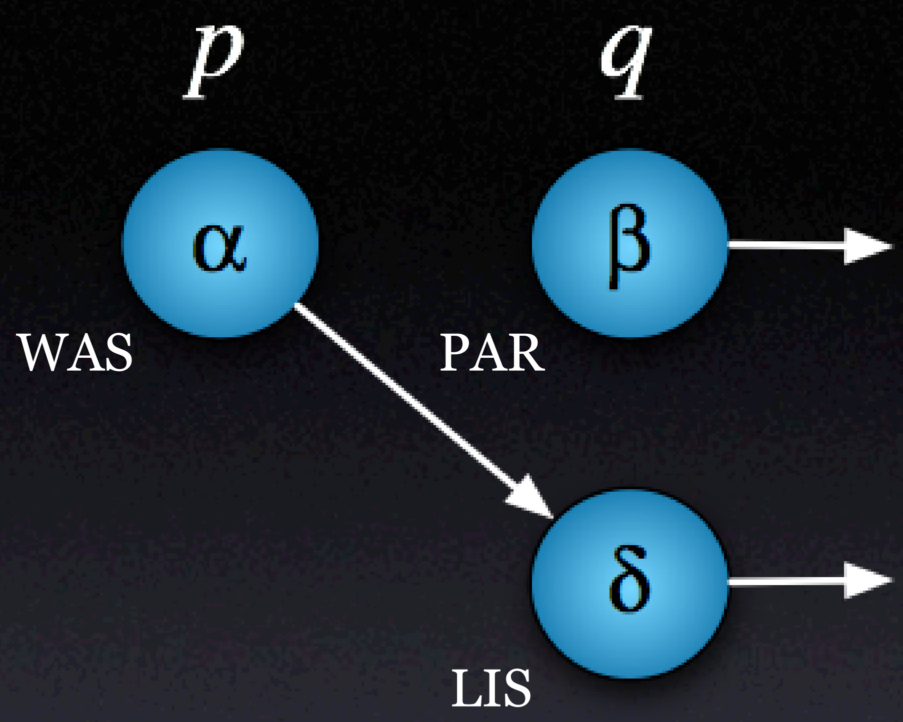
Operation Redirect_Link(**preserves** p: Position;
preserves q: Position);
updates Target;
requires Is_Taken(p);
ensures ∀r: Location,

$$\text{Target}(r) = \begin{cases} q & \textbf{if } r = p \\ \#\text{Target}(r) & \textbf{otherwise} \end{cases};$$



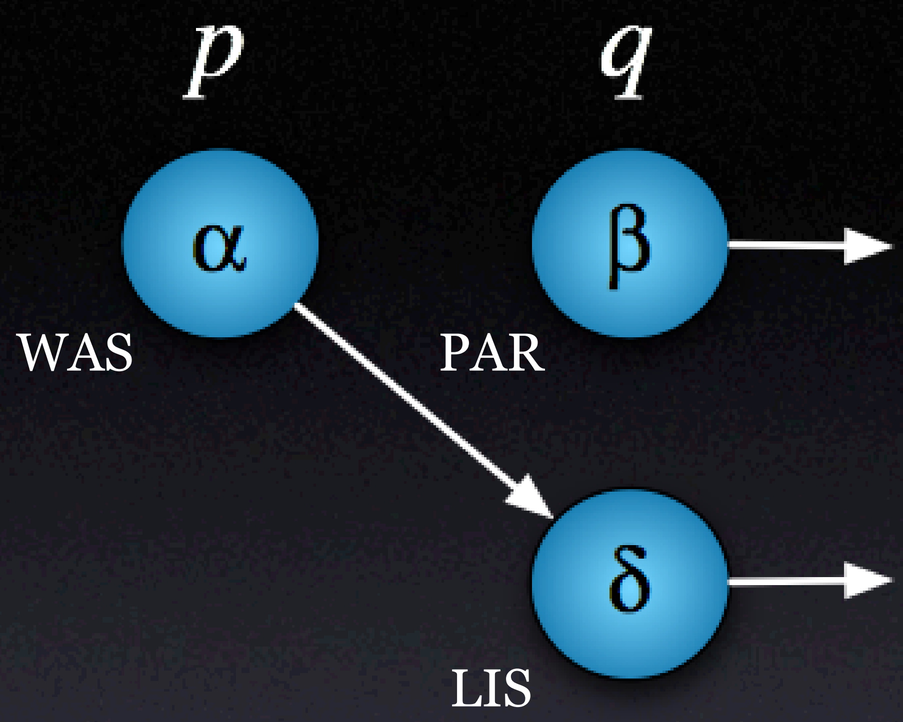
before

Redirect_Link(p, q)



before

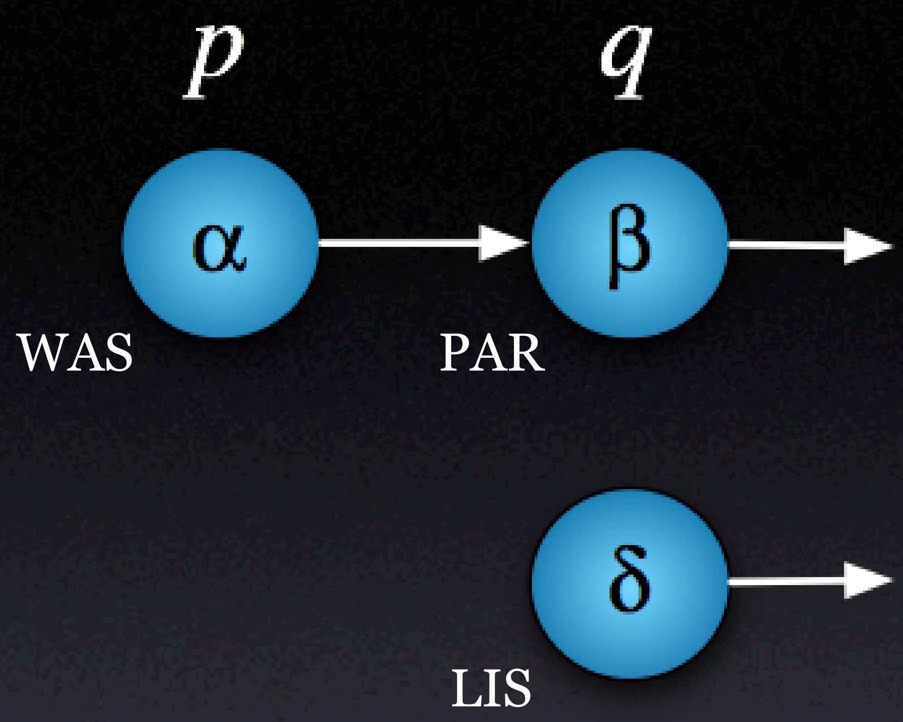
Redirect_Link(p, q)



before

Target = { WAS $\mapsto$ LIS, … }
Contents = { WAS $\mapsto$ α, PAR $\mapsto$ β, LIS $\mapsto$ δ, … }
Is_Taken = { WAS $\mapsto$ **true**, PAR $\mapsto$ **true**, LIS $\mapsto$ **true**, … }
p = WAS
q = PAR

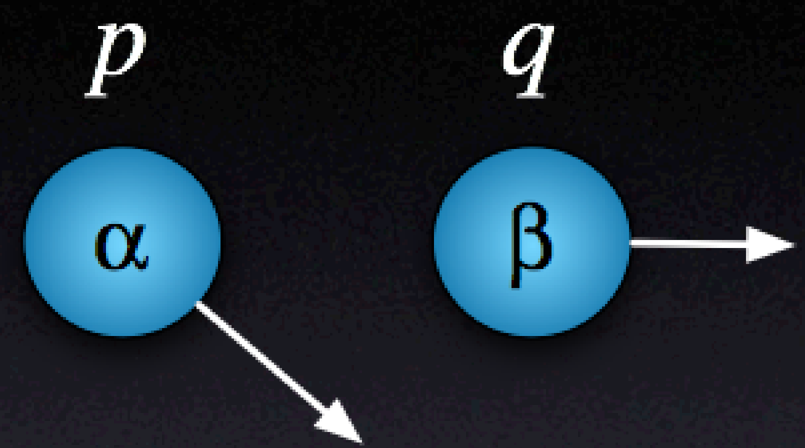
Redirect_Link(p, q)



Target = { WAS $\mapsto$ PAR, … }
Contents = { WAS $\mapsto$ α,
PAR $\mapsto$ β,
LIS $\mapsto$ δ, … }
Is_Taken = { WAS $\mapsto$ **true**,
PAR $\mapsto$ **true**,
LIS $\mapsto$ **true**, … }
p = WAS
q = PAR

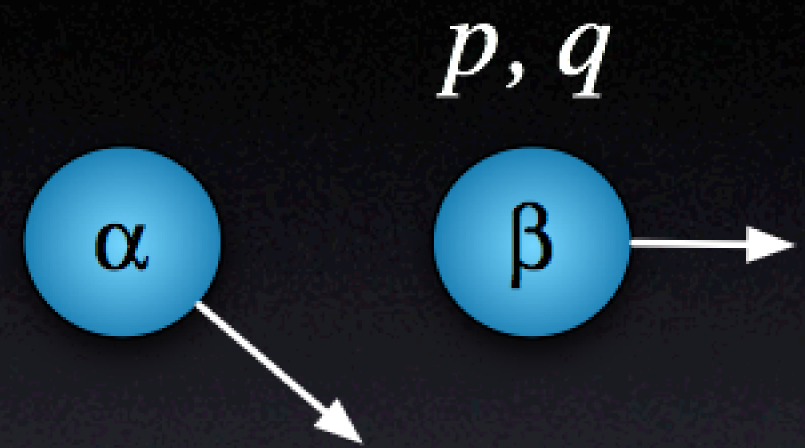
after

Redirect_Link(p, q)



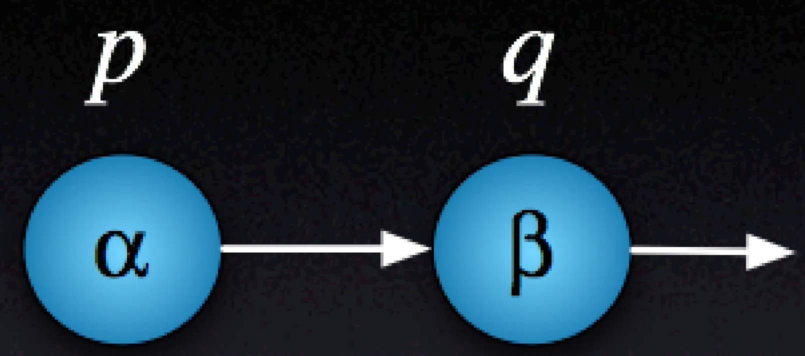
before

Relocate(p, q)



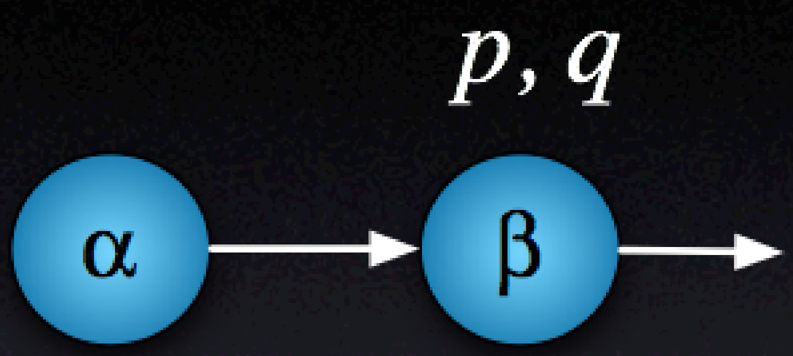
after

Relocate(p, q)



before

Follow_Link(p)



after

Follow_Link(p)

Implementation

Operation invocations such as

- Relocate(p, q)
- Follow_Link(p)
- Redirect_Link(p, q)

are implemented internally by copying a memory address, not by invoking an operation.

Splice Operation

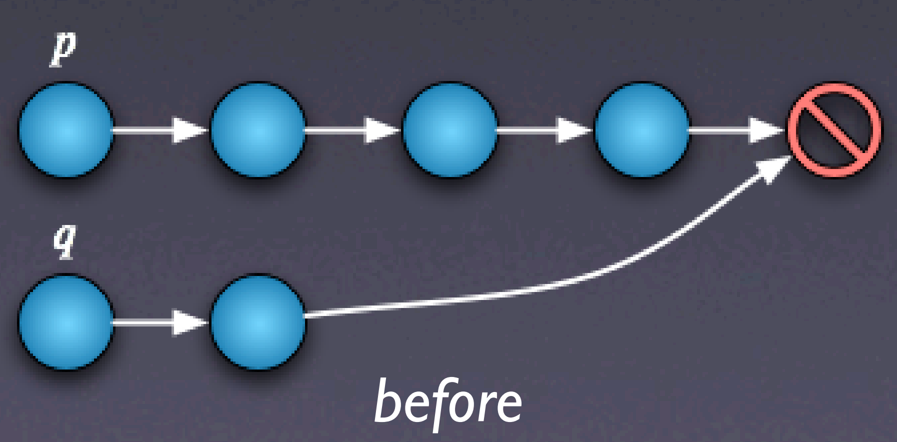
Operation Splice (**preserves** p: Position; **clears** q: Position);
  updates Target;

- **precondition:** p and q point to disjoint and acyclic singly-linked lists of locations, and p's list is at least as long as q's
- **postcondition:** p's resulting list is an interleaving of q's incoming list with the first locations of p's incoming list.

Splice Operation

Operation Splice (**preserves** p: Position; **clears** q: Position);
updates Target;

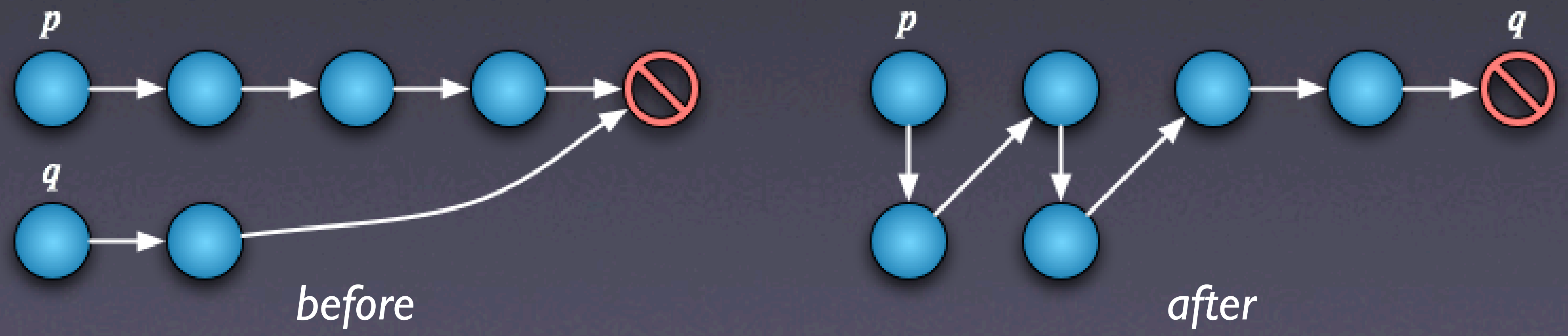
- **precondition:** p and q point to disjoint and acyclic singly-linked lists of locations, and p's list is at least as long as q's
- **postcondition:** p's resulting list is an interleaving of q's incoming list with the first locations of p's incoming list.



Splice Operation

Operation Splice (**preserves** p: Position; **clears** q: Position);
    updates Target;

- **precondition:** p and q point to disjoint and acyclic singly-linked lists of locations, and p's list is at least as long as q's
- **postcondition:** p's resulting list is an interleaving of q's incoming list with the first locations of p's incoming list.

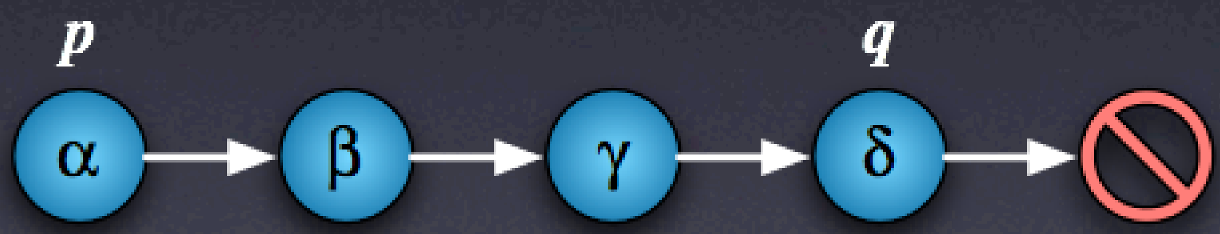


Definitions

Is_Reachable_in (hops: **N**; p, q: Location): **B**

Is_Reachable (p, q: Location): **B**

Distance(p, q: Location): **N**

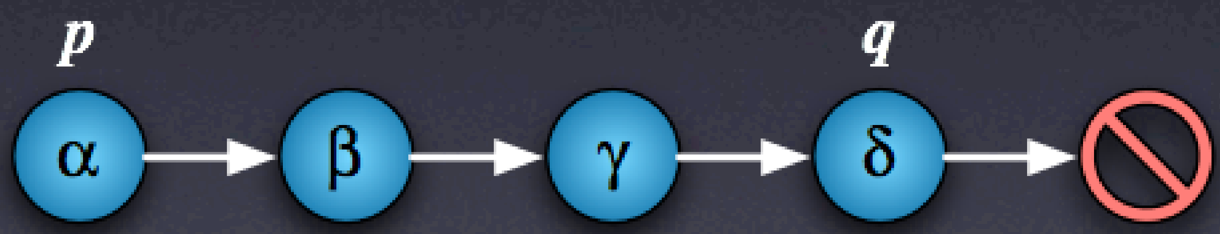


Definitions

Is_Reachable_in (hops: **N**; p, q: Location): **B**

Is_Reachable (p, q: Location): **B**

Distance(p, q: Location): **N**



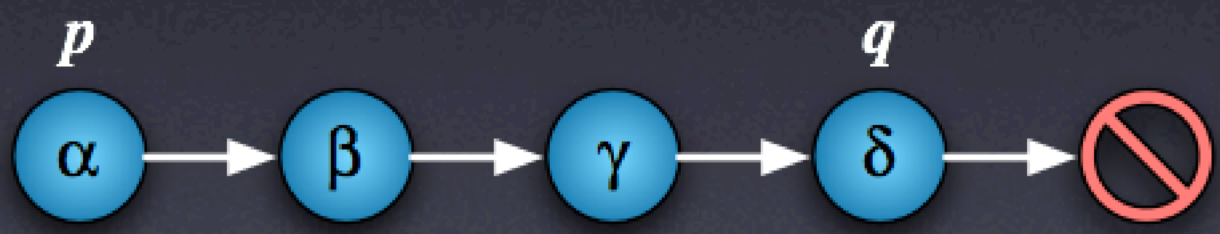
Is_Reachable_in(3, p, q)

Definitions

Is_Reachable_in (hops: **N**; p, q: Location): **B**

Is_Reachable (p, q: Location): **B**

Distance(p, q: Location): **N**



Is_Reachable_in(3, p, q)

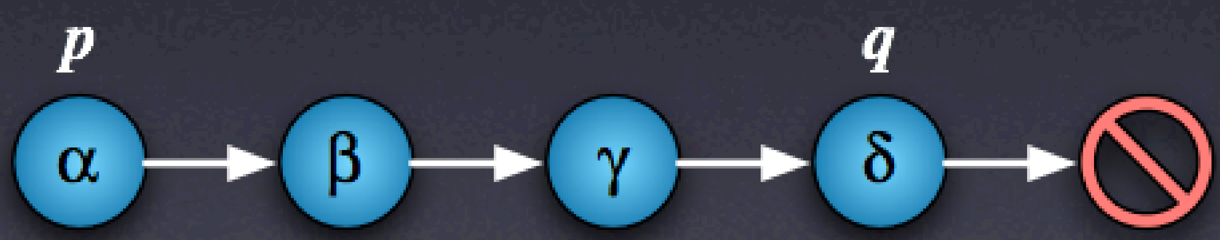
Is_Reachable(p, Void)

Definitions

Is_Reachable_in (hops: **N**; p, q: Location): **B**

Is_Reachable (p, q: Location): **B**

Distance(p, q: Location): **N**



Is_Reachable_in(3, p, q)

Is_Reachable(p, Void)

Distance(q, Void) = 1

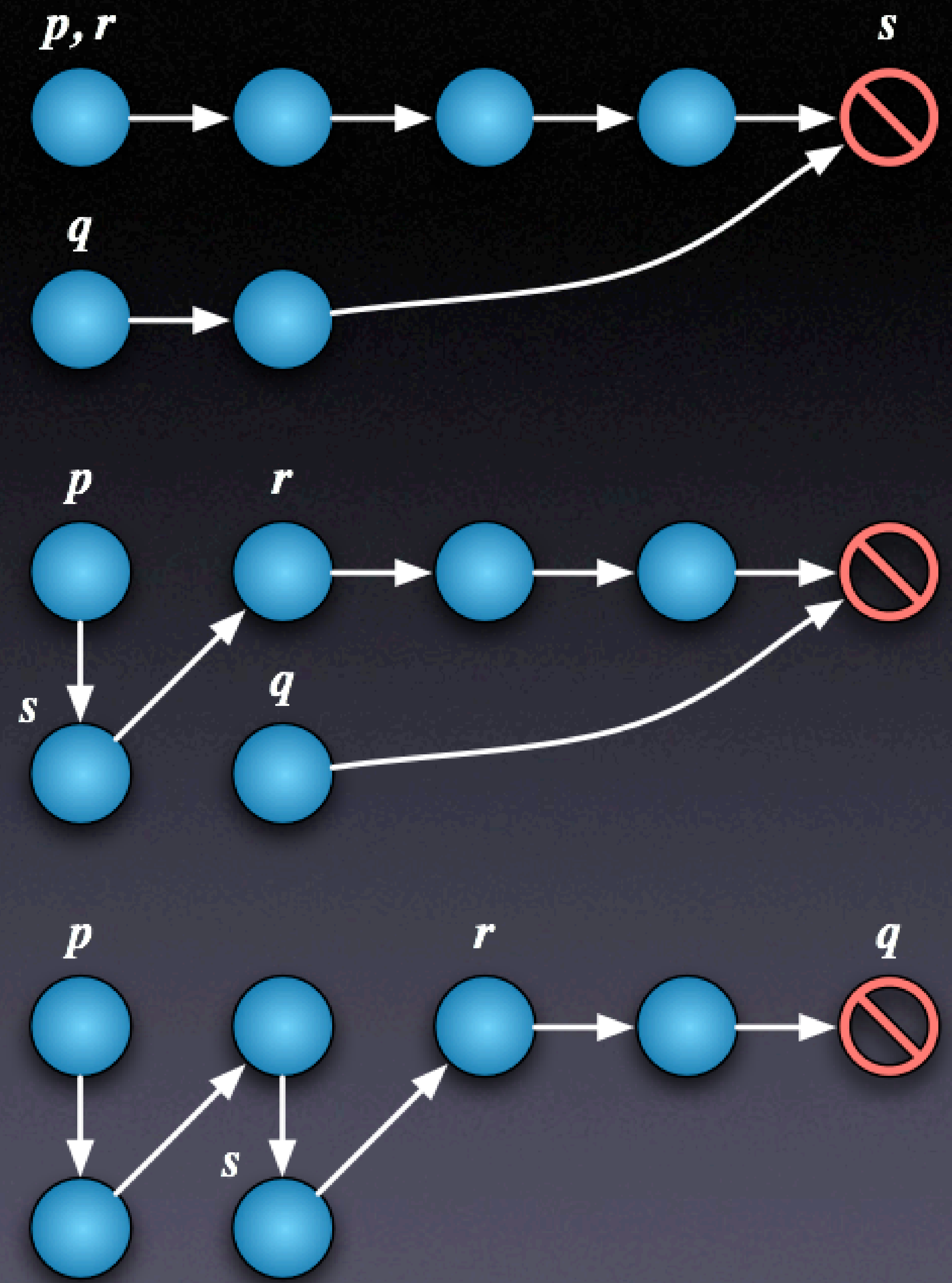
Lightweight Specification

Operation Splice (**preserves** p: Position; **clears** q: Position);
    updates Target;
    requires ($\exists$ k_1, k_2 : **N** $\ni$
            Is_Reachable_in(k_1, p, Void) **and**
            Is_Reachable_in(k_2, q, Void) **and**
            $k_2 \leq k_1$) **and**
        ($\forall$ r : Location,
            if Is_Reachable(p, r) **and** Is_Reachable(q, r)
            then r = Void);
    ensures Is_Reachable(p, Void);

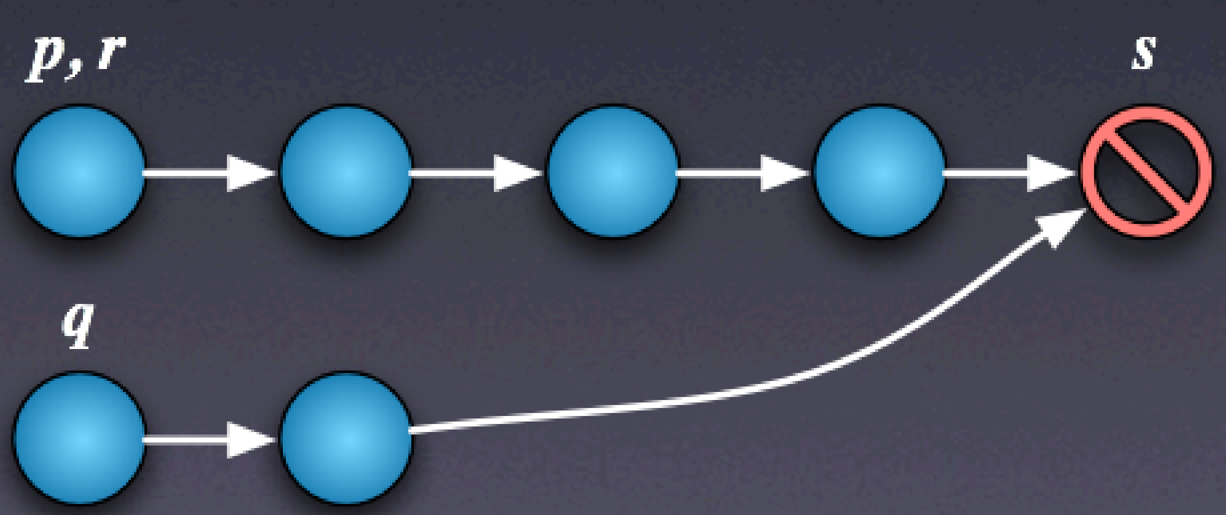
Splice Procedure

```
Operation Splice (preserves p: Position; clears q: Position );
    updates Target;
Procedure
    Var r: Position;
    Var s: Position;
    Relocate(r, p);
    While (not Is_At_Void(q))
    do
        Relocate(s, r);
        Follow_Link(r);
        Redirect_Link(s, q);
        Follow_Link(s);
        Follow_Link(q);
        Redirect_Link(s, r);
    end;
end Splice;
```

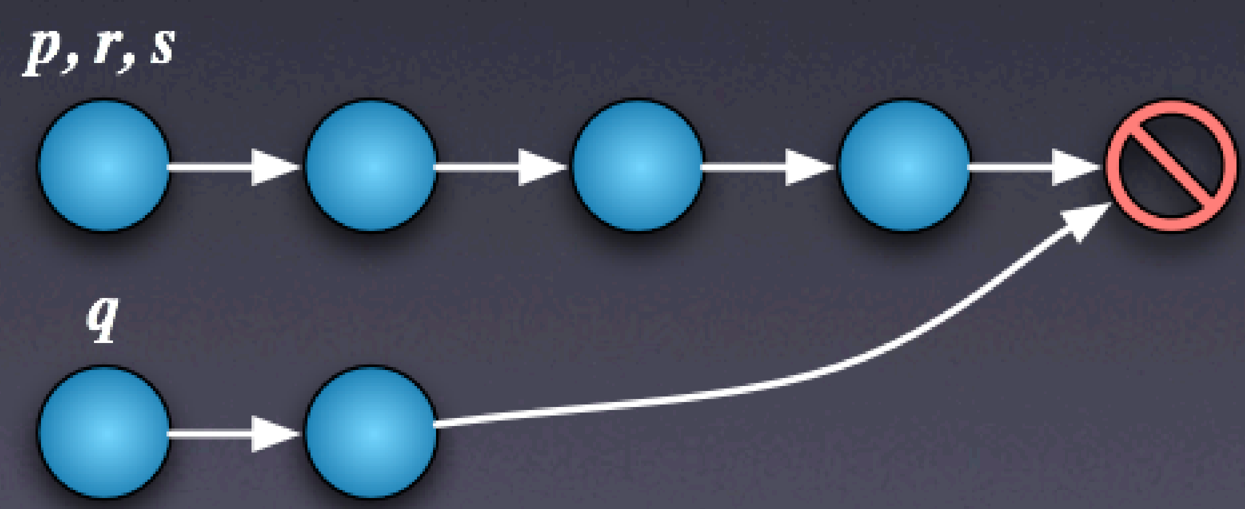
```
While (not Is_At_Void(q))
do
    Relocate(s, r);
    Follow_Link(r);
    Redirect_Link(s, q);
    Follow_Link(s);
    Follow_Link(q);
    Redirect_Link(s, r);
end;
```



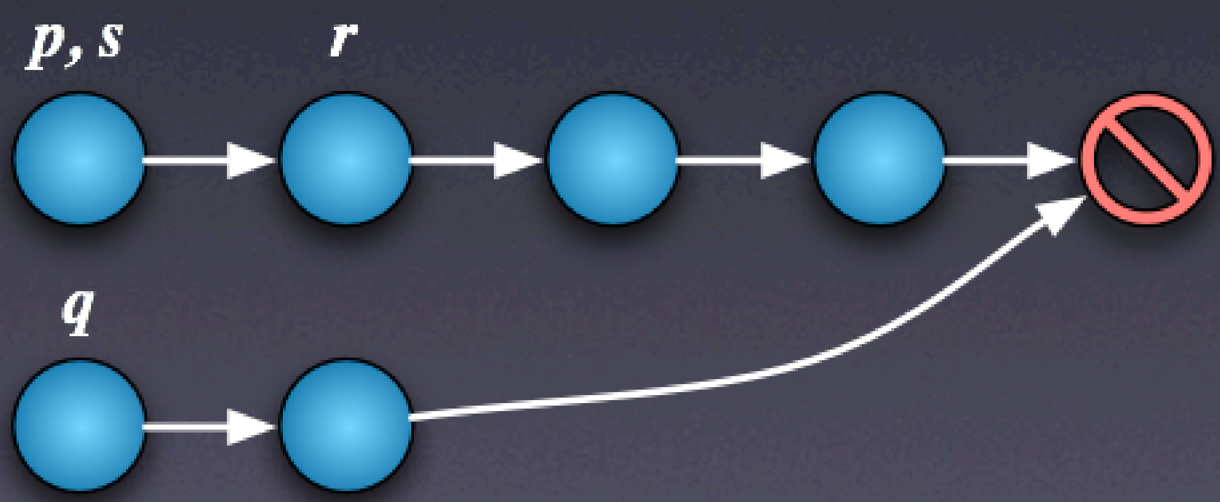
```
While (not Is_At_Void(q))
do
    Relocate(s, r);
    Follow_Link(r);
    Redirect_Link(s, q);
    Follow_Link(s);
    Follow_Link(q);
    Redirect_Link(s, r);
end;
```



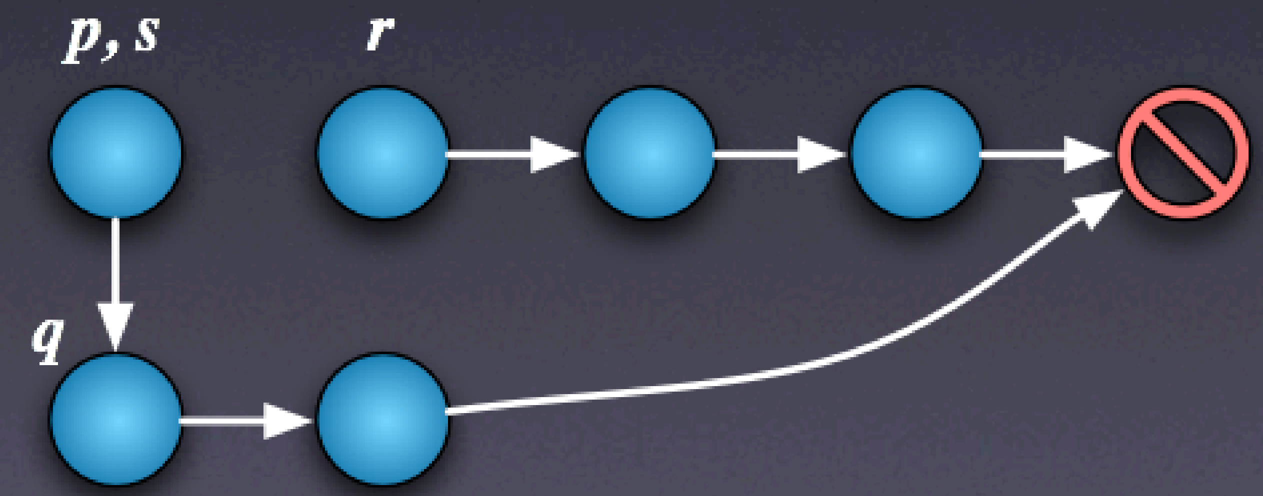
```
While (not Is_At_Void(q))
do
    Relocate(s, r);
    Follow_Link(r);
    Redirect_Link(s, q);
    Follow_Link(s);
    Follow_Link(q);
    Redirect_Link(s, r);
end;
```



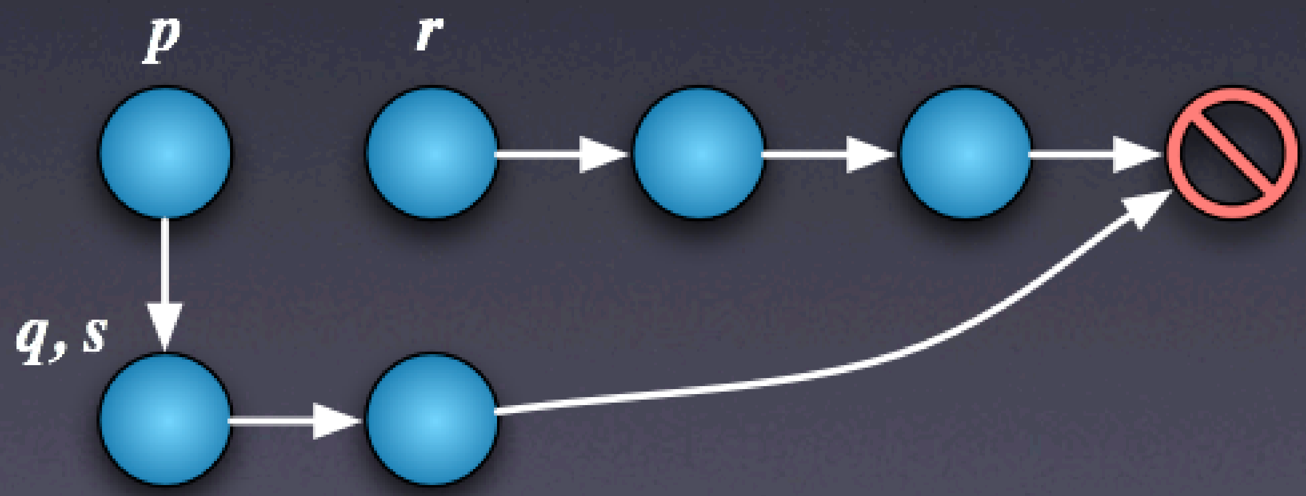
```
While (not Is_At_Void(q))
do
    Relocate(s, r);
    Follow_Link(r);
    Redirect_Link(s, q);
    Follow_Link(s);
    Follow_Link(q);
    Redirect_Link(s, r);
end;
```



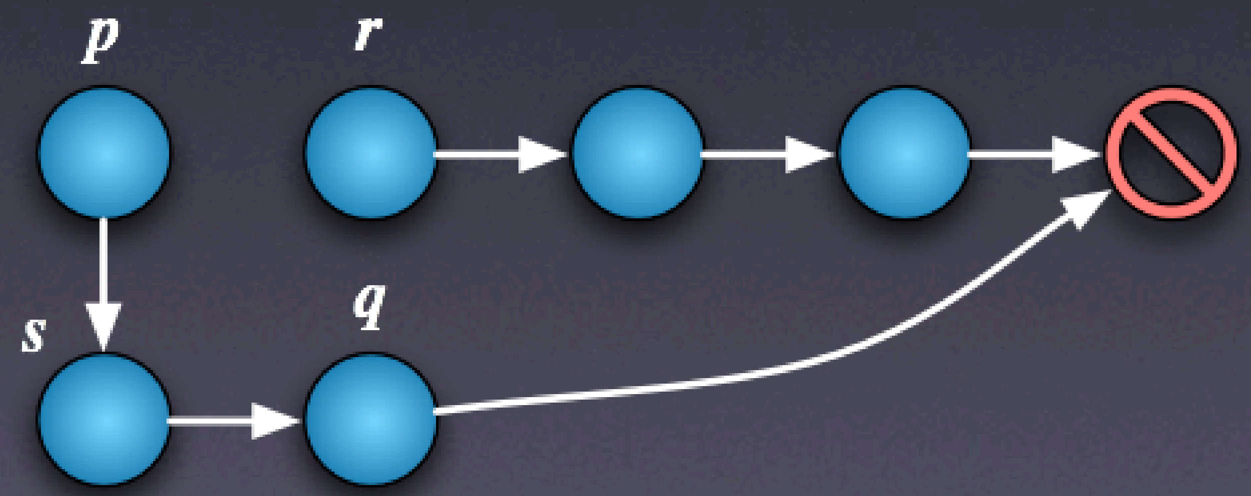
```
While (not Is_At_Void(q))
do
    Relocate(s, r);
    Follow_Link(r);
    Redirect_Link(s, q);
    Follow_Link(s);
    Follow_Link(q);
    Redirect_Link(s, r);
end;
```



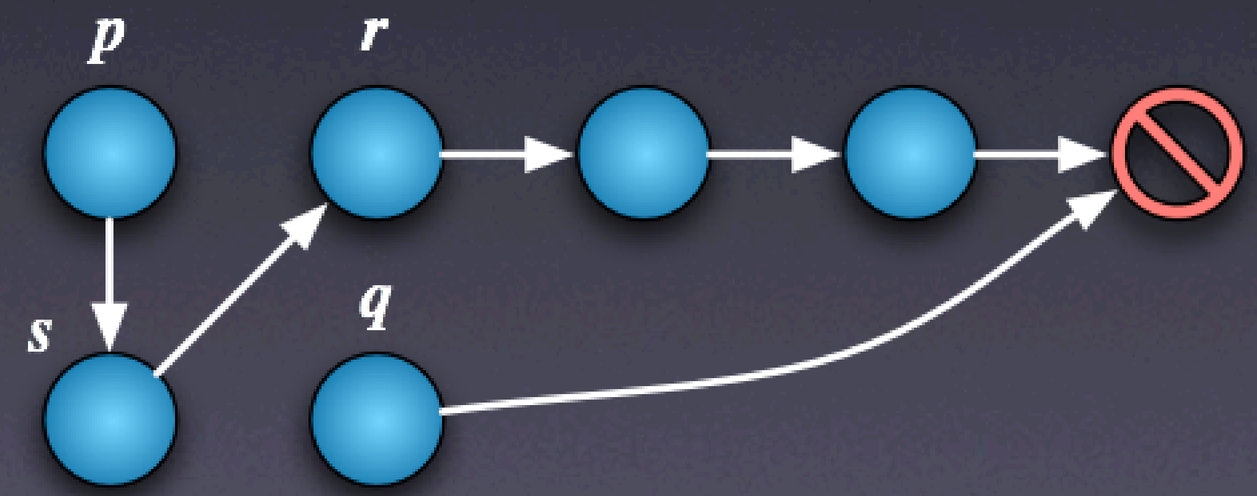
```
While (not Is_At_Void(q))
do
    Relocate(s, r);
    Follow_Link(r);
    Redirect_Link(s, q);
    Follow_Link(s);
    Follow_Link(q);
    Redirect_Link(s, r);
end;
```



```
While (not Is_At_Void(q))
do
    Relocate(s, r);
    Follow_Link(r);
    Redirect_Link(s, q);
    Follow_Link(s);
    Follow_Link(q);
    Redirect_Link(s, r);
end;
```



```
While (not Is_At_Void(q))
do
    Relocate(s, r);
    Follow_Link(r);
    Redirect_Link(s, q);
    Follow_Link(s);
    Follow_Link(q);
    Redirect_Link(s, r);
end;
```



Splice Procedure

```
Procedure
    Var r: Position;
    Var s: Position;
    Relocate(r, p);
    While (not Is_At_Void(q))

        maintaining ???
    do
        Relocate(s, r);
        Follow_Link(r);
        Redirect_Link(s, q);
        Follow_Link(s);
        Follow_Link(q);
        Redirect_Link(s, r);
    end;
end Splice;
```

```
Operation Splice (preserves p: Position;
                  clears q: Position);
    updates Target;
    requires ( ∃ k₁, k₂ : N ∋
            Is_Reachable_in(k₁, p, Void) and
            Is_Reachable_in(k₂, q, Void) and
            k₂ ≤ k₁ ) and
        ( ∀ r : Location,
            if Is_Reachable(p, r) and Is_Reachable(q, r)
            then r = Void );
    ensures Is_Reachable(p, Void);
```

Splice Procedure

```
Procedure
    Var r: Position;
    Var s: Position;
    Relocate(r, p);
    While (not Is_At_Void(q))

        maintaining Is_Reachable(p, Void);
    do
        Relocate(s, r);
        Follow_Link(r);
        Redirect_Link(s, q);
        Follow_Link(s);
        Follow_Link(q);
        Redirect_Link(s, r);
    end;
end Splice;
```

```
Operation Splice (preserves p: Position;
                  clears q: Position);
    updates Target;
    requires ( ∃ k₁, k₂ : N ∋
            Is_Reachable_in(k₁, p, Void) and
            Is_Reachable_in(k₂, q, Void) and
            k₂ ≤ k₁ ) and
        ( ∀ r : Location,
            if Is_Reachable(p, r) and Is_Reachable(q, r)
            then r = Void );
    ensures Is_Reachable(p, Void);
```

Splice Procedure

```
Procedure
    Var r: Position;
    Var s: Position;
    Relocate(r, p);
    While (not Is_At_Void(q))
        decreasing ???
        maintaining Is_Reachable(p, Void);
    do
        Relocate(s, r);
        Follow_Link(r);
        Redirect_Link(s, q);
        Follow_Link(s);
        Follow_Link(q);
        Redirect_Link(s, r);
    end;
end Splice;
```

Operation Splice (**preserves** p: Position;
clears q: Position);
updates Target;
requires ($\exists\ k_1, k_2 : \mathbf{N} \ni$
Is_Reachable_in(k_1, p, Void) **and**
Is_Reachable_in(k_2, q, Void) **and**
$k_2 \leq k_1$) **and**
($\forall$ r : Location,
if Is_Reachable(p, r) **and** Is_Reachable(q, r)
then r = Void);
ensures Is_Reachable(p, Void);

Splice Procedure

```
Procedure
    Var r: Position;
    Var s: Position;
    Relocate(r, p);
    While (not Is_At_Void(q))
        decreasing Distance(q, Void);
        maintaining Is_Reachable(p, Void);
    do
        Relocate(s, r);
        Follow_Link(r);
        Redirect_Link(s, q);
        Follow_Link(s);
        Follow_Link(q);
        Redirect_Link(s, r);
    end;
end Splice;
```

```
Operation Splice (preserves p: Position;
                  clears q: Position);
    updates Target;
    requires ( ∃ k₁, k₂ : N ∋
            Is_Reachable_in(k₁, p, Void) and
            Is_Reachable_in(k₂, q, Void) and
            k₂ ≤ k₁ ) and
        ( ∀ r : Location,
            if Is_Reachable(p, r) and Is_Reachable(q, r)
            then r = Void );
    ensures Is_Reachable(p, Void);
```

Loop Invariant Proof

```
Procedure
    Var r: Position;
    Var s: Position;
    Relocate(r, p);
    While (not Is_At_Void(q))
        decreasing Distance(q, Void);
        maintaining Is_Reachable(p, Void);
    do
        Relocate(s, r);
        Follow_Link(r);
        Redirect_Link(s, q);
        Follow_Link(s);
        Follow_Link(q);
        Redirect_Link(s, r);
    end;
end Splice;
```

1. **Initialization** – Is the invariant true at the start of the loop?
2. **Maintenance** – Is the invariant true from one iteration to the next?
3. **Termination** – Does the invariant allow you to prove what you need to?

Loop Invariant Proof

Lemma #1: Is_Reachable(q, Void);

Lemma #2: Is_Reachable(r, Void);

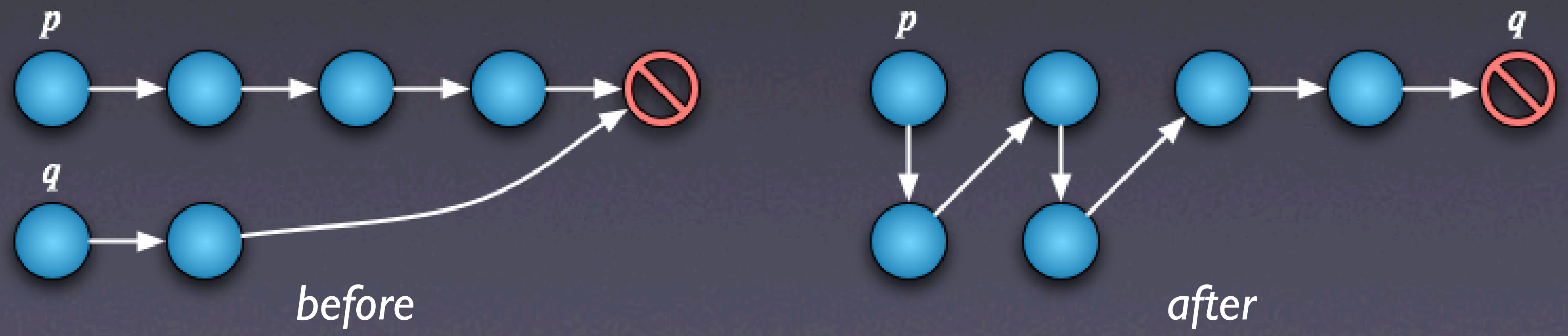
Lemma #3: Is_Reachable(p, r) **or** Is_Reachable(p, q);

```
While (not Is_At_Void(q))
do
    Relocate(s, r);
    Follow_Link(r);
    Redirect_Link(s, q);
    Follow_Link(s);
    Follow_Link(q);
    Redirect_Link(s, r);
end;
```

Splice Operation

Operation Splice (**preserves** p: Position; **clears** q: Position);
    updates Target;

- **precondition:** p and q point to disjoint and acyclic singly-linked lists of locations, and p's list is at least as long as q's
- **postcondition:** p's resulting list is an interleaving of q's incoming list with the first locations of p's incoming list.



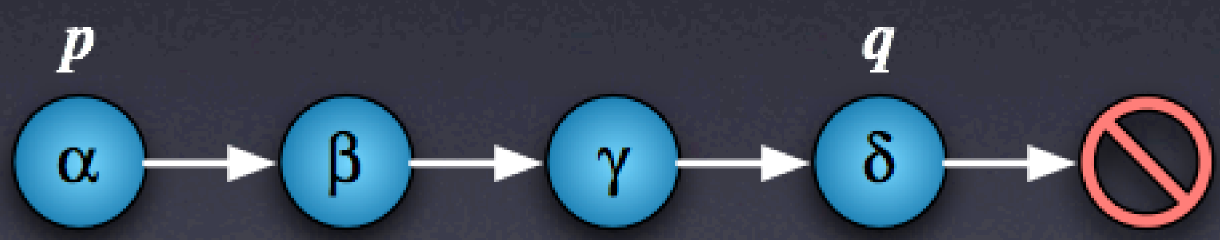
String Notation

The following notations are defined in a module that specifies the properties of mathematical strings.

$\langle x \rangle$	a string containing x
$\alpha \circ \beta$	the concatenation of strings α and β
$\alpha \leq ! \geq (\beta, \gamma)$	asserts that α is a perfect shuffle of strings β and γ

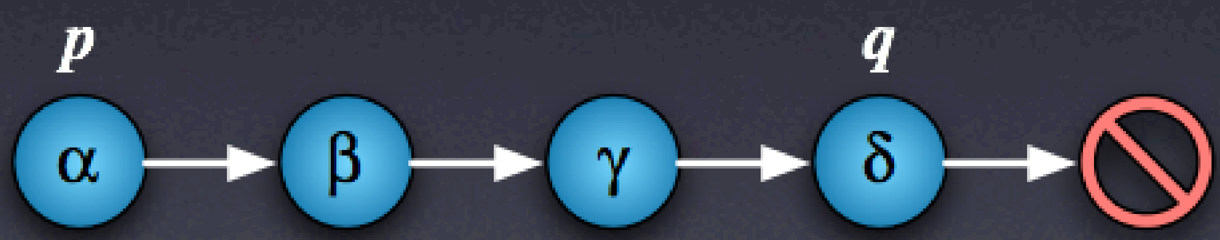
Definitions

Is_Info_Str(p, q: Location, α: Str(Info): **B**



Definitions

Is_Info_Str(p, q: Location, α: Str(Info): **B**



Is_Info_Str(p, q, ⟨ α, β, γ ⟩)

Heavyweight Specification

Operation Splice (**preserves** p: Position; **clears** q: Position);

updates Target;

requires *** Same as in lightweight specification ***

ensures (∀ t: Location, **if not** Is_Reachable(#p, t) **and**

not Is_Reachable(#q, t) **then** Target(t) = #Target(t)) **and**

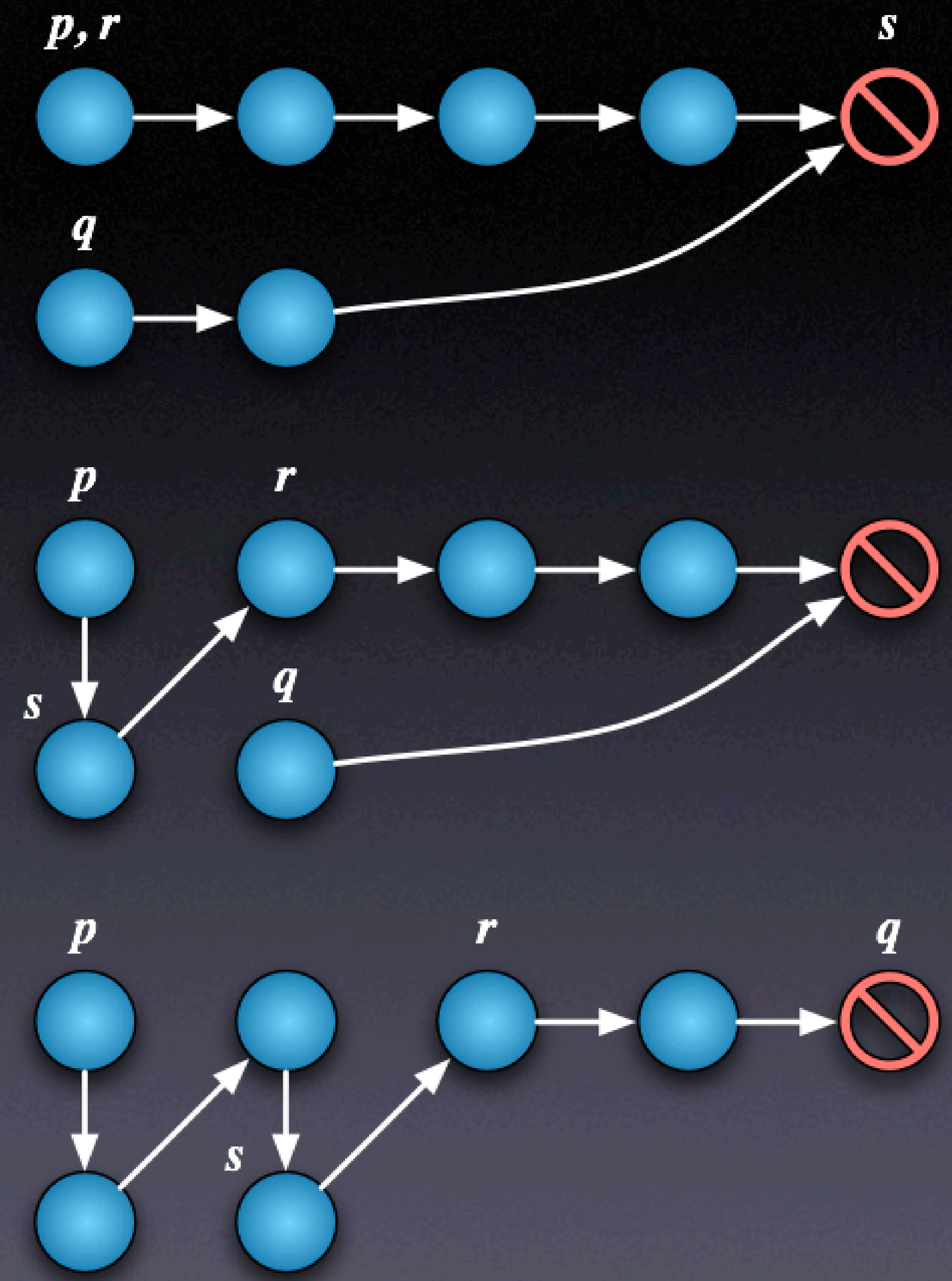
(∀ p0, old_p0, old_q0: Str(Info),

if (Is_Info_Str(p, Void, p0) **and**

Is_Info_Str(#p, Void, old_p0) **and**

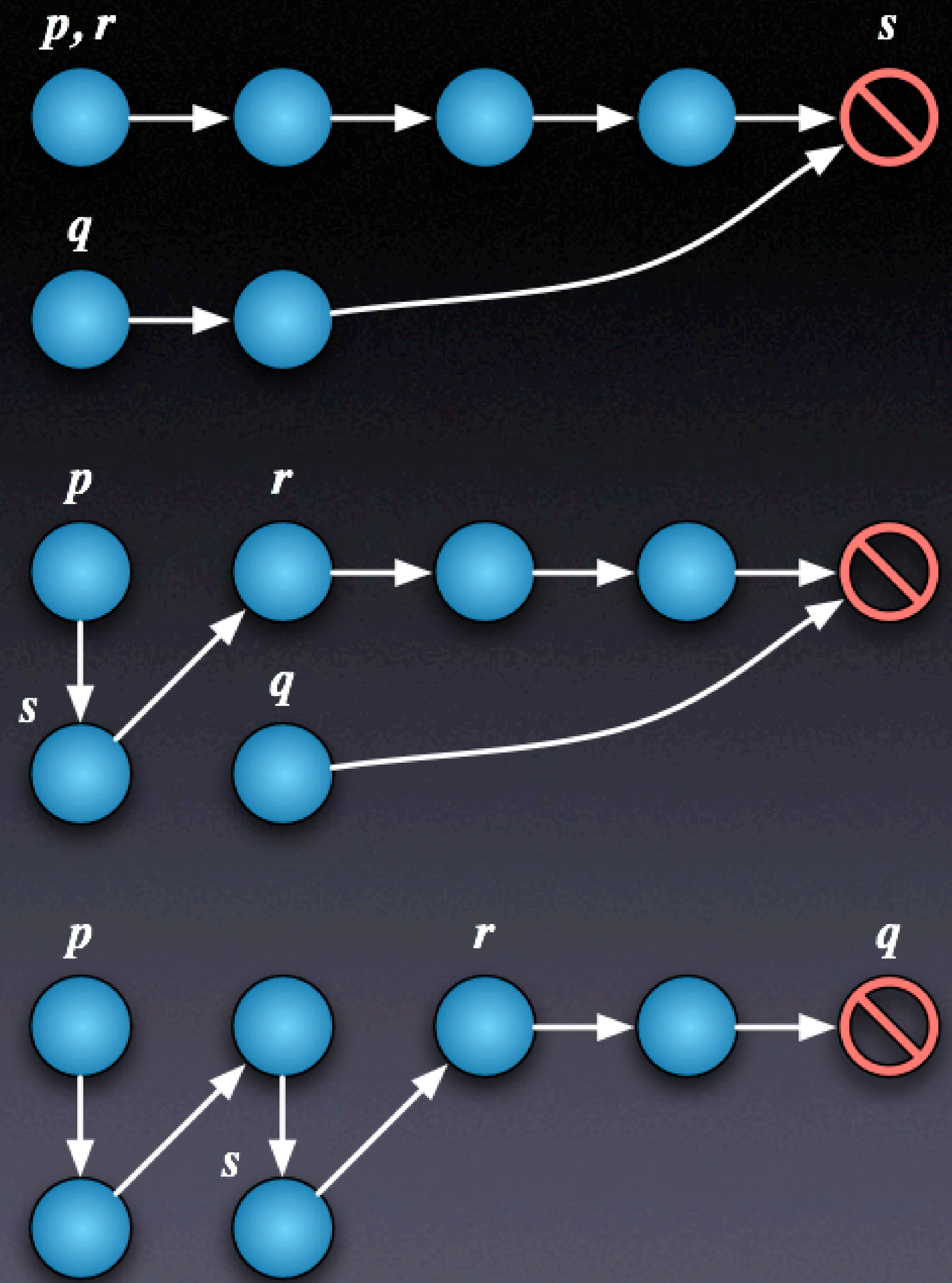
Is_Info_Str(#q, Void, old_q0))

then p0 ≤!≥ (old_p0, old_q0));


p, r
s
q
p
r
s
q
p
r
q
s

Let *rq_shuffle* be a perfect shuffle of info strings *r0* and *q0*.

Then *pr* ○ *rq_shuffle* is a perfect shuffle of info strings *old_p0* and *old_q0*.



Loop Invariant

maintaining (∀ t: Location, **if not** Is_Reachable(#p, t) **and**
not Is_Reachable(#q, t) **then** Target(t) = #Target(t)) **and**
(∀ pr, r0, q0, old_p0, old_q0, rq_shuffle: Str(Info),
if (Is_Info_Str(p, r, p0) **and**
Is_Info_Str(r, Void, r0) **and** Is_Info_Str(q, Void, q0)
Is_Info_Str(#p, Void, old_p0) **and**
Is_Info_Str(#q, Void, old_q0) **and**
rq_shuffle ≤!≥ (r0, q0)
then pr ○ rq_shuffle ≤!≥ (old_p0, old_q0));

Loop Invariant

```
maintaining ( ∀ t: Location, if not Is_Reachable(#p, t) and
    not Is_Reachable(#q, t) then Target(t) = #Target(t) ) and
  ( ∀ pr, ro, qo, old_po, old_qo, rq_shuffle: Str(Info),
    if ( Is_Info_Str(p, r, po) and
          Is_Info_Str(r, Void, ro) and Is_Info_Str(q, Void, qo)
          Is_Info_Str(#p, Void, old_po) and
          Is_Info_Str(#q, Void, old_qo) and
          rq_shuffle ≤!≥ (ro, qo)
    then pr ○ rq_shuffle ≤!≥ (old_po, old_qo) );
```

Sketch of Proof

Initialization

We need to show pr ○ rq_shuffle ≤!≥ (old_p0, old_q0)

Since p = r, it suffices to show

empty_string ○ pq_shuffle ≤!≥ (old_p0, old_q0)

Termination

We know pr ○ rq_shuffle ≤!≥ (old_p0, old_q0)

Since q = Void, rq_shuffle = r0, we know

pr ○ r0 ≤!≥ (old_p0, old_q0)

Sketch of Proof

Maintenance

Assume: $pr \circ rq\_shuffle \leq ! \geq (old\_po, old\_qo)$
Show: $pr' \circ rq\_shuffle' \leq ! \geq (old\_po, old\_qo)$

$ro = \langle x \rangle \circ ro'$
$qo = \langle y \rangle \circ qo'$

$rq\_shuffle = \langle x \rangle \circ \langle y \rangle \circ rq\_shuffle'$

$ps' = pr \circ \langle x \rangle$
$pr' = pr \circ \langle x \rangle \circ \langle y \rangle$

$pr' \circ rq\_shuffle' = pr \circ rq\_shuffle$

```
While (not Is_At_Void(q))
do
    Relocate(s, r);
    Follow_Link(r);
    Redirect_Link(s, q);
    Follow_Link(s);
    Follow_Link(q);
    Redirect_Link(s, r);
end;
```

Summary

Traditional Analysis of Pointers

- Fully automated
- Relatively fast
- Low (but present) false positive rate
- Limited in what it can prove

Specification-Based Approach

- Partly automated
- Requires programmer supplied assertions
- Handles both lightweight and heavyweight specifications

Related Work

- Region-based shape analysis
- Logic of stores
- Constraint solver with Alloy
- ESC/Java
- LOOP compiler
- Safe pointers and checked pointers
- Pointer component

Questions

Additional Slides

Objective

Allow programmers to reason about pointers and programs that involve pointers using the same techniques they use to reason about programs without pointers.

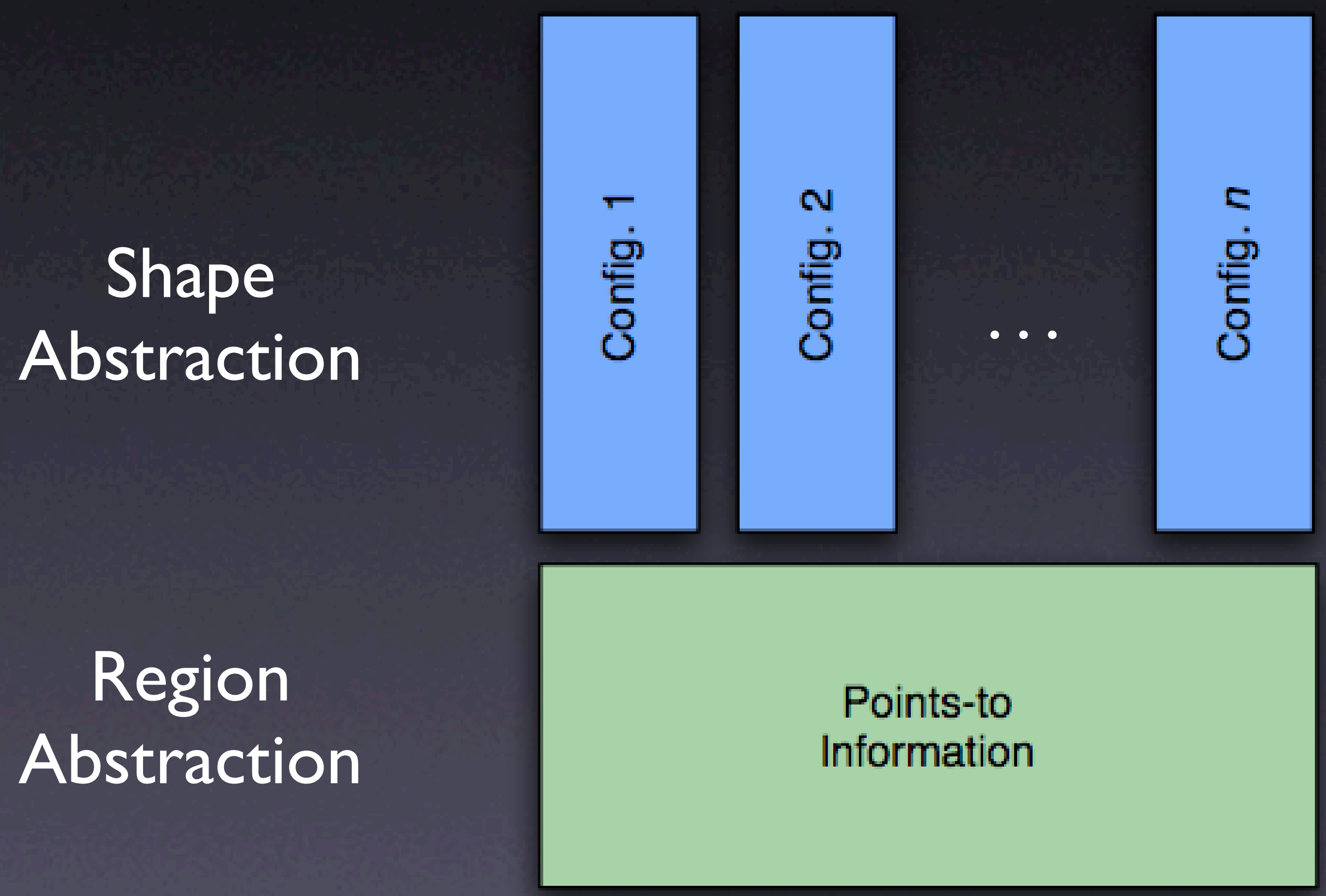
- The technique should not require special language semantics or proof rules that apply only to pointers.
- The technique should not be limited in what it can prove about programs with pointers.

Approach

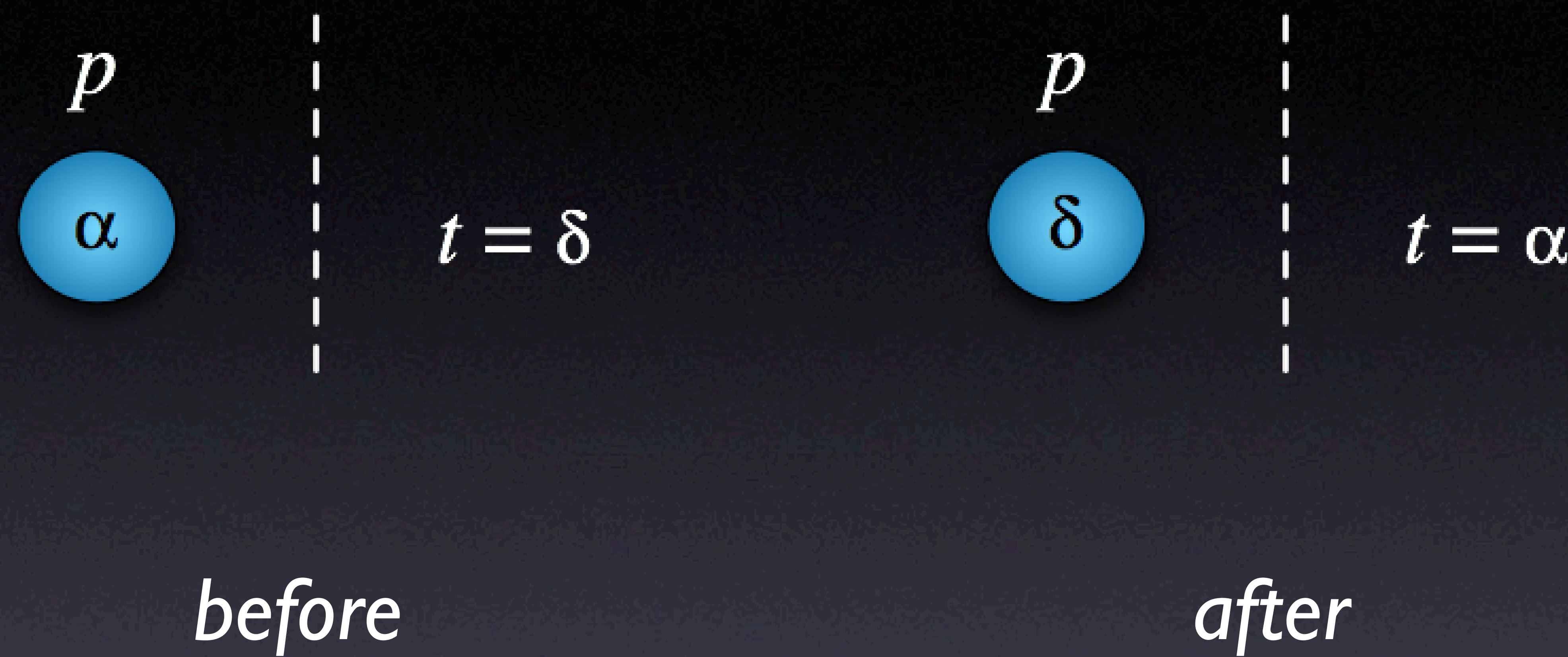
Introduce a fully specified component into the programming language that captures the functional and performance behavior of pointers.

- Programmers can reason about pointers as they reason about any other component.
- For appropriate performance behavior, the compiler must implement the component differently than it implements other components.

Shape Analysis
Memory Abstraction



"Each configuration characterizes the state of one single heap location, called the *tracked location*."



Swap_Contents(p, t)

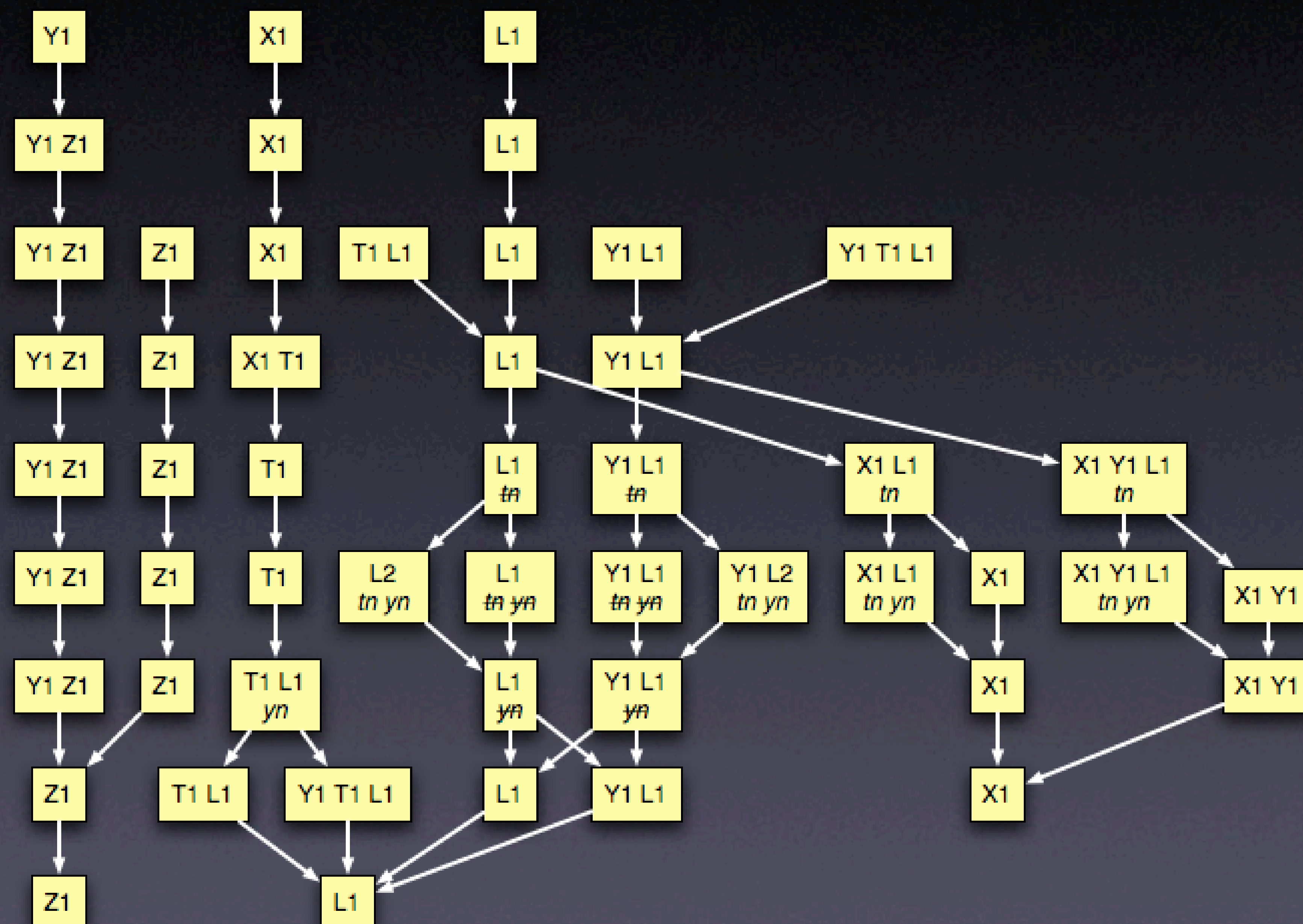
Shape Analysis
Overview

Reason about invariants that describe the "shapes" of dynamic data structures.

- A memory location is not referenced by more than one other location.
- A tree structure is maintained by a program.
- A list does not contain cycles.
- No accesses through dangling references.
- Memory leaks do not occur.

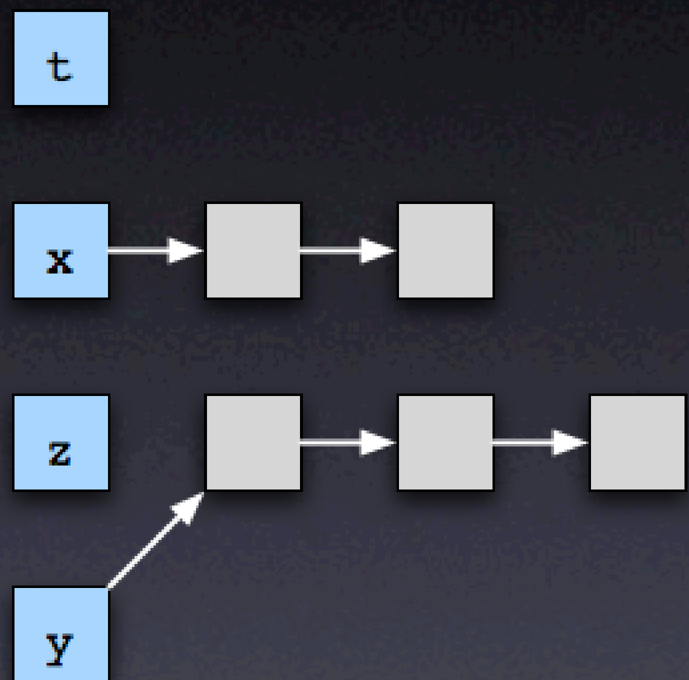
Shape Analysis: Tracking the State

```
08: List *z = y;

09: while (x != NULL)

10:   t = x;

11:   x = t->n;

12:   t->n = y->n;

13:   y->n = t;

14:   y = y->n->n;

16: return z;
```

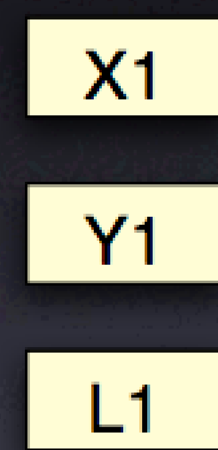


Shape Analysis

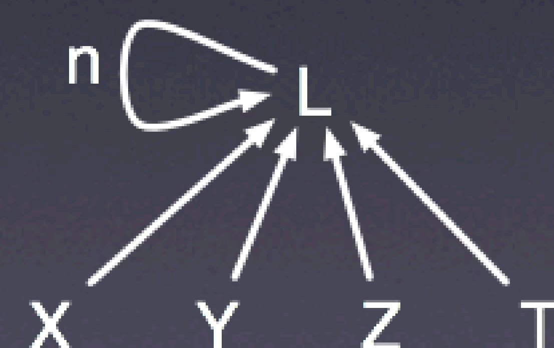
Input state and abstraction



Example state at the beginning of splice



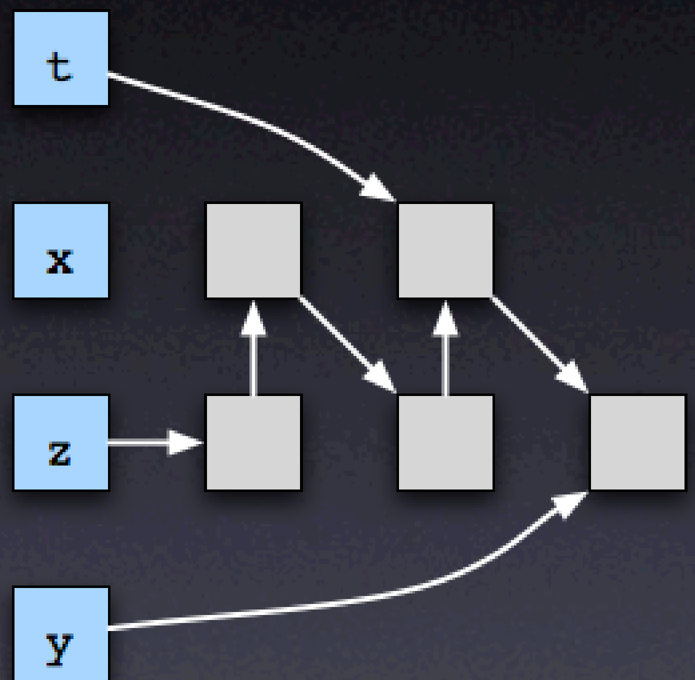
Configuration



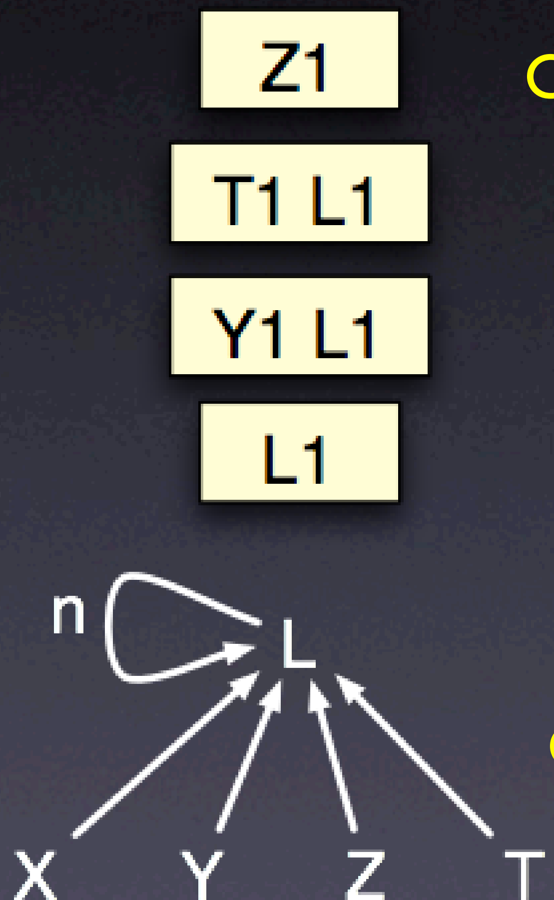
Region Points-to Component

Shape Analysis

Output state and abstraction



Example state at the end of splice

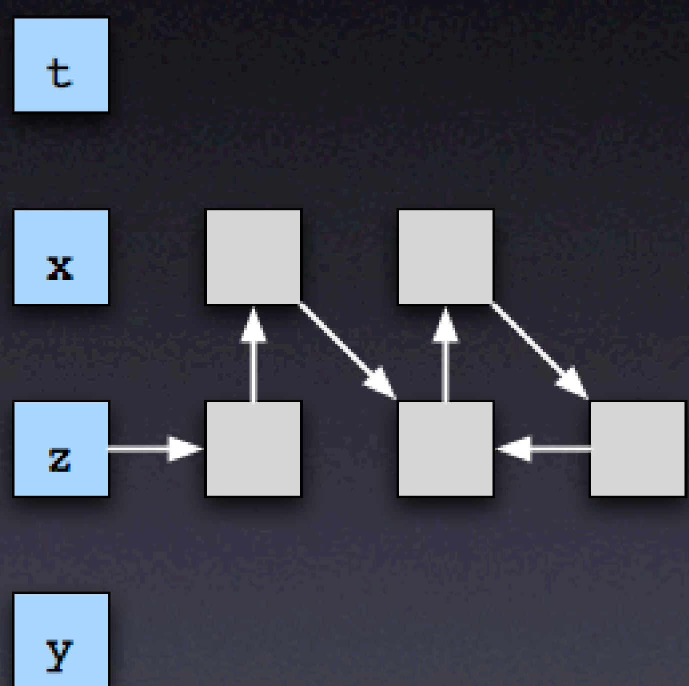


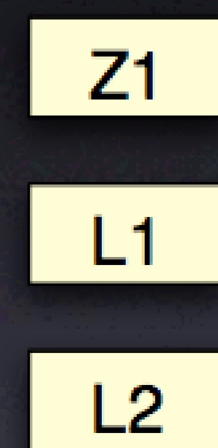
Configuration

Region Points-to Component

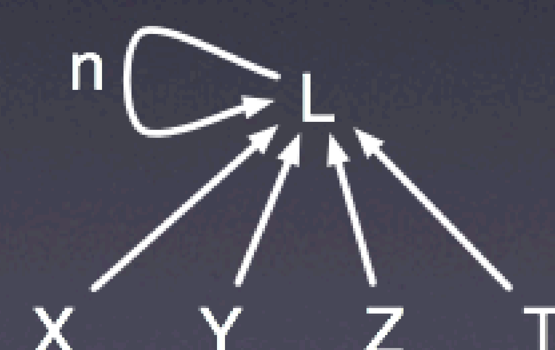
Shape Analysis

L2 presents a problem





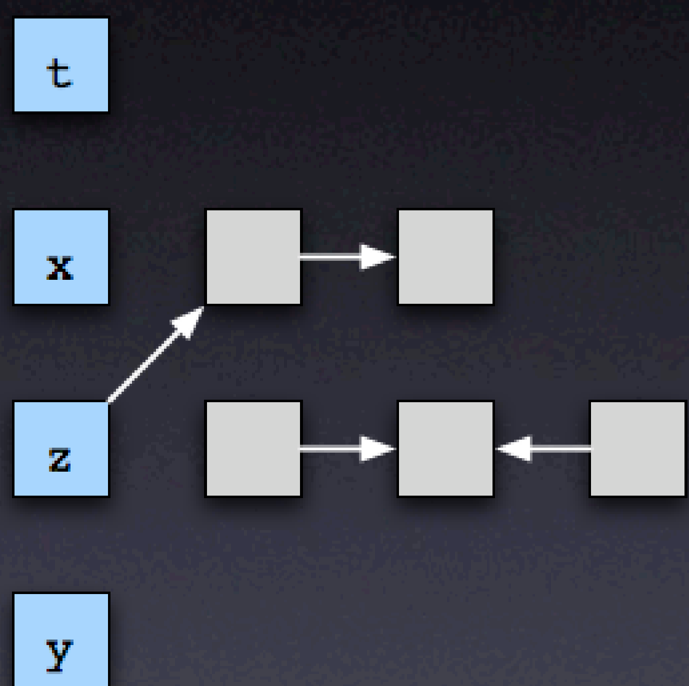
Configuration

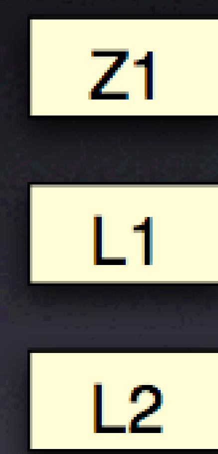


Region
Points-to
Component

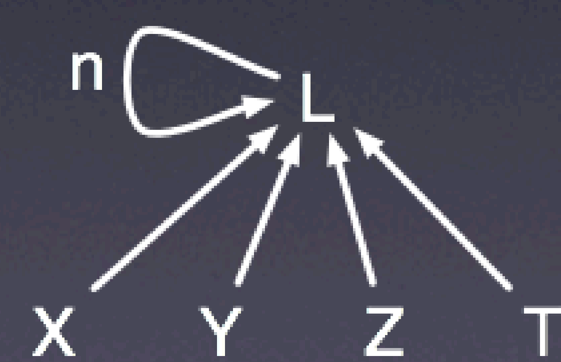
Shape Analysis

L2 with acyclic output



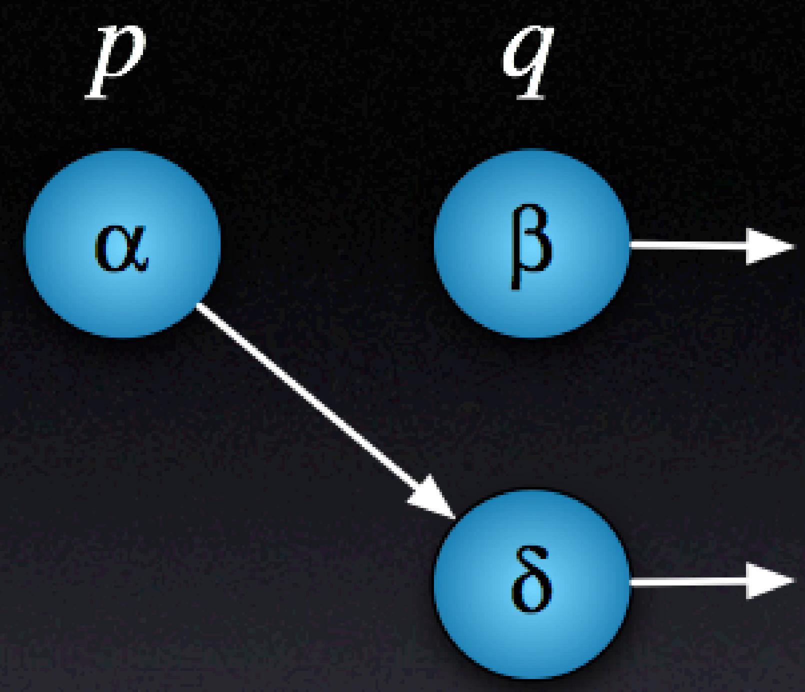


Configuration



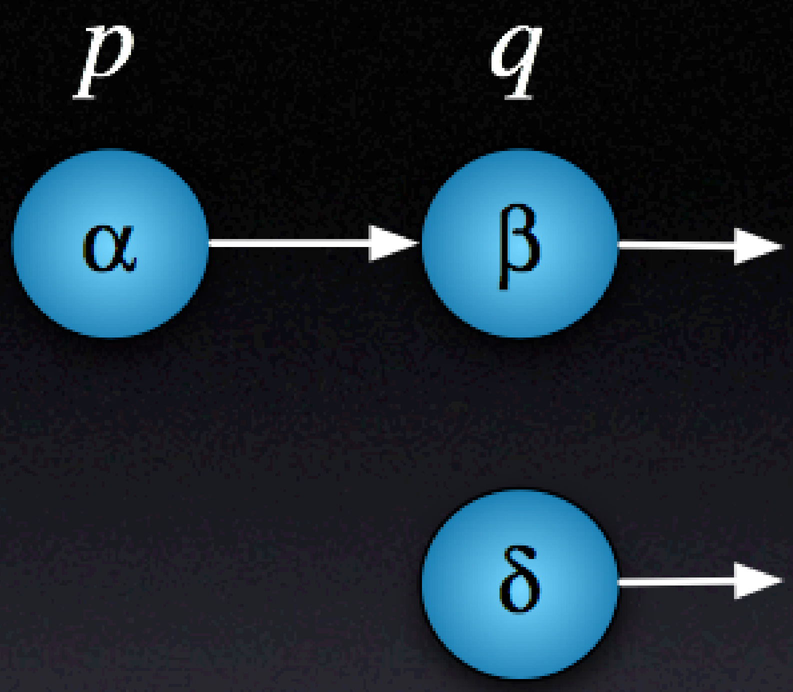
Region
Points-to
Component

Pointer Component



before

Redirect_Link(p, q)



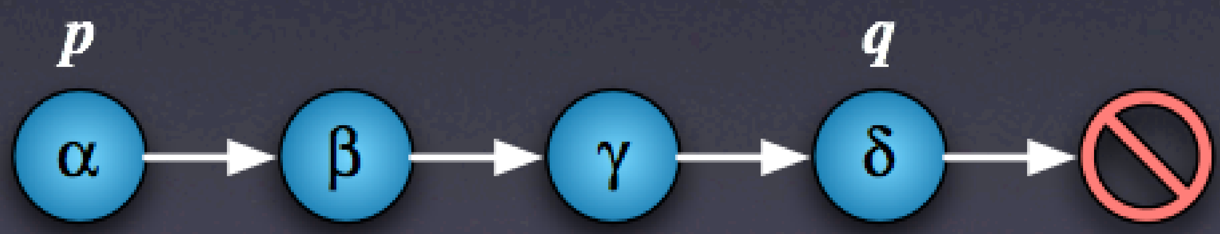
after

Redirect_Link(p, q)

Splice Operation

Definitions

Definition Var Is_Reachable_in (hops: **N**; p, q: Location): **B**
= $\text{Target}^{\text{hops}}(p) = q$ **and** $\forall$ k: **N**, **if** $\text{Target}^{k}(p) = q$ **then** $k \geq \text{hops}$;

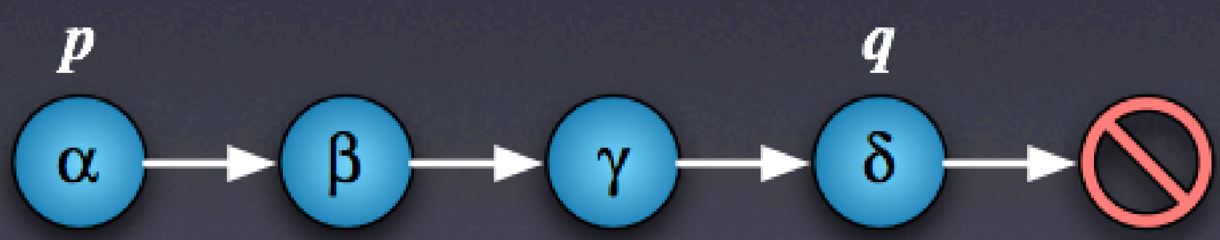


Is_Reachable_in(3, p, q)

Definitions

Definition Var Is_Reachable (p, q: Location): **B**

= ∃ k: **N** ∋ Is_Reachable_in(k, p, q);

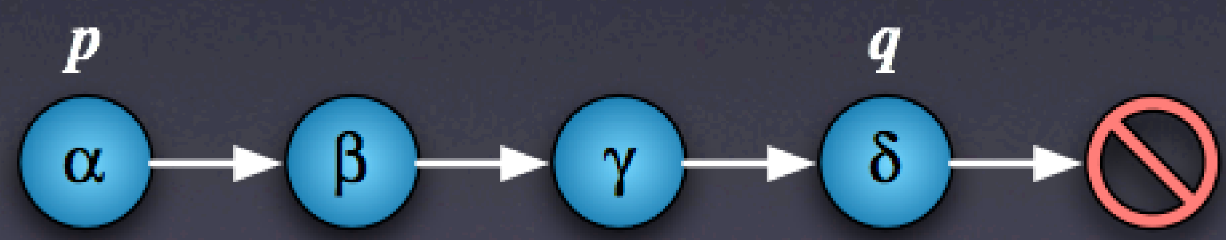


Is_Reachable(p, Void)

Definitions

Definition Var Distance(p, q: Location): **N**

$$= \begin{cases} k & \textbf{if } \text{Is\_Reachable\_in}(k, p, q) \\ 0 & \textbf{otherwise} \end{cases} ;$$



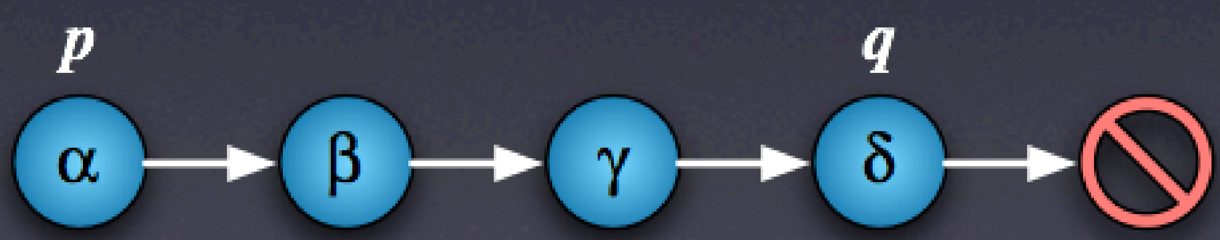
Distance(q, Void) = 1

Definition

Definition Var Is_Info_Str(p, q: Location, α: Str(Info): **B**

= $\exists$ n: **N** $\ni$ Is_Reachable_in(n, p, q) **and**

$\alpha = \prod_{k=1..n} \langle$ Contents(Target^k(p)) $\rangle$;



Is_Info_Str(p, q, $\langle \alpha, \beta, \gamma \rangle$)