# Relaxed Linear References for Lock-free Data Structures

**Elias Castegren**, Tobias Wrigstad

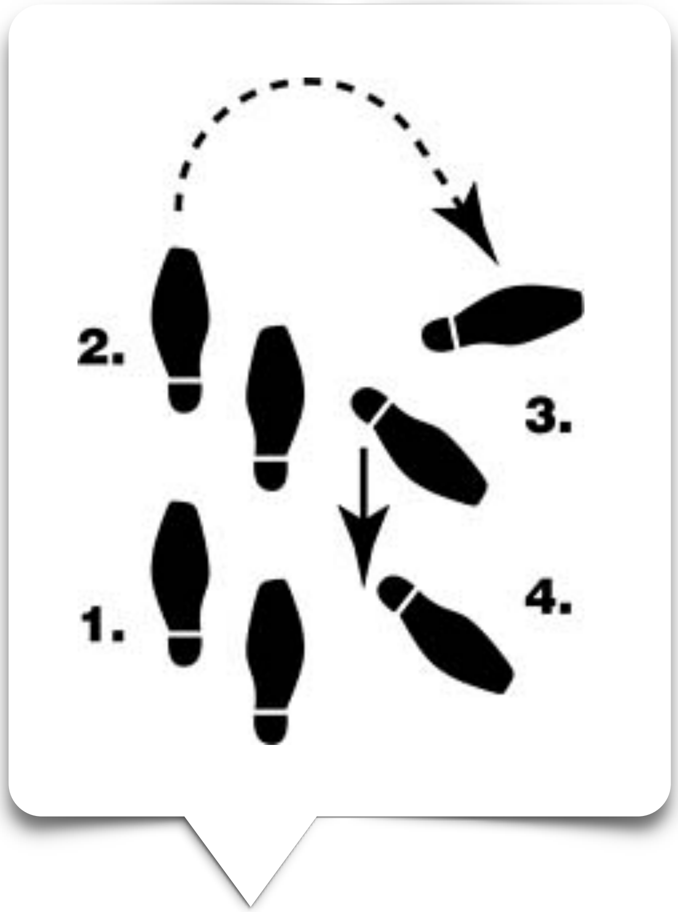
ECOOP'17, Barcelona

http://www.kristianstadik.se/www/wordpress/wp-content/uploads/2016/06/midsommar.jpg

http://www.vikstroms.nu/cms/wp-content/uploads/2014/06/Midsommar-2014.png

http://pix.tagapi.aller.se/sf.php?src=http://stilexperten.mabra.com/wp-content/uploads/sites/36/2016/06/383ab62d304910ab9e9bb9a0f197d918.jpg&w=640&h=400&a=c

*http://2.bp.blogspot.com/-ed71CELc5pY/T-h5pPLBR-I/AAAAAAAAAck/9hILp4JdDJg/s1600/annicasxperia+juni+026.jpg*




# ”Små grodorna är lustiga att se”

*https://upload.wikimedia.org/wikipedia/en/e/e7/The_Swedish_Chef.jpg*

*https://68.media.tumblr.com/5cab6229774bfb3fa6a35bf3479b6f8c/tumblr_inline_nht61yoHMA1r0y63m.jpg*

*https://s3.amazonaws.com/gs-geo-images/3738763b-2dc4-47fb-81d2-345908ddbc07_l.jpg*

# Lock-free Programming

- Fine-grained concurrency achieved by following some protocol

  1. Read a shared value (speculation)
  2. Perform some local computation
  3. If the speculation is still valid, make the local value global (publication)

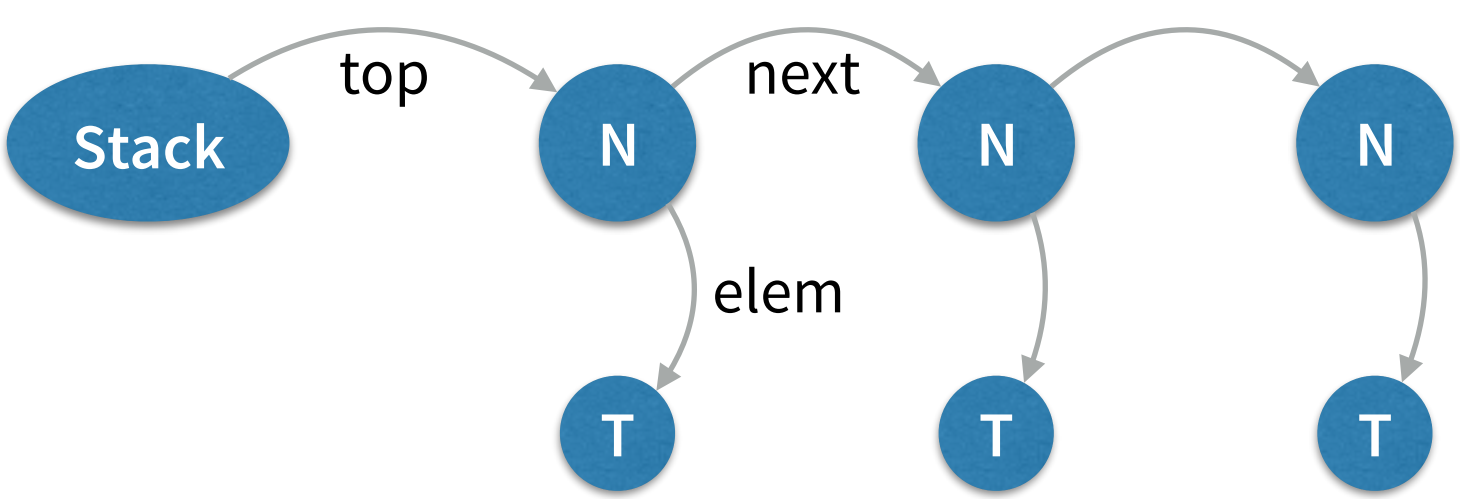
- Requires atomic operations, e.g. CAS (compare-and-swap)

```
def CAS(x.f, y, z) : bool {
  if (x.f == y) {      } Performed atomically
    x.f = z;
    return true;
  } else
    return false;
}
```

# Example: A Lock-free Stack [Treiber]

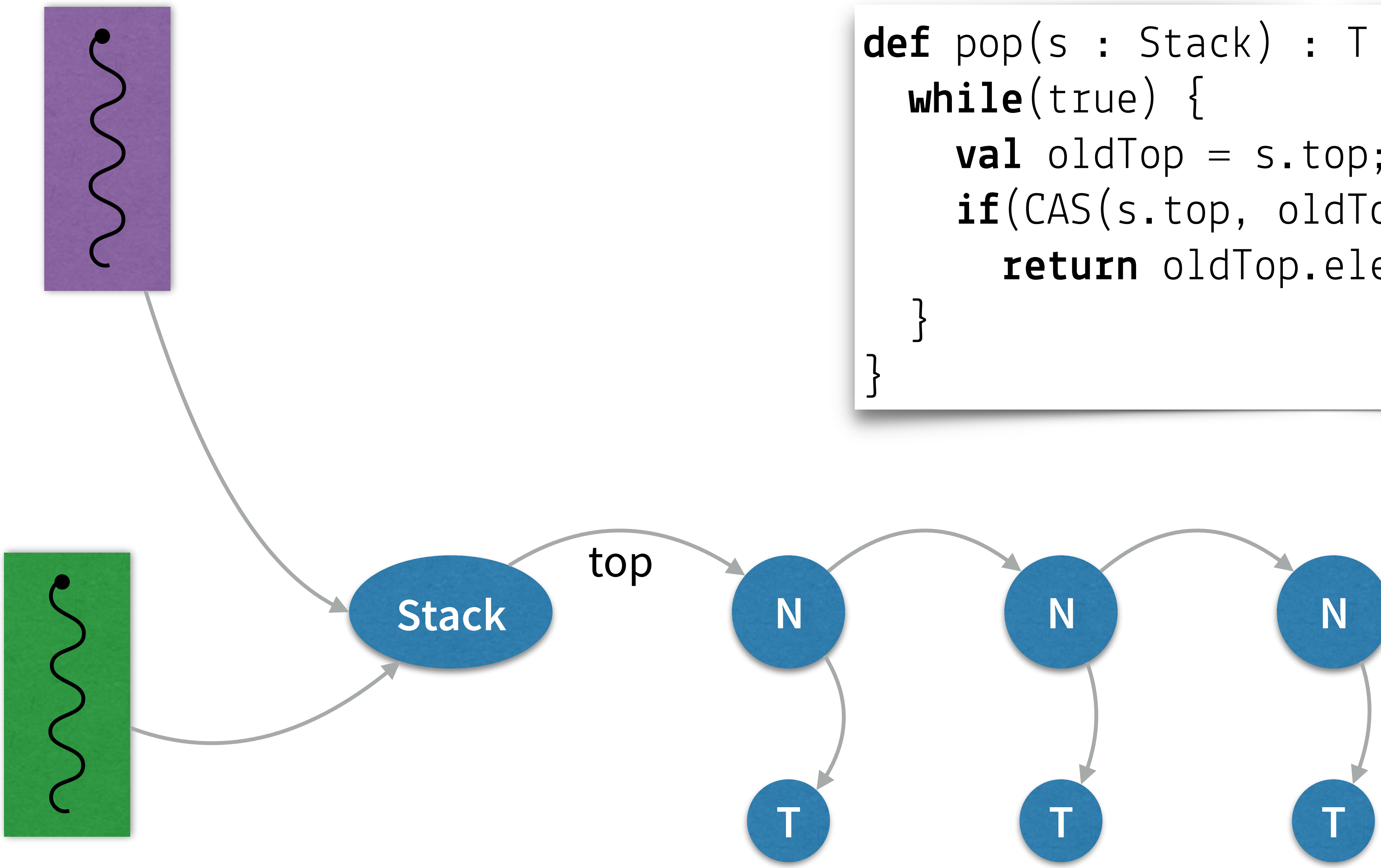
```
struct Stack {
  var top : Node;
}
```

```
struct Node {
  var elem : T;
  val next : Node;
}
```



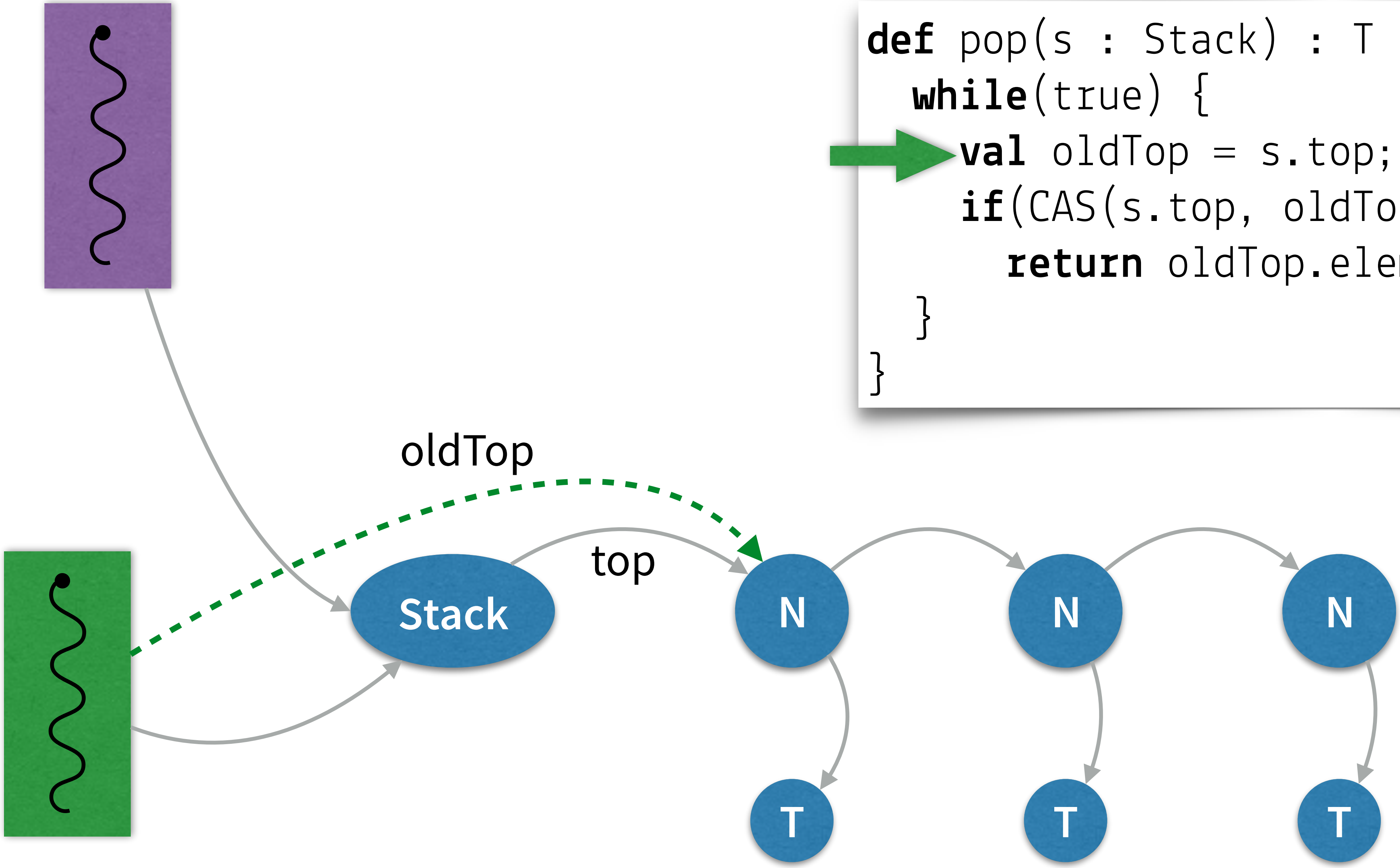
# Example: A Lock-free Stack [Treiber]

```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAS(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```

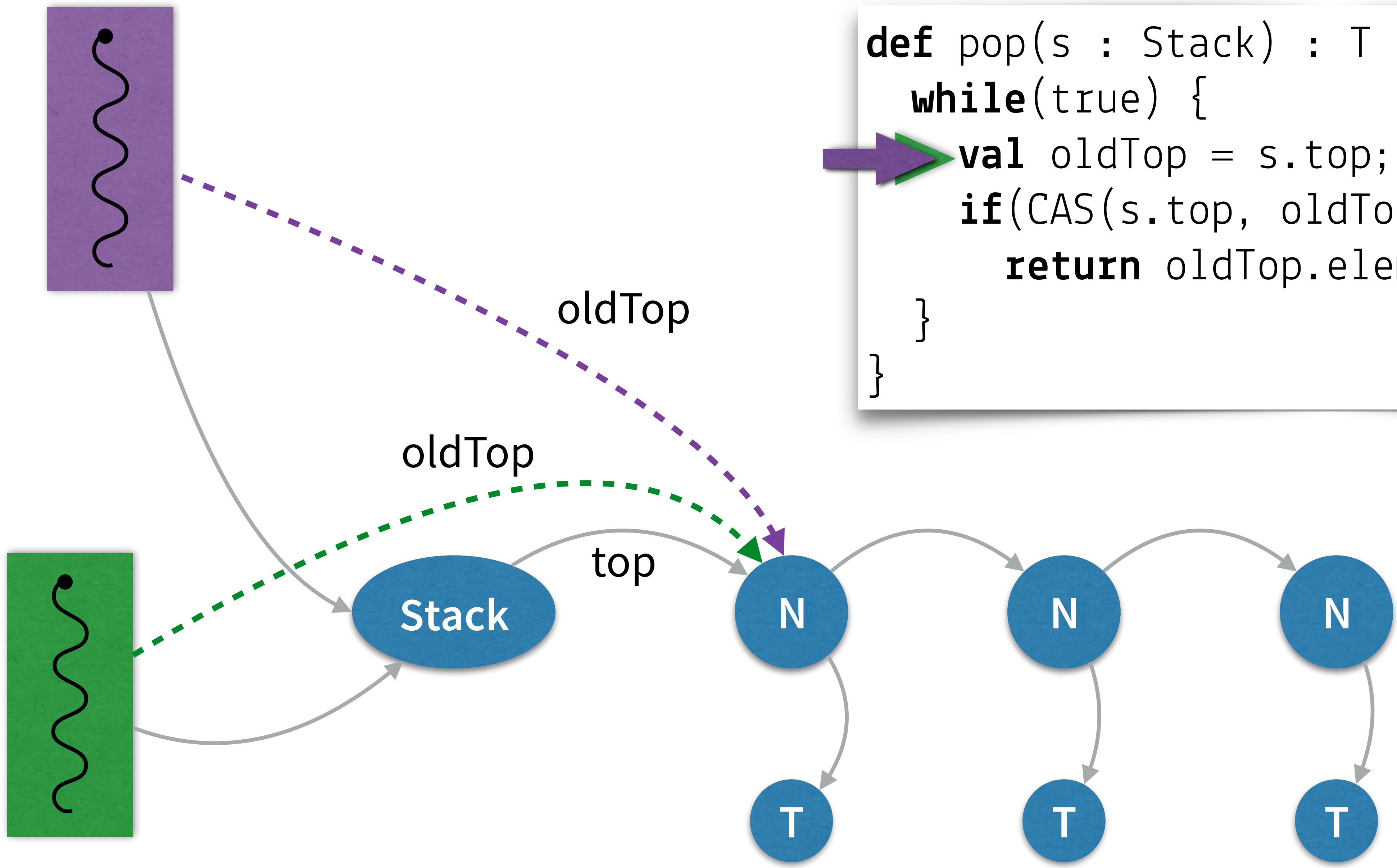


# Example: A Lock-free Stack [Treiber]

```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAS(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```

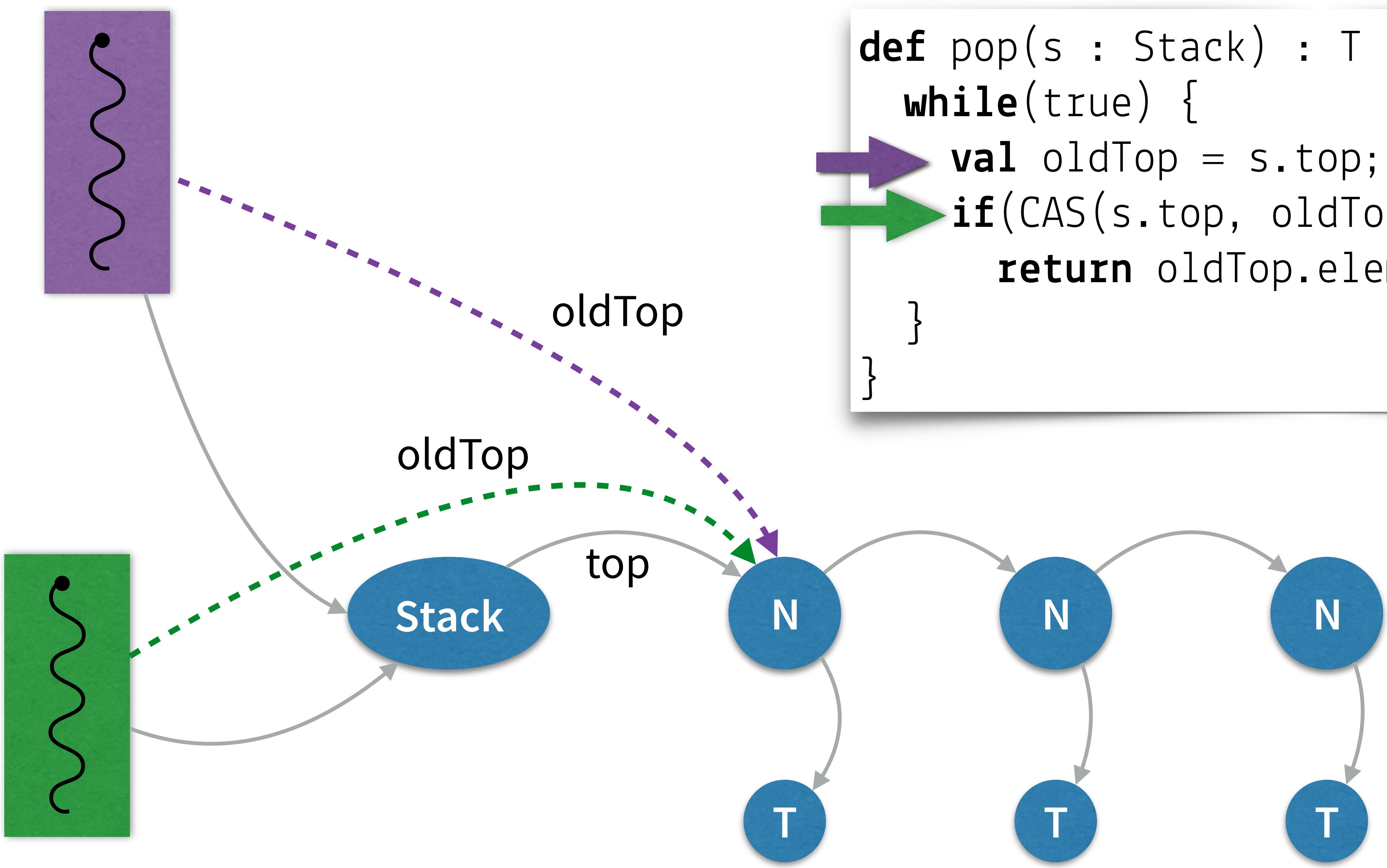


# Example: A Lock-free Stack [Treiber]




```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAS(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```

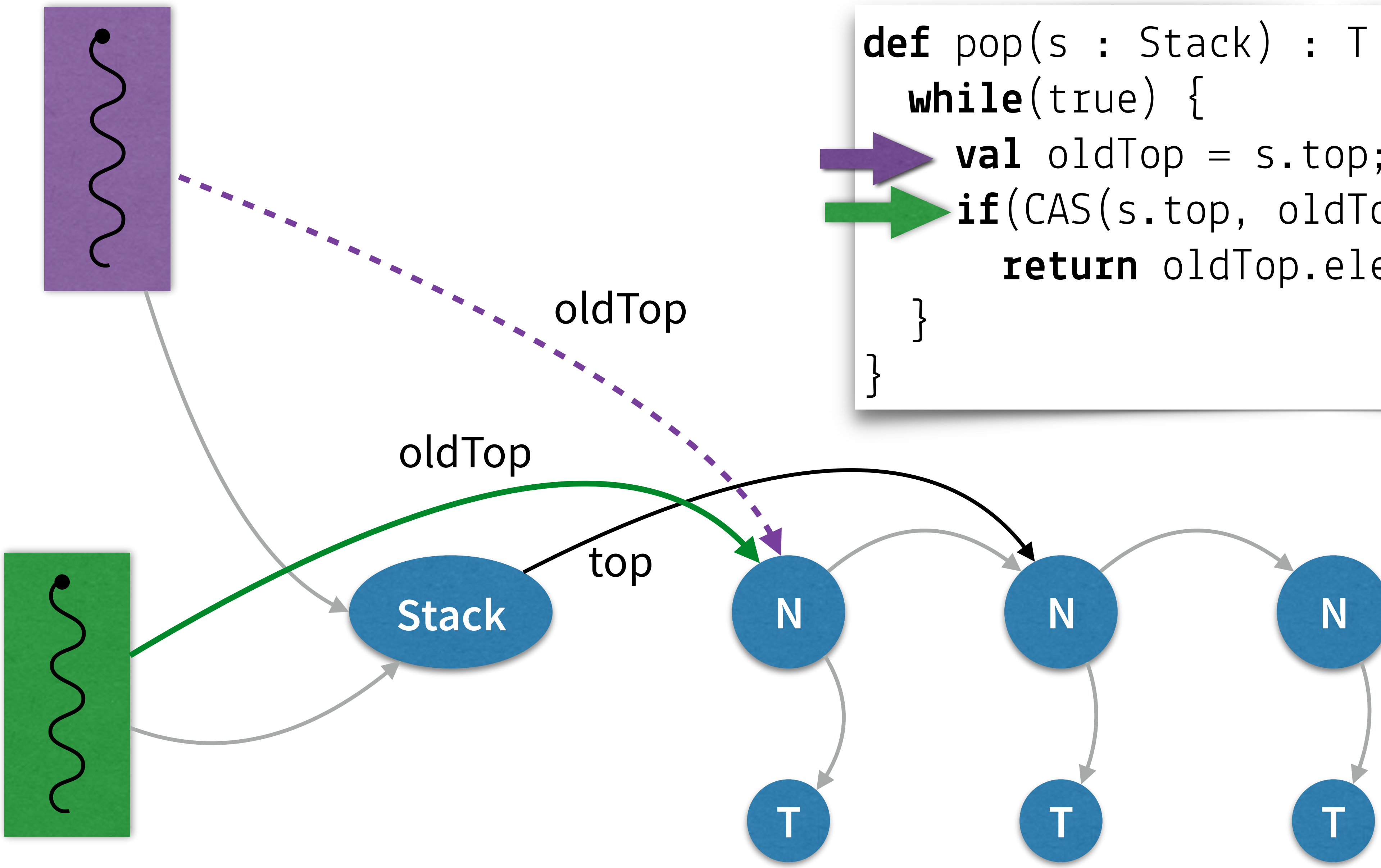
# Example: A Lock-free Stack [Treiber]



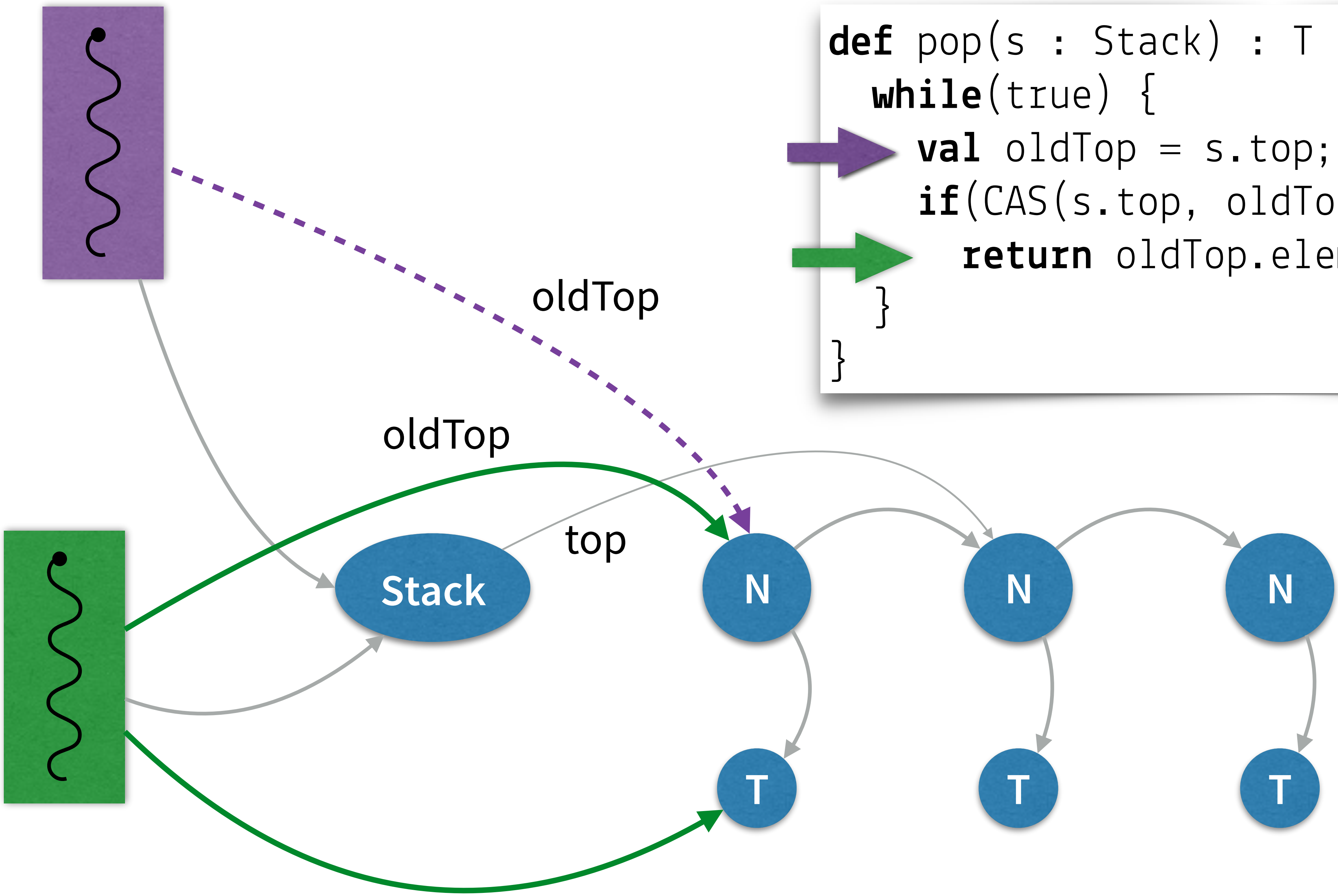

```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAS(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```

# Example: A Lock-free Stack [Treiber]

```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAS(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```

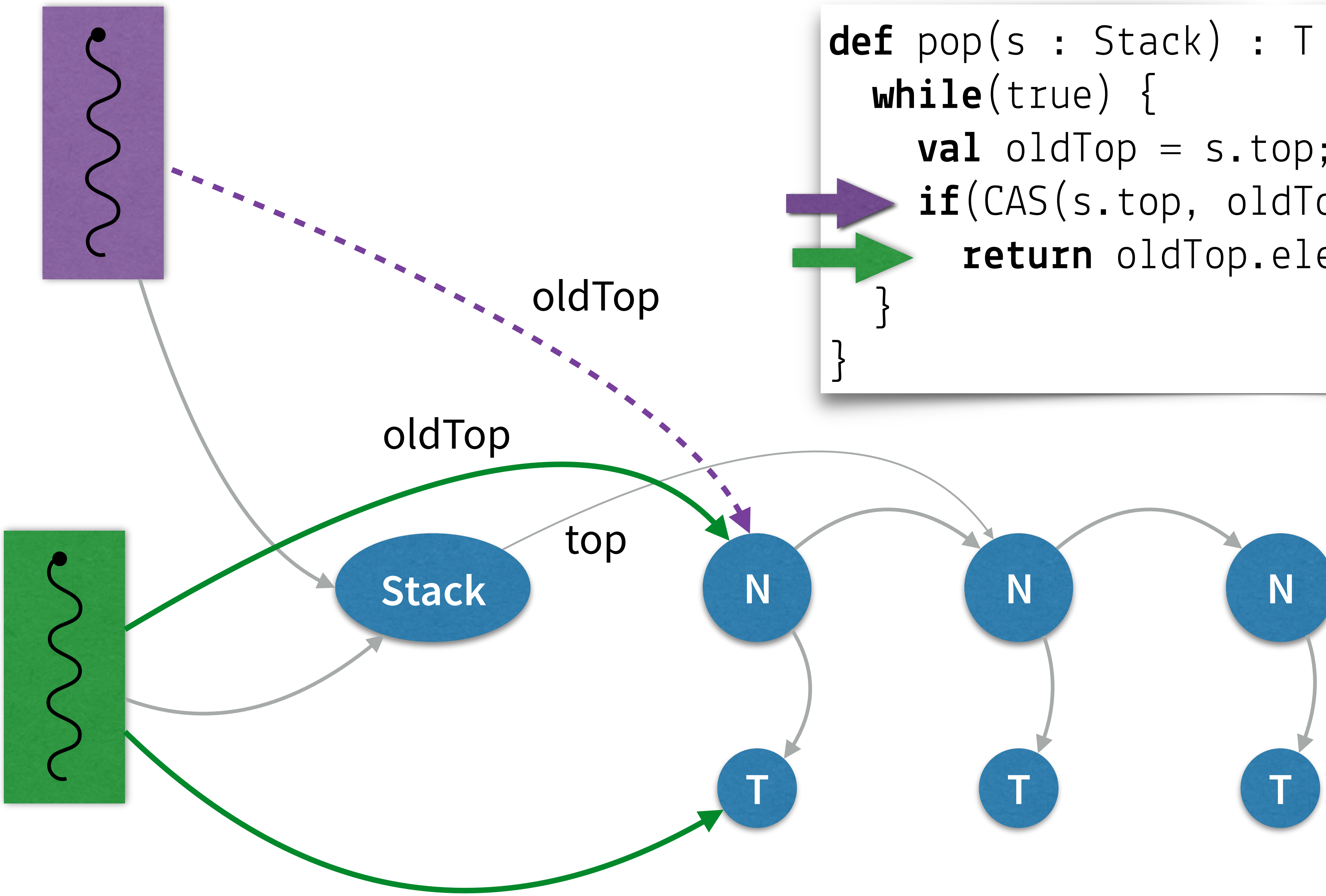


# Example: A Lock-free Stack [Treiber]



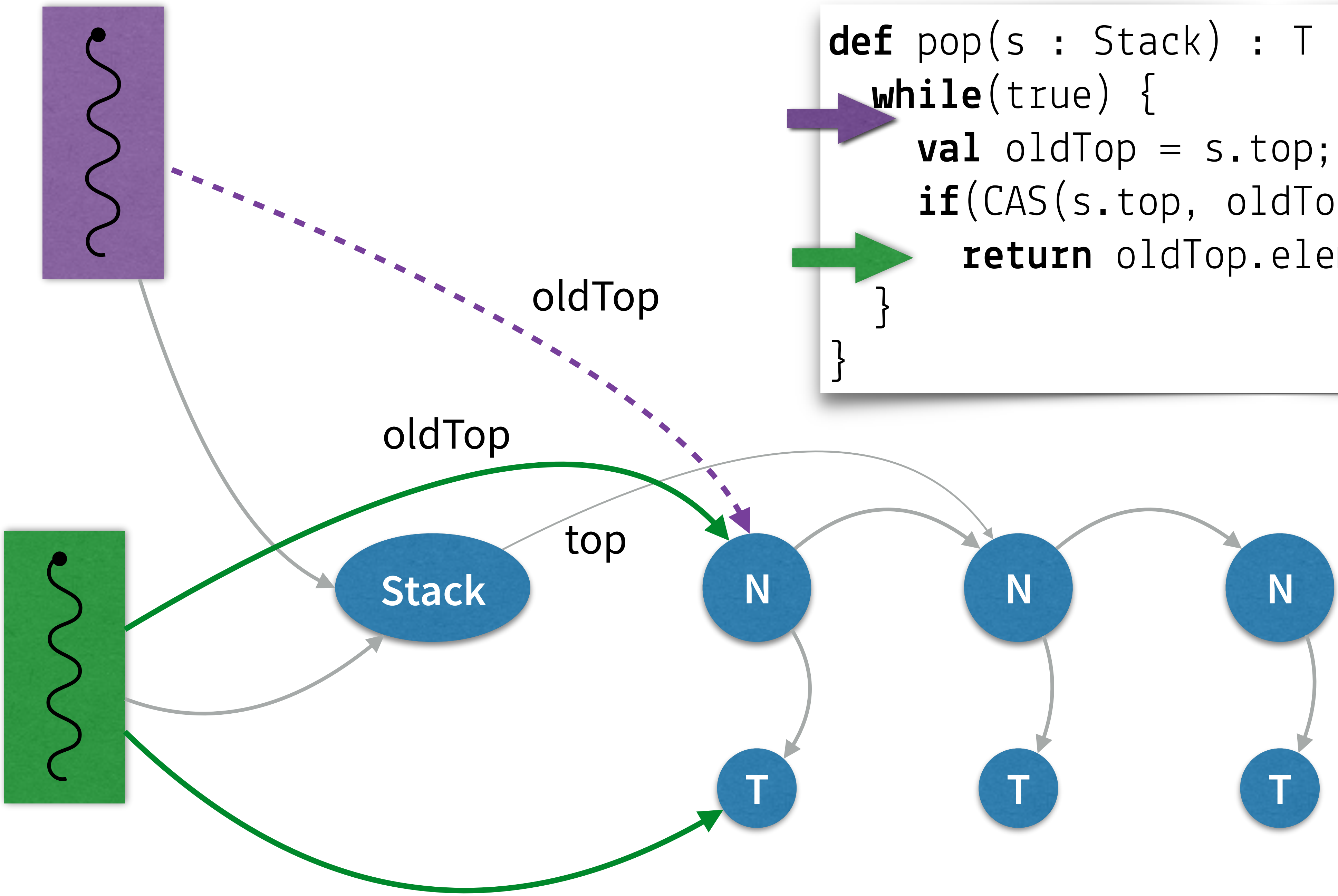

```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAS(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```

# Example: A Lock-free Stack [Treiber]




```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAS(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```

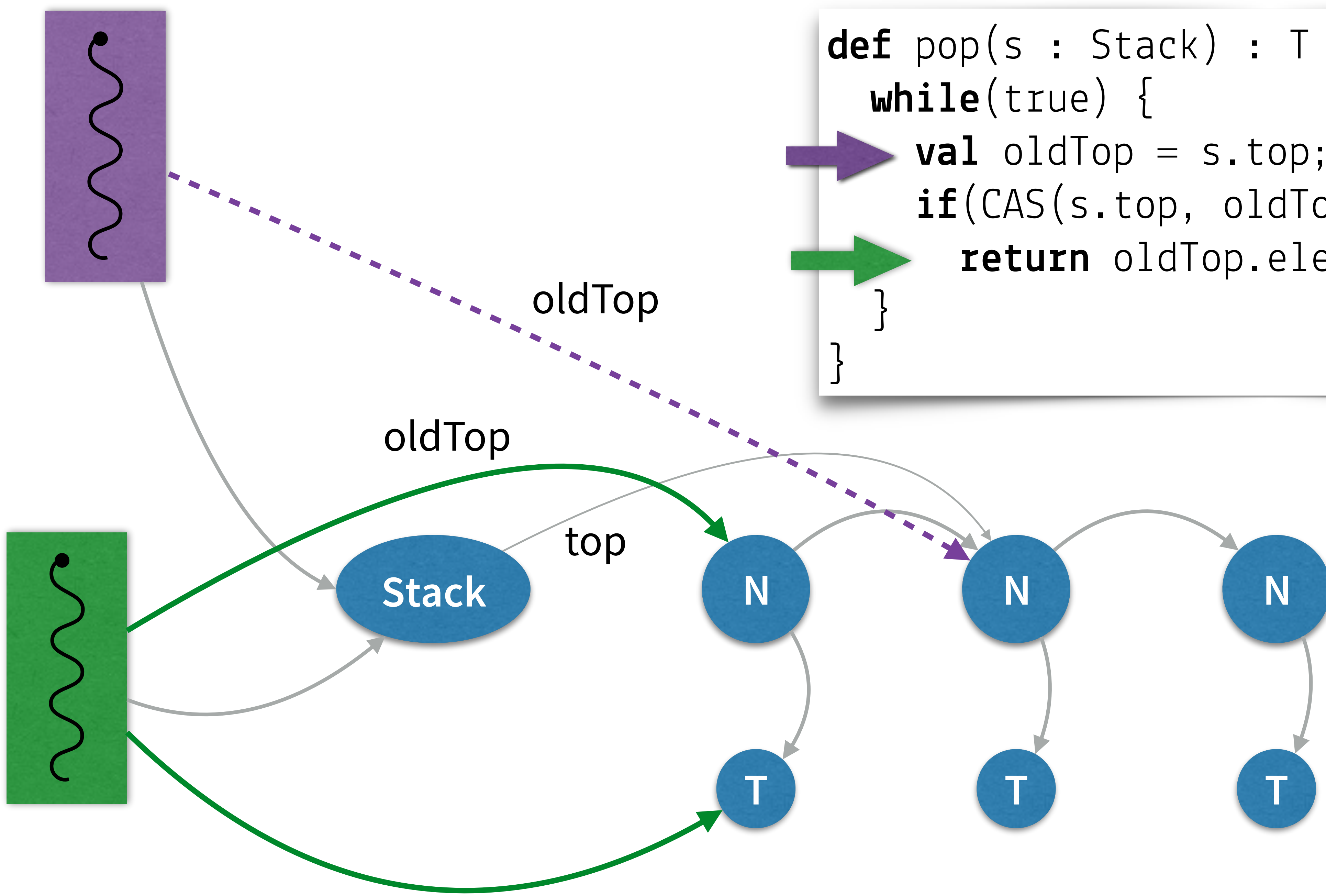
# Example: A Lock-free Stack [Treiber]




```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAS(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```

# Example: A Lock-free Stack [Treiber]

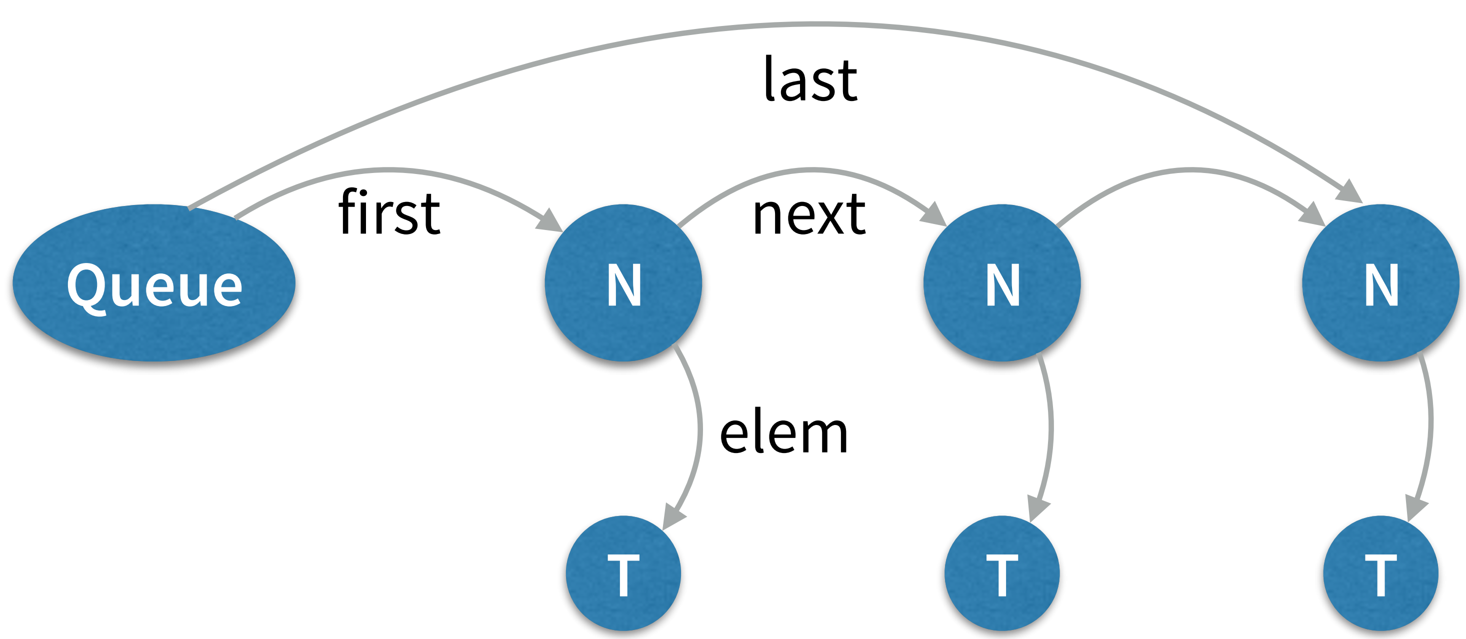
```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAS(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```



# Example: A Lock-free Queue [Michael & Scott]

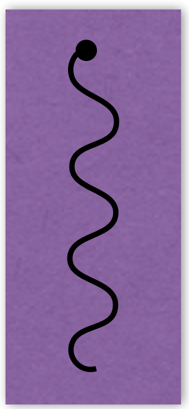
```
struct Queue {
  var first : Node;
  var last : Node
}
```

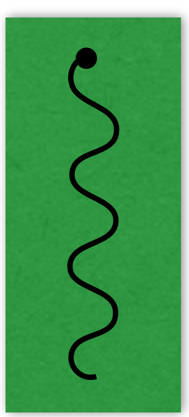
```
struct Node {
  var elem : T;
  var next : Node;
}
```

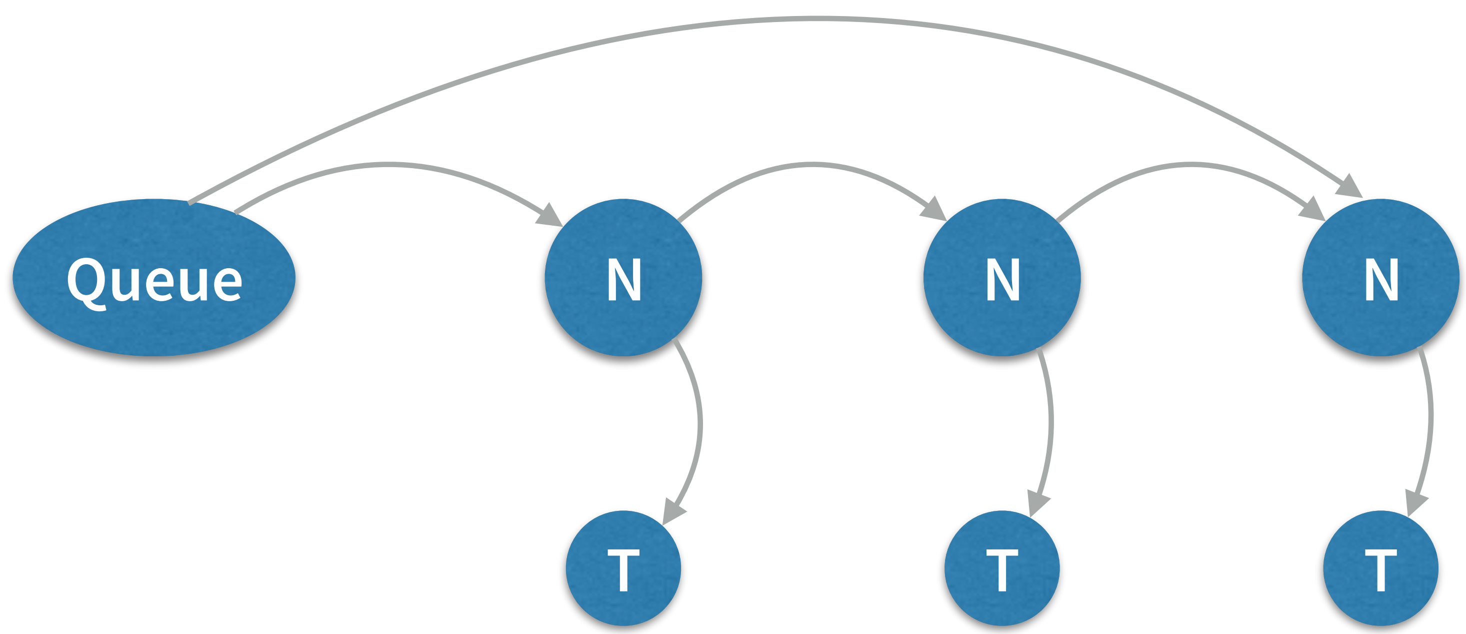


# Example: A Lock-free Queue [Michael & Scott]

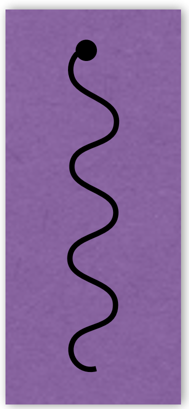
```
def enqueue(q : Queue, x : Elem) : void {
  val n = new Node(x);
  val done = false;
  while(not done) {
    val oldLast = q.last;
    done = CAS(oldLast.next, null, n);
    q.last = oldLast.next;
  }
}
```



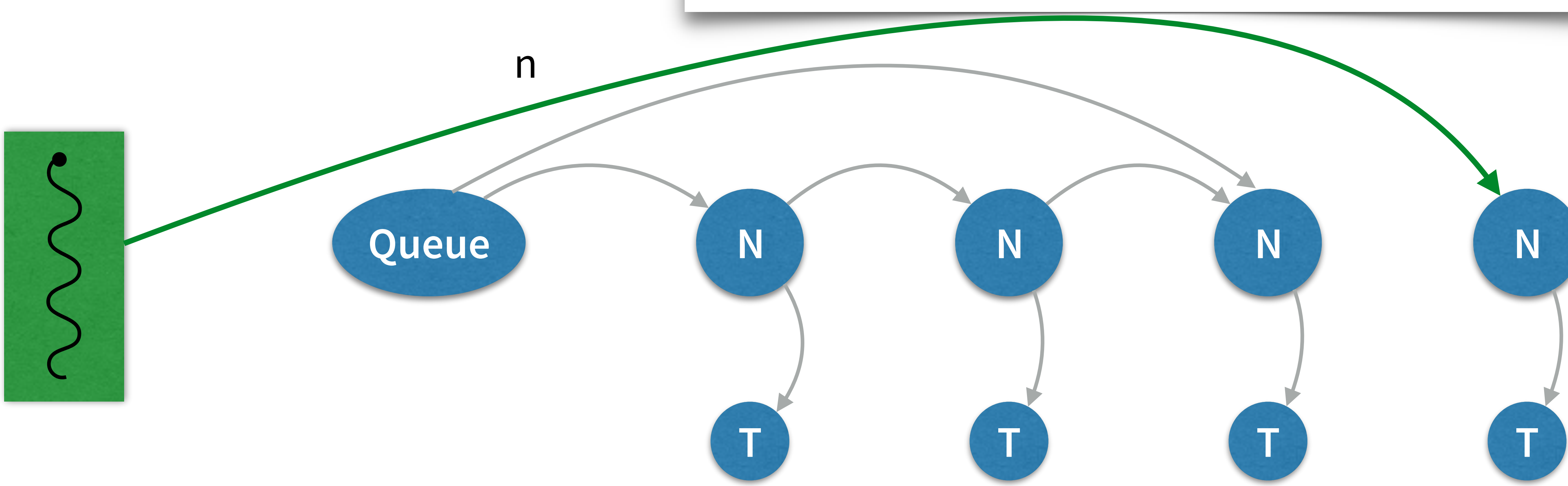




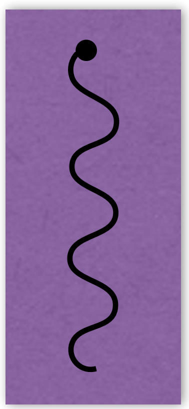
# Example: A Lock-free Queue [Michael & Scott]



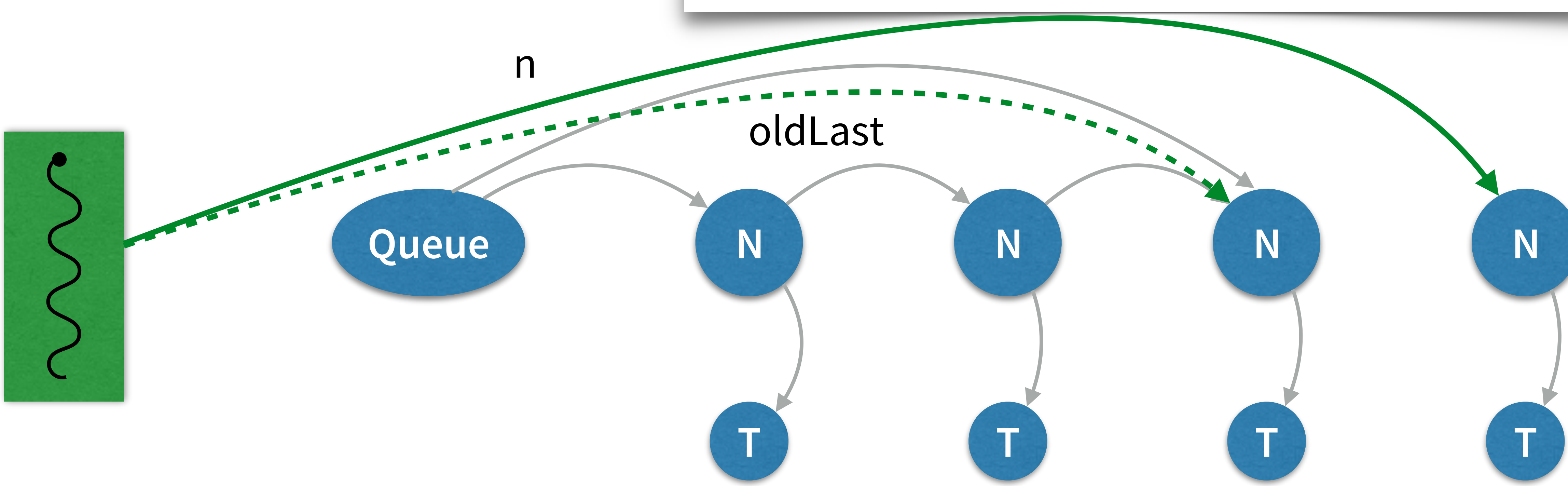
```
def enqueue(q : Queue, x : Elem) : void {
  val n = new Node(x);
  val done = false;
  while(not done) {
    val oldLast = q.last;
    done = CAS(oldLast.next, null, n);
    q.last = oldLast.next;
  }
}
```



# Example: A Lock-free Queue [Michael & Scott]

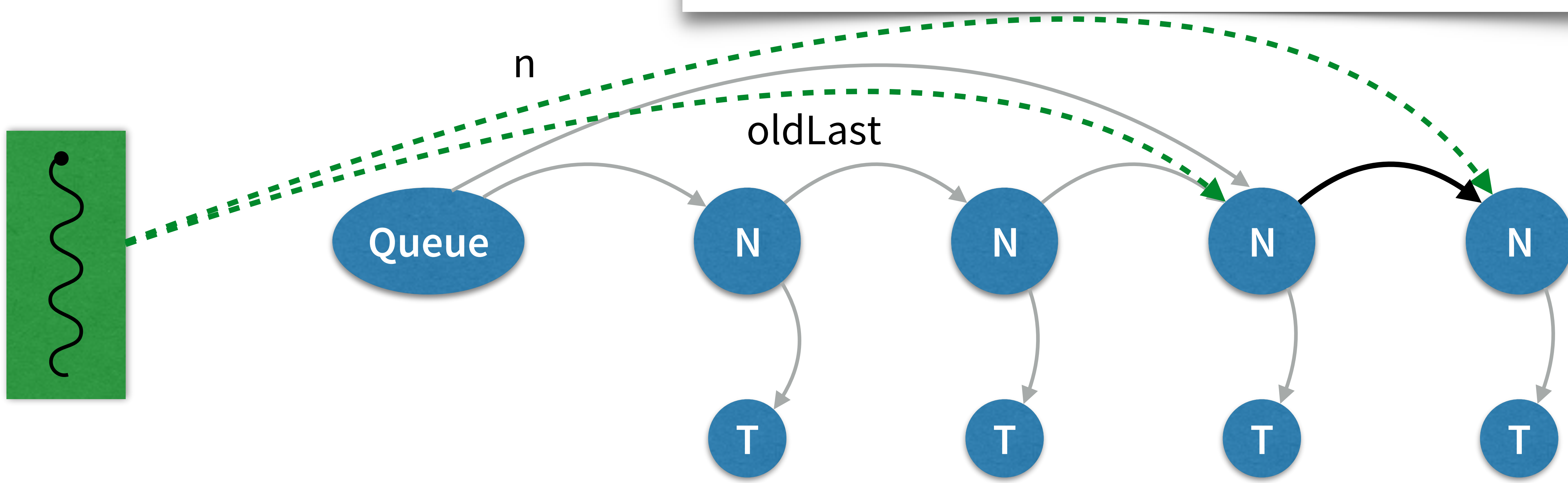


```
def enqueue(q : Queue, x : Elem) : void {
  val n = new Node(x);
  val done = false;
  while(not done) {
    val oldLast = q.last;
    done = CAS(oldLast.next, null, n);
    q.last = oldLast.next;
  }
}
```



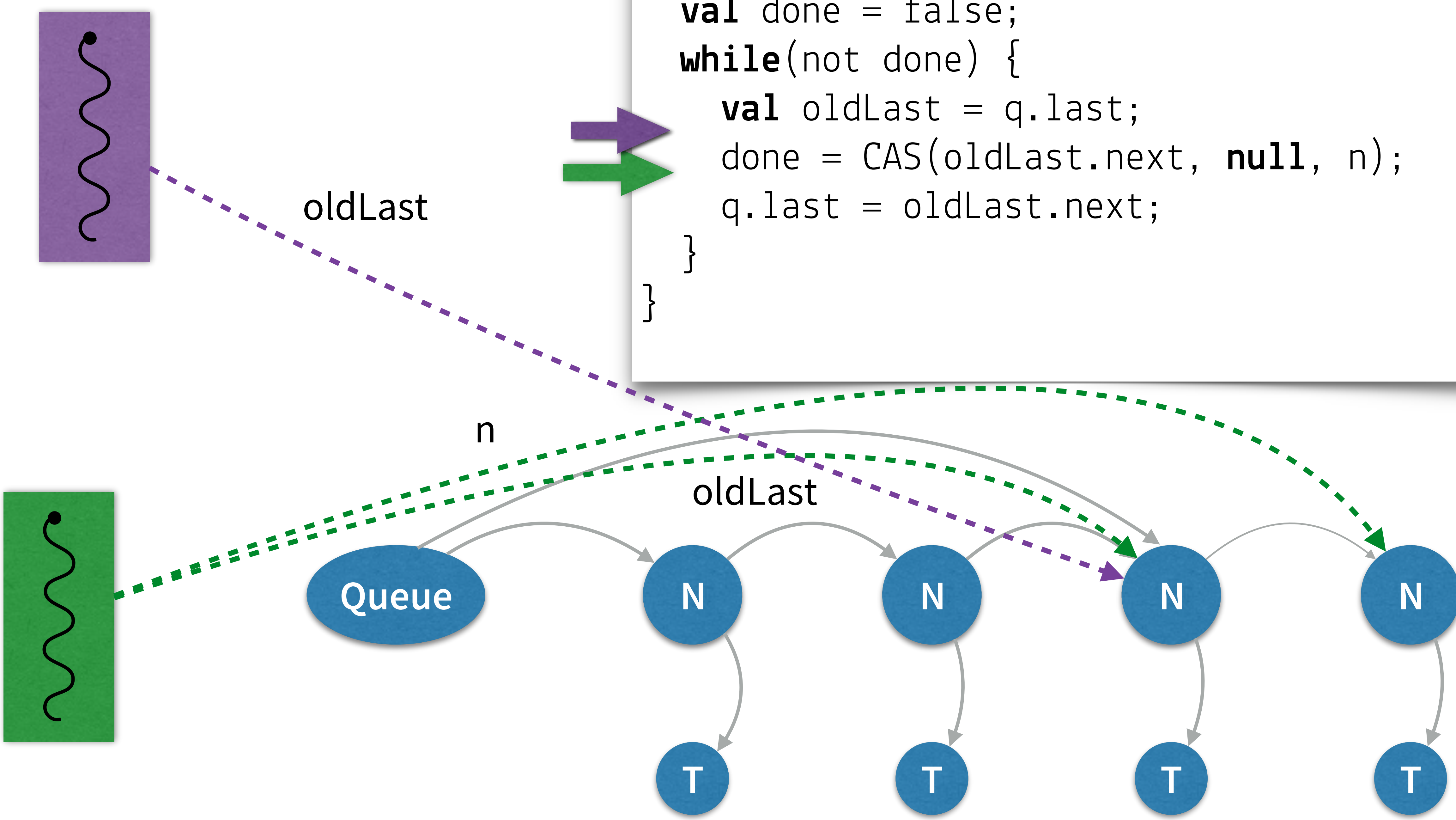
# Example: A Lock-free Queue [Michael & Scott]

```
def enqueue(q : Queue, x : Elem) : void {
  val n = new Node(x);
  val done = false;
  while(not done) {
    val oldLast = q.last;
    done = CAS(oldLast.next, null, n);
    q.last = oldLast.next;
  }
}
```



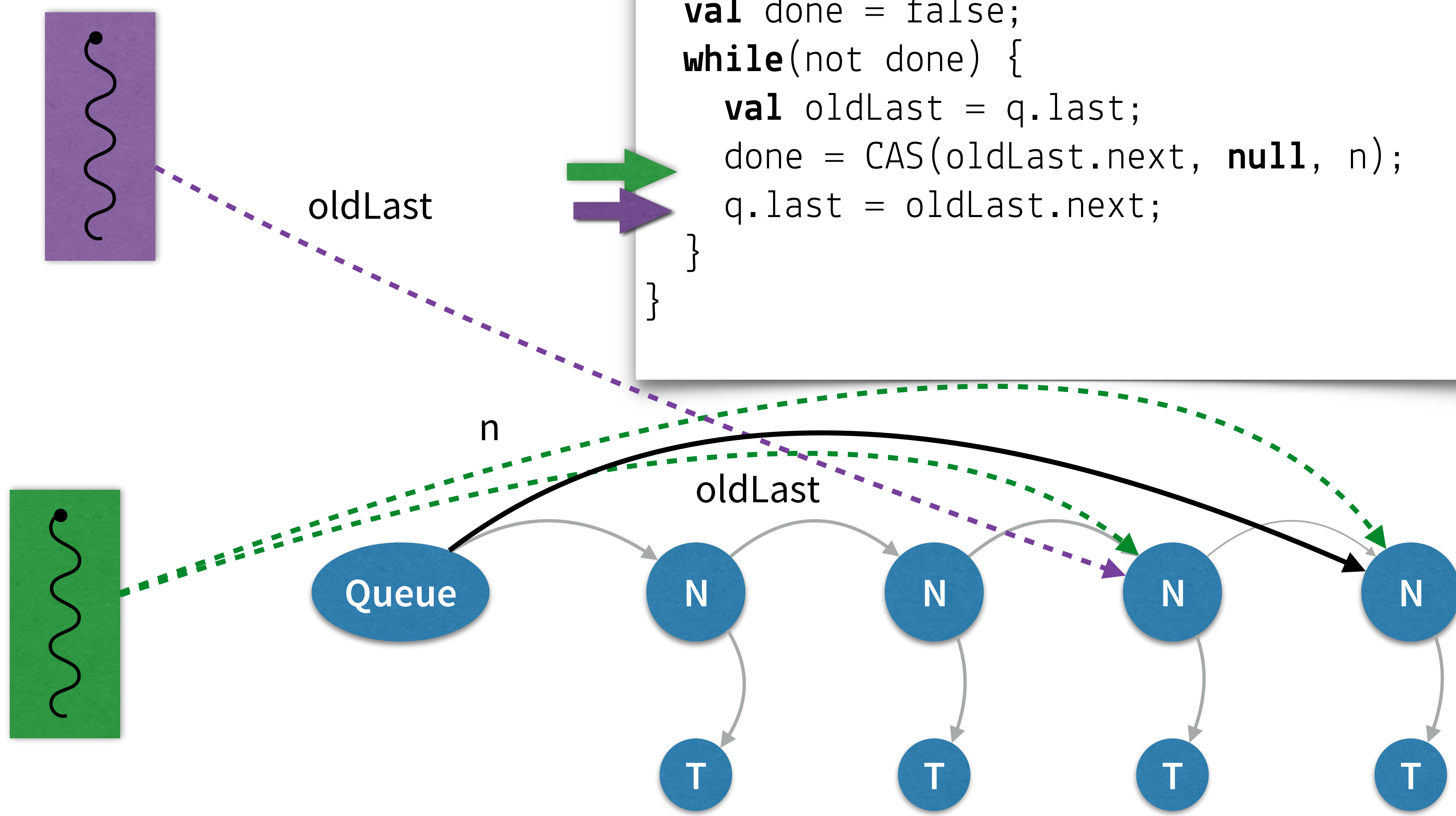
# Example: A Lock-free Queue [Michael & Scott]

```
def enqueue(q : Queue, x : Elem) : void {
  val n = new Node(x);
  val done = false;
  while(not done) {
    val oldLast = q.last;
    done = CAS(oldLast.next, null, n);
    q.last = oldLast.next;
  }
}
```



# Example: A Lock-free Queue [Michael & Scott]

```
def enqueue(q : Queue, x : Elem) : void {
  val n = new Node(x);
  val done = false;
  while(not done) {
    val oldLast = q.last;
    done = CAS(oldLast.next, null, n);
    q.last = oldLast.next;
  }
}
```



# Finding Patterns in Lock-free Programming

- Speculation
- Publication
- Acquisition

```
def enqueue(q : Queue, x : Elem) : void {
  val n = new Node(x);
  val done = false;
  while(not done) {
    val oldLast = q.last;
    done = CAS(oldLast.next, null, n);
    q.last = oldLast.next;
  }
}
```

```
def dequeue(q : Queue) : T {
  ...
}
```

```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAS(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```

```
def push(s : Stack, x : T) : void {
  ...
}
```

# Finding Patterns in Lock-free Programming

- Speculation
- Publication
- Acquisition

```
def enqueue(q : Queue, x : Elem) : void {
  val n = new Node(x);
  val done = false;
  while(not done) {
    val oldLast = q.last;
    done = CAS(oldLast.next, null, n);
    q.last = oldLast.next;
  }
}
```

**Field 'last' is subject to races and must be written to with a CAS.**

```
def dequeue(q : Queue) : T {
  ...
}
```

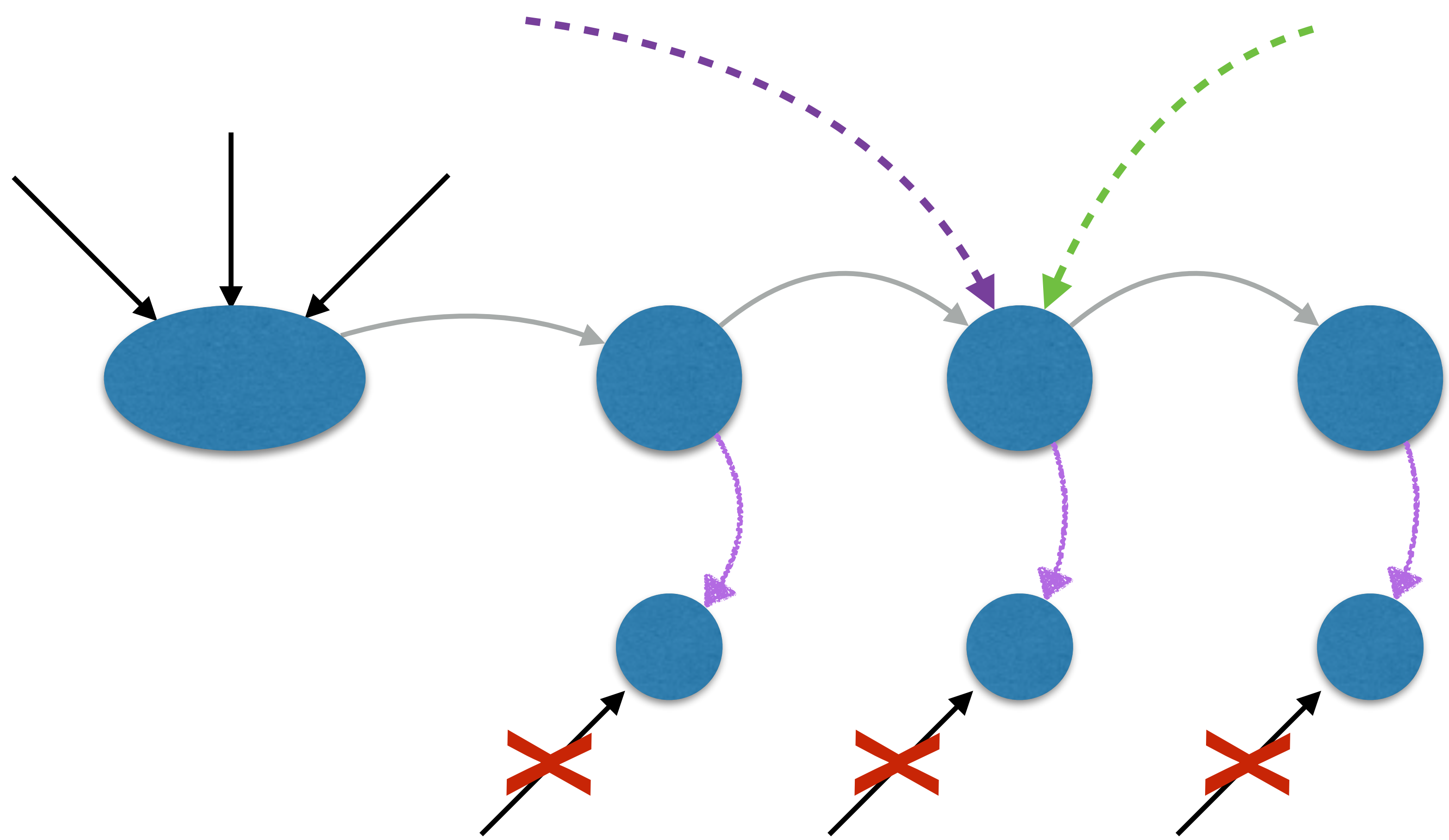
```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAS(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```

```
def push(s : S                  : void {
  ...
}
```

Our idea:
Give these concepts meaningful types!

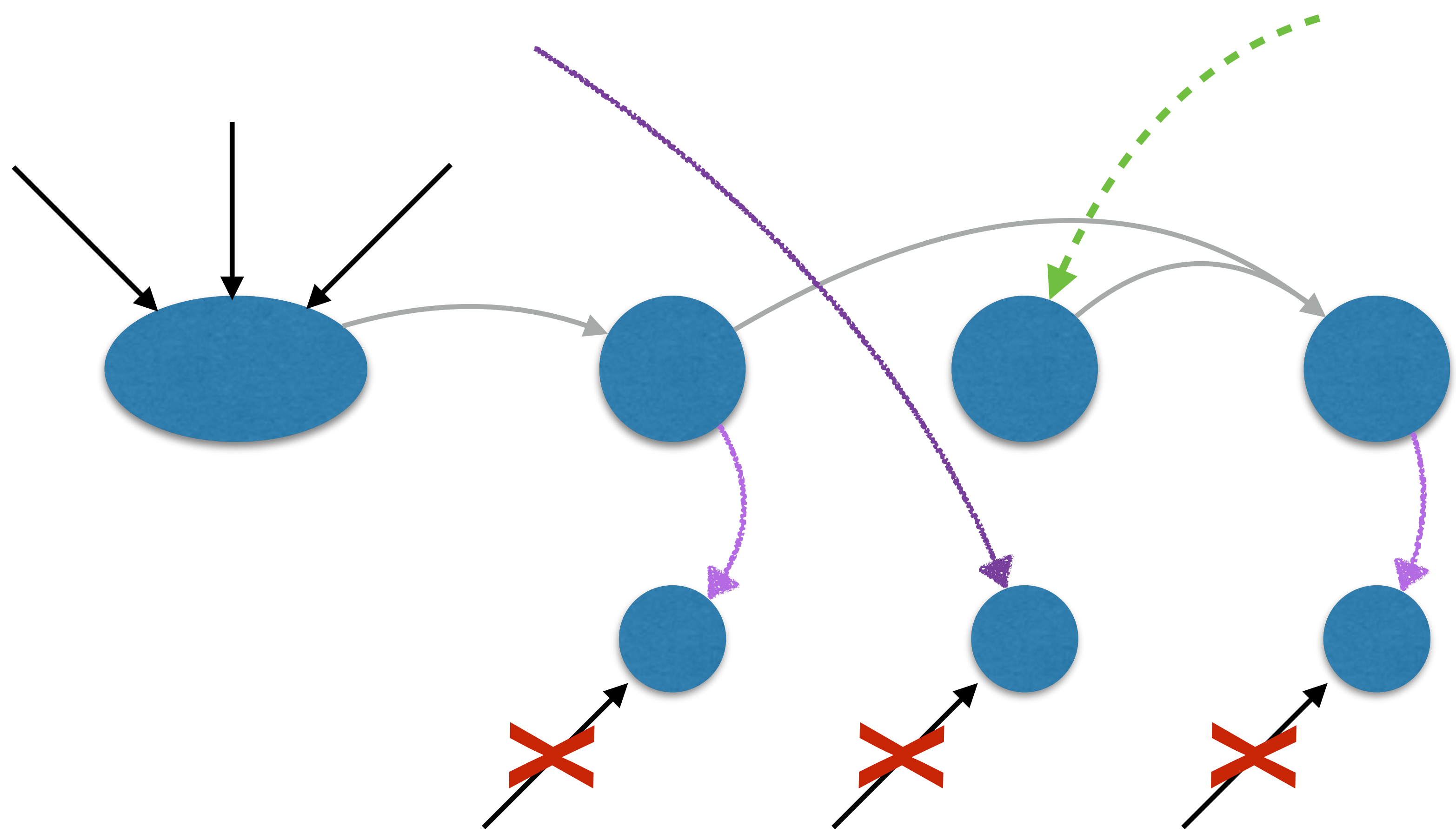
# Why Types?

- Why not model checking, concurrent separation logic, etc.?
- Modularity: no need for inter-procedural analysis
- Language expressivity: allow concurrent data structures storing linear references



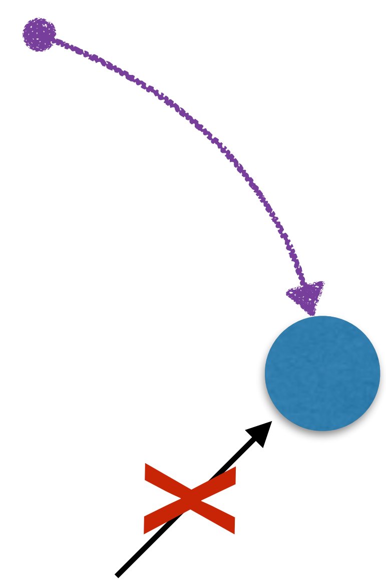
# Why Types?

- Why not model checking, concurrent separation logic, etc.?
- Modularity: no need for inter-procedural analysis
- Language expressivity: allow concurrent data structures storing linear references



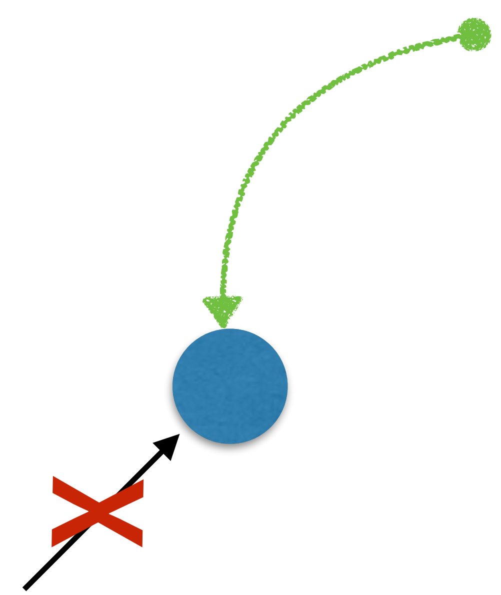
# Linear References for Ownership Transfer

- A linear reference is the only reference to an object
- Prohibits shared access to objects
- Makes ownership explicit



# Linear References for Ownership Transfer

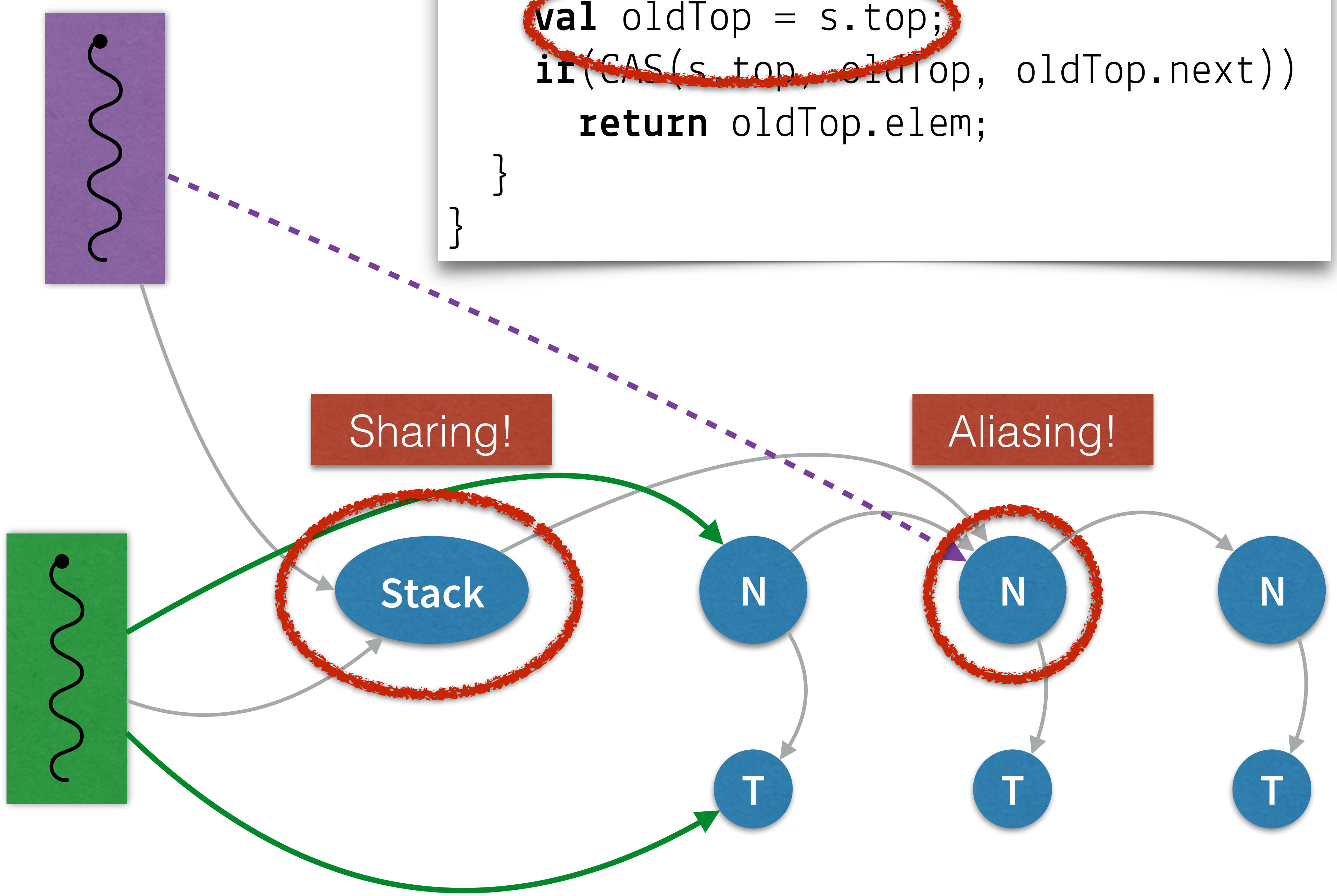
- A linear reference is the only reference to an object
- Prohibits shared access to objects
- Makes ownership explicit



- A correct lock-free data-structure can guarantee exclusive ownership of an element
  - Can we use linear references to capture this?

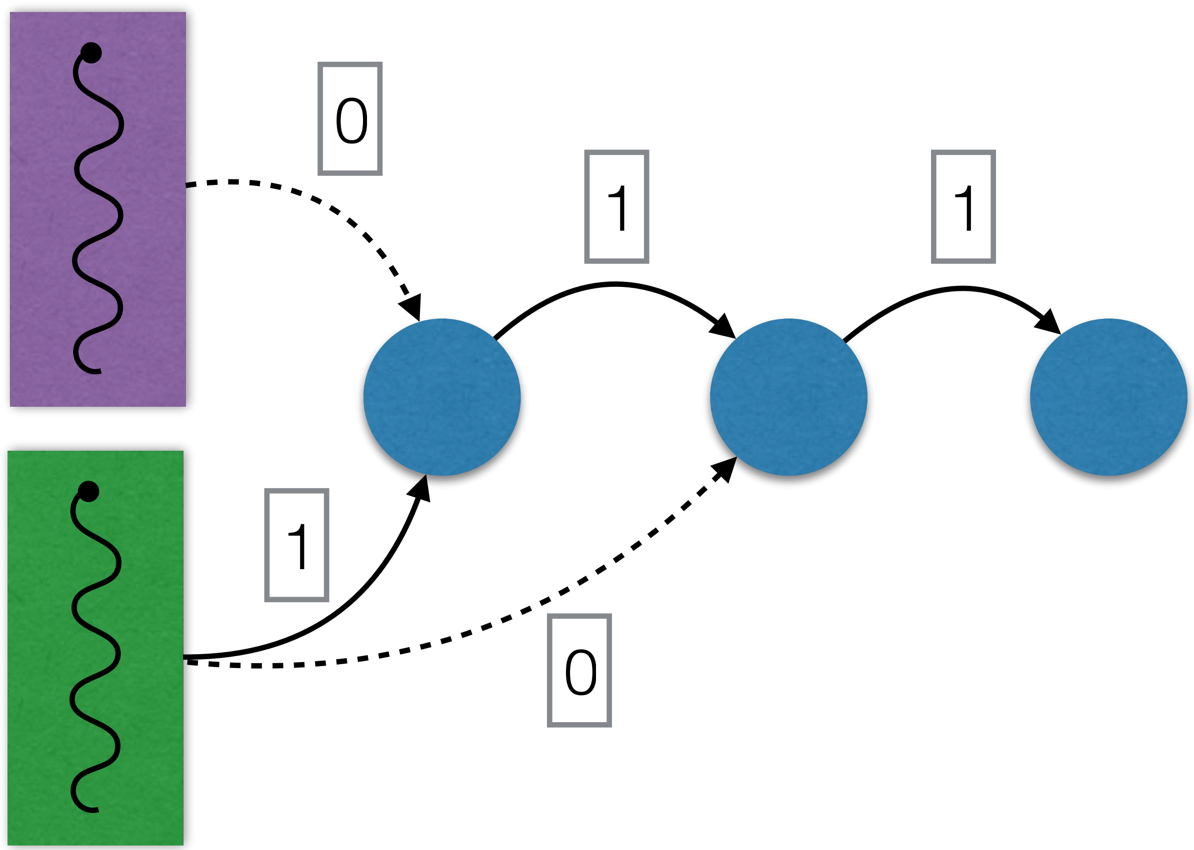
# Linear References and Lock-free Programming

```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAS(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```



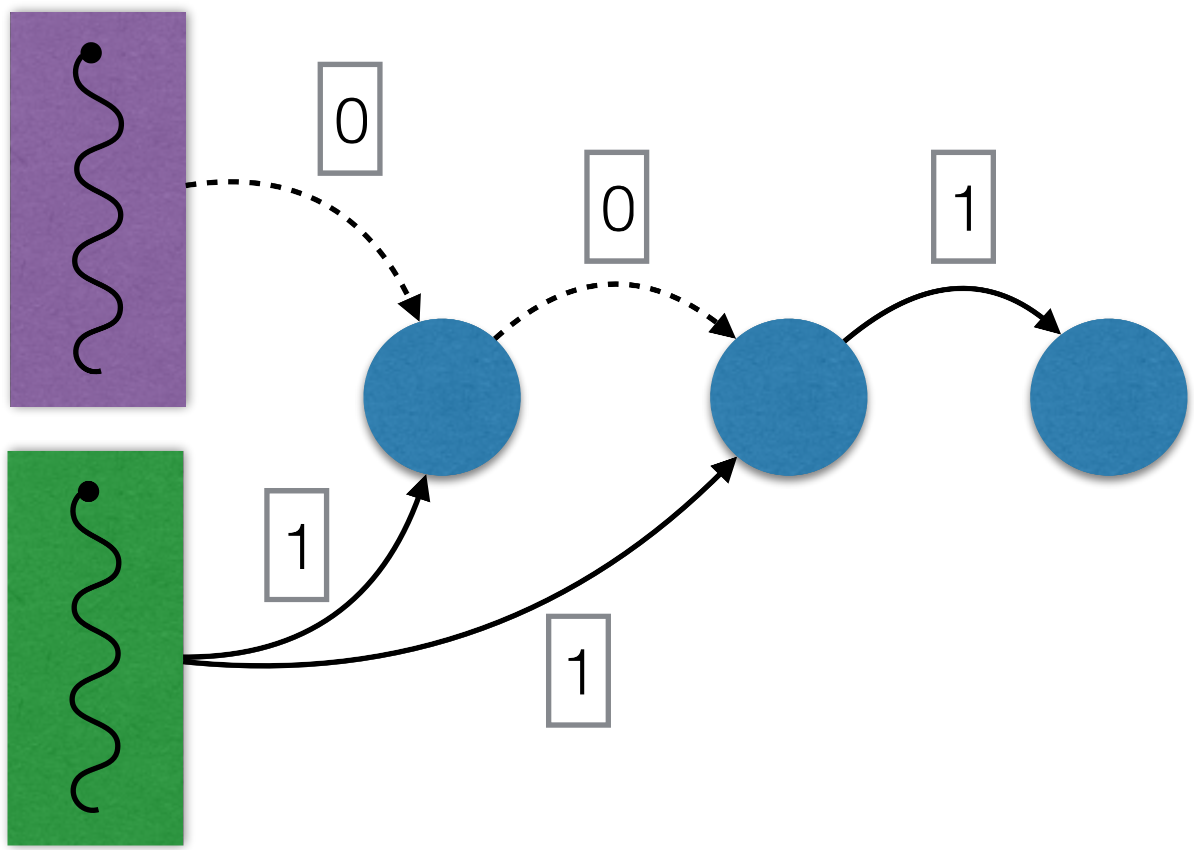
# Relaxed Linear References

- Separate aliasing from ownership
  - Unbounded aliasing, linear ownership
  - Ownership can be *atomically* transferred between aliases



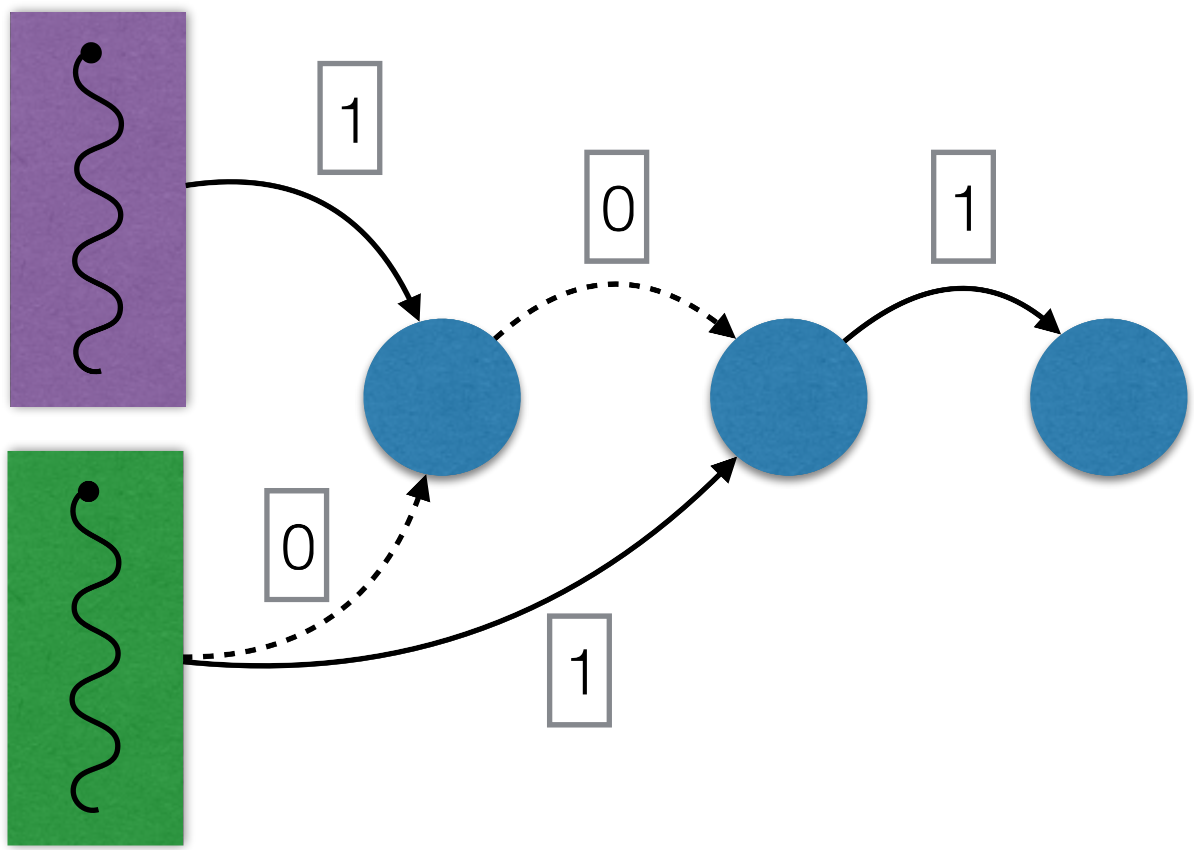
# Relaxed Linear References

- Separate aliasing from ownership
  - Unbounded aliasing, linear ownership
  - Ownership can be *atomically* transferred between aliases



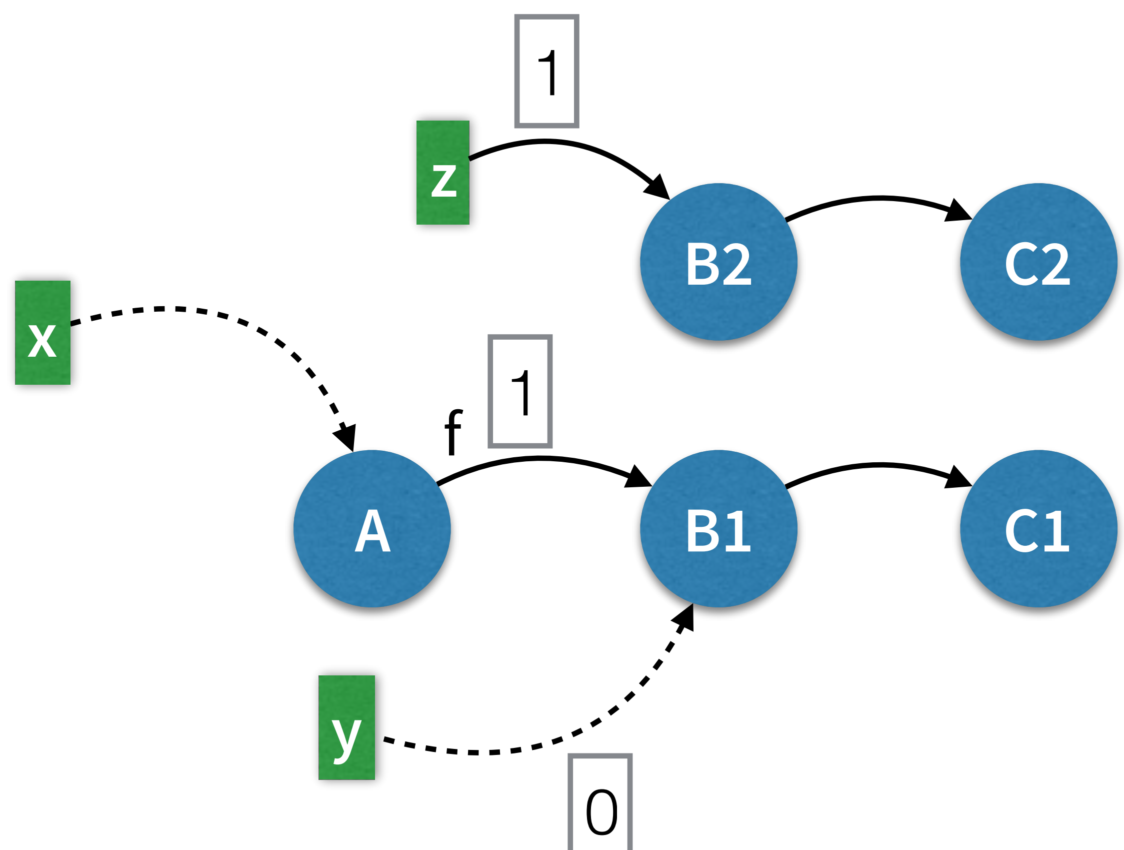
# Relaxed Linear References

- Separate aliasing from ownership
  - Unbounded aliasing, linear ownership
  - Ownership can be *atomically* transferred between aliases



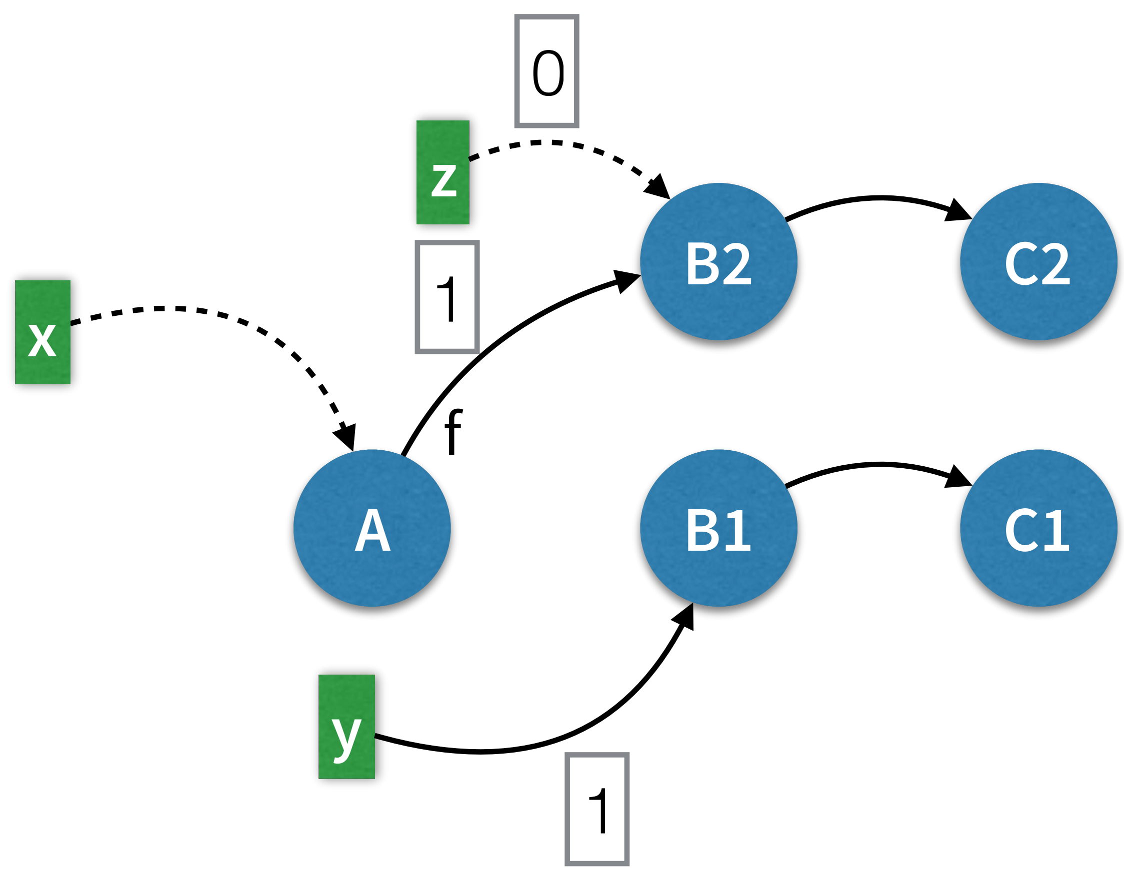
# Atomic Ownership Transfer with ~~CAS~~ CAT

`CAT(x.f, y, z)` — "Transfer ownership of `x.f` to `y` and of `z` to `x.f`"



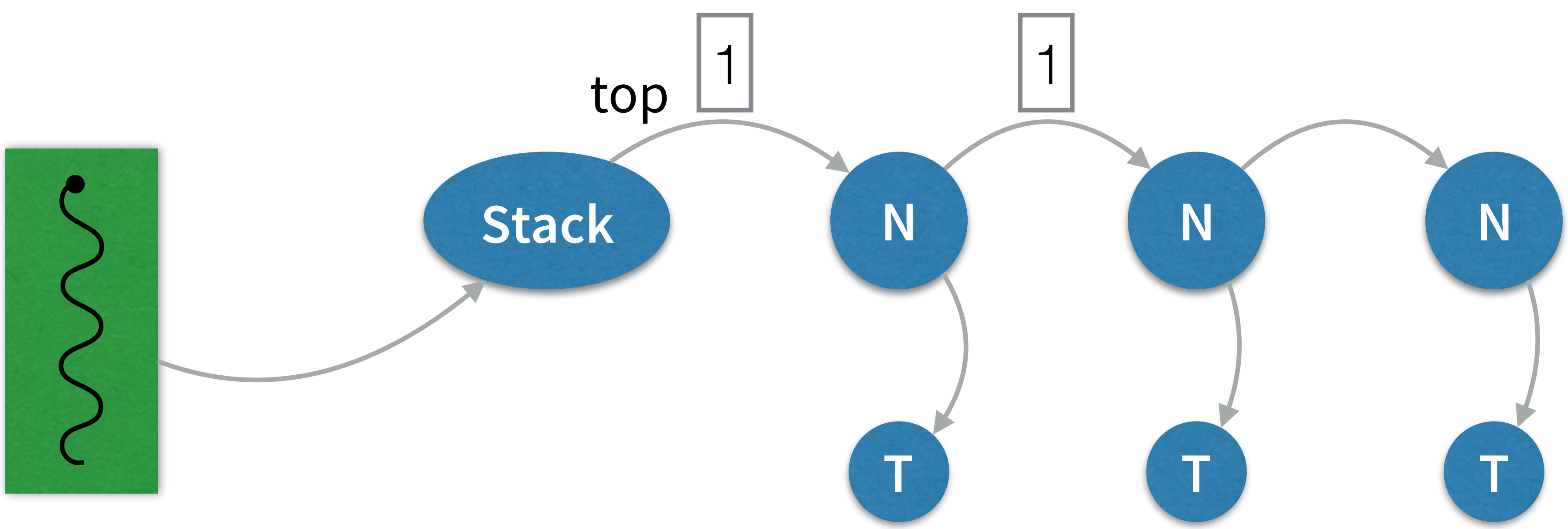
# Atomic Ownership Transfer with ~~CAS~~ CAT

`CAT(x.f, y, z)` — ”Transfer ownership of `x.f` to `y` and of `z` to `x.f`”



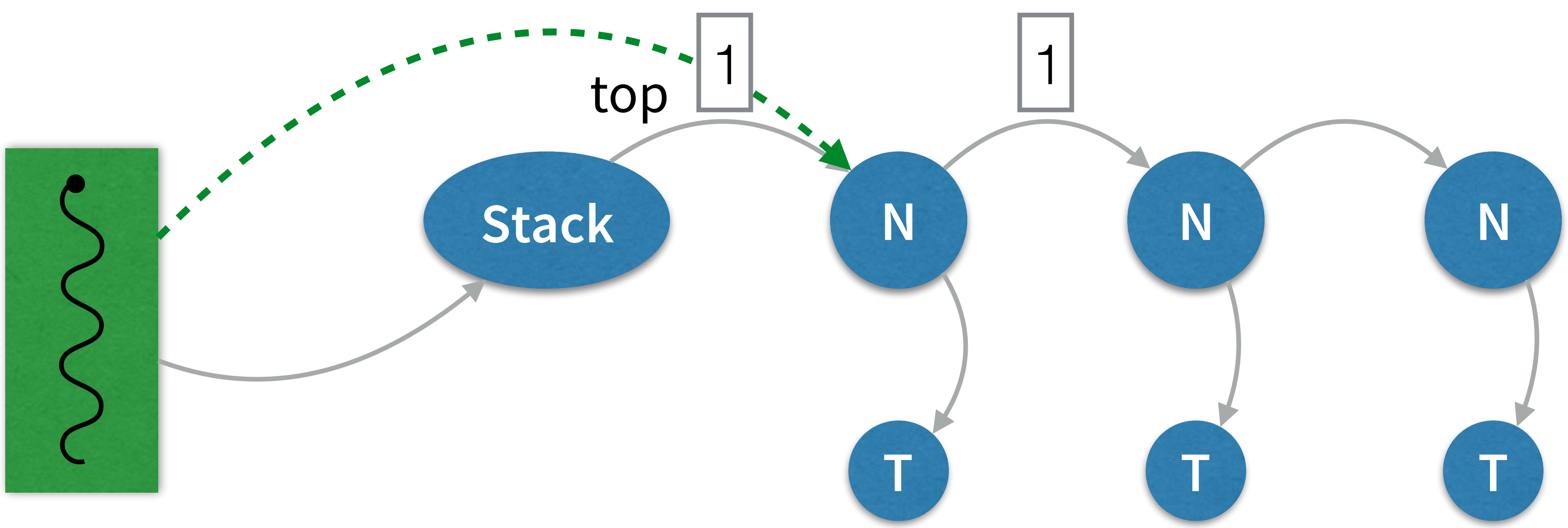
# Ownership Transfer in Action

```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAT(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```



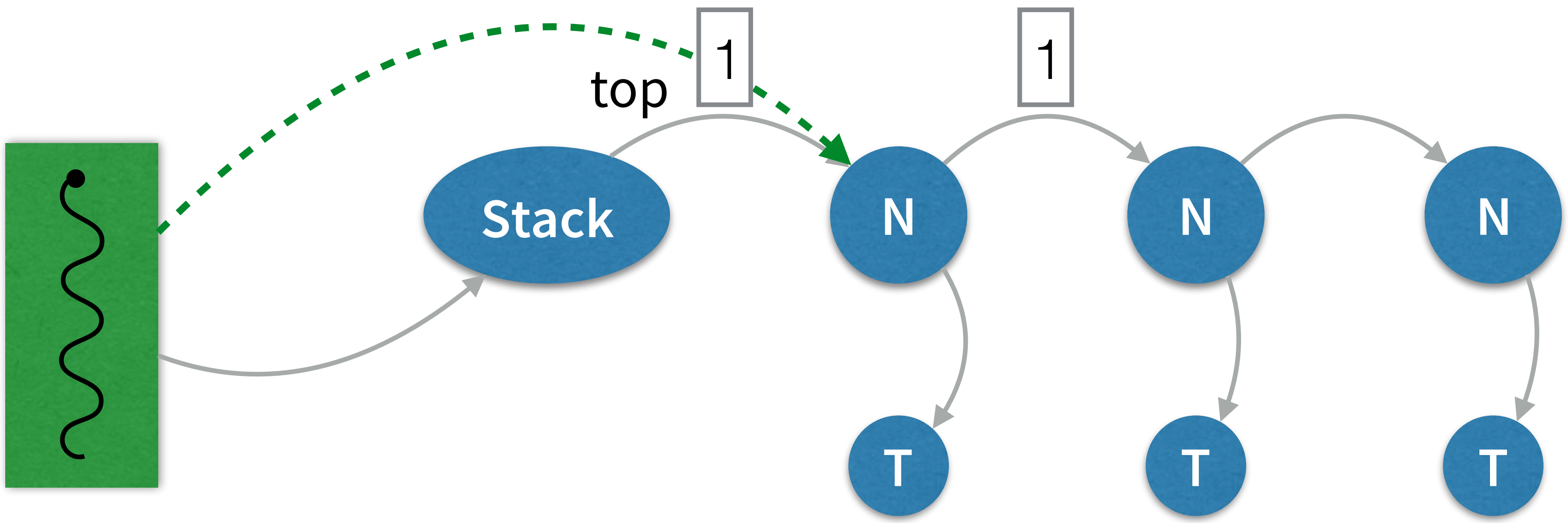
# Ownership Transfer in Action

```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAT(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```



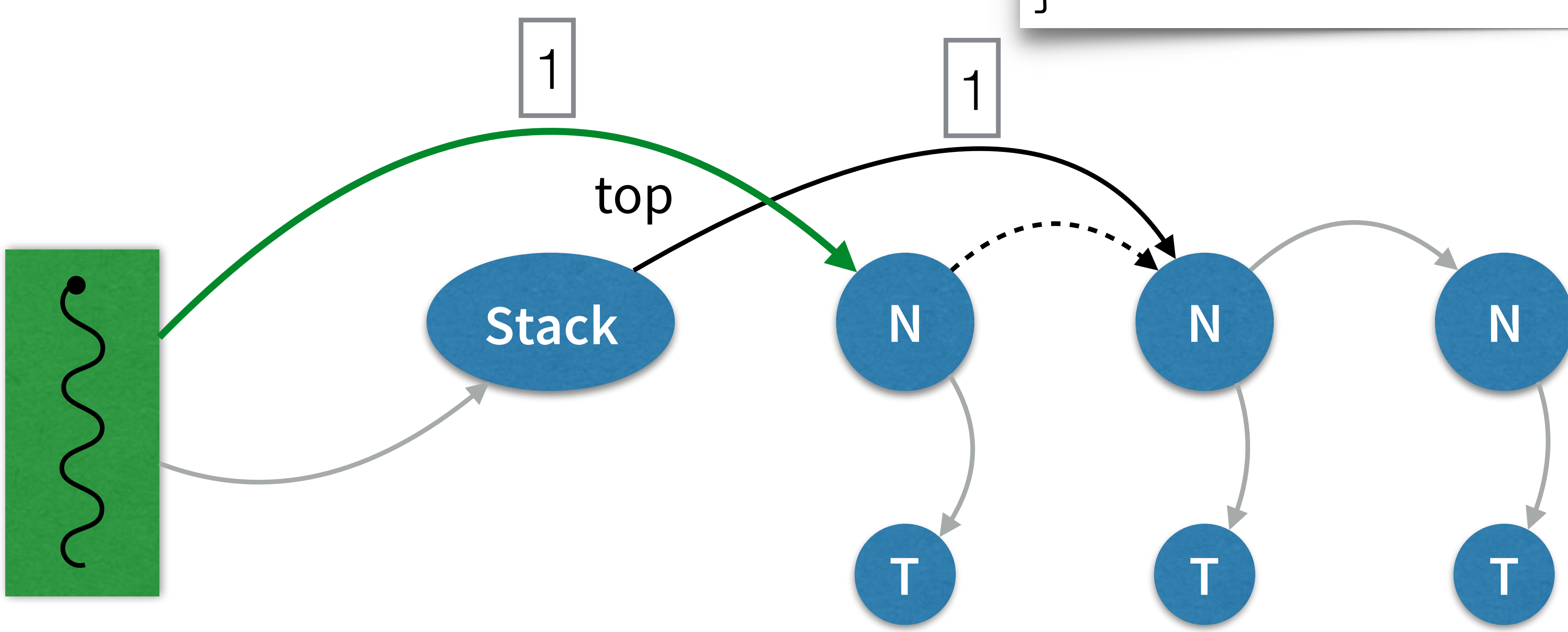
# Ownership Transfer in Action

```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAT(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```



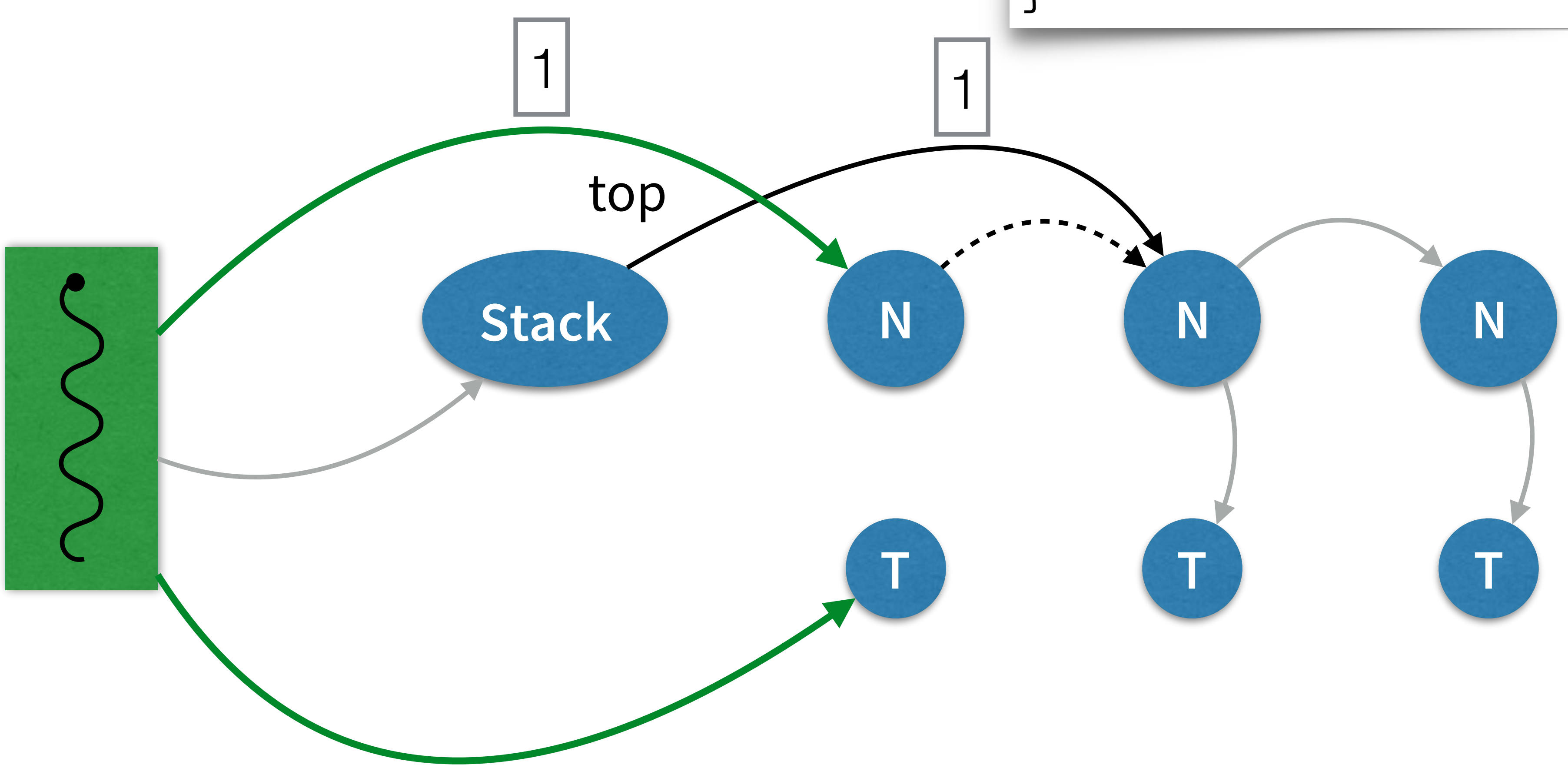
# Ownership Transfer in Action

```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAT(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```

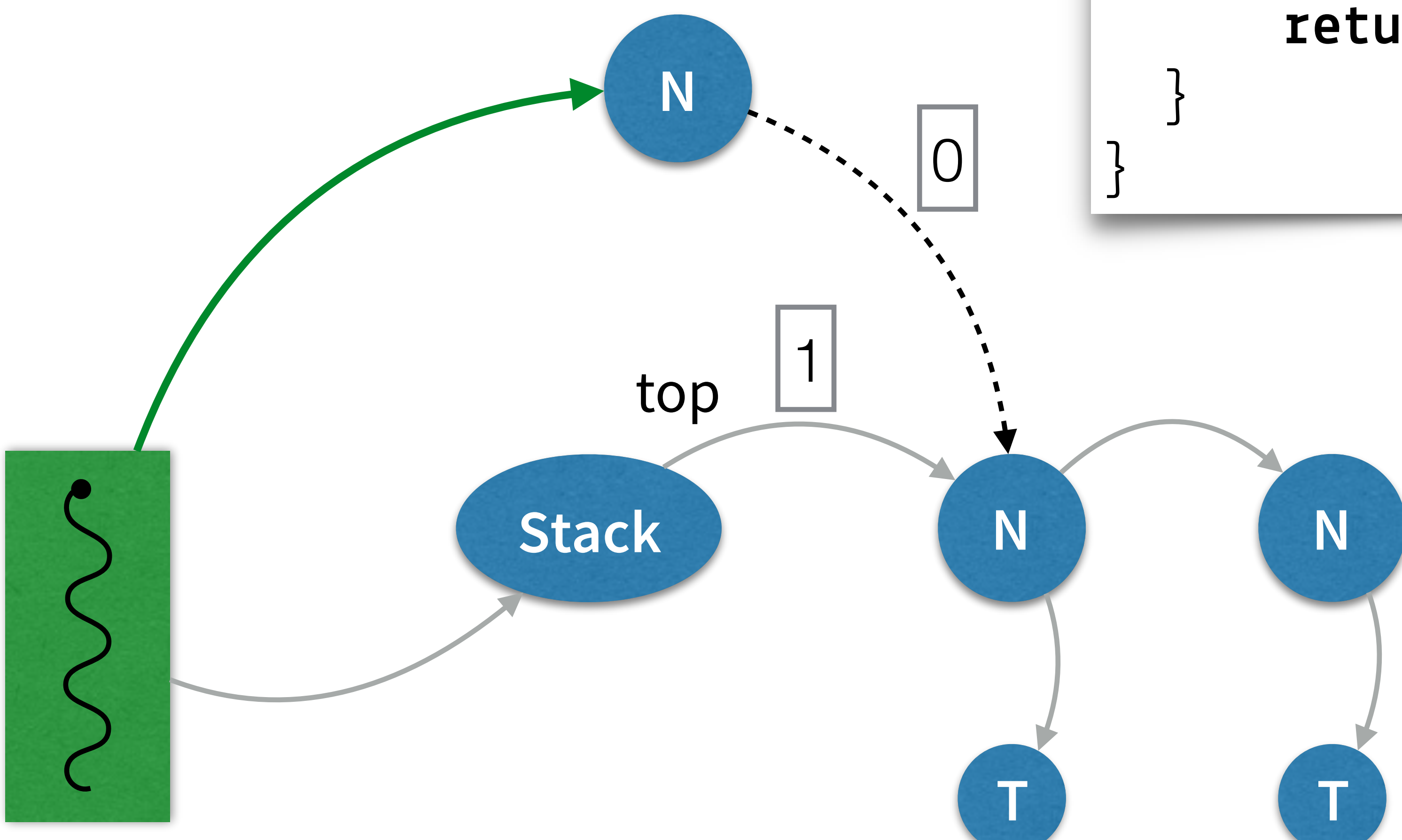


# Ownership Transfer in Action

```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAT(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```



# Ownership Transfer in Action



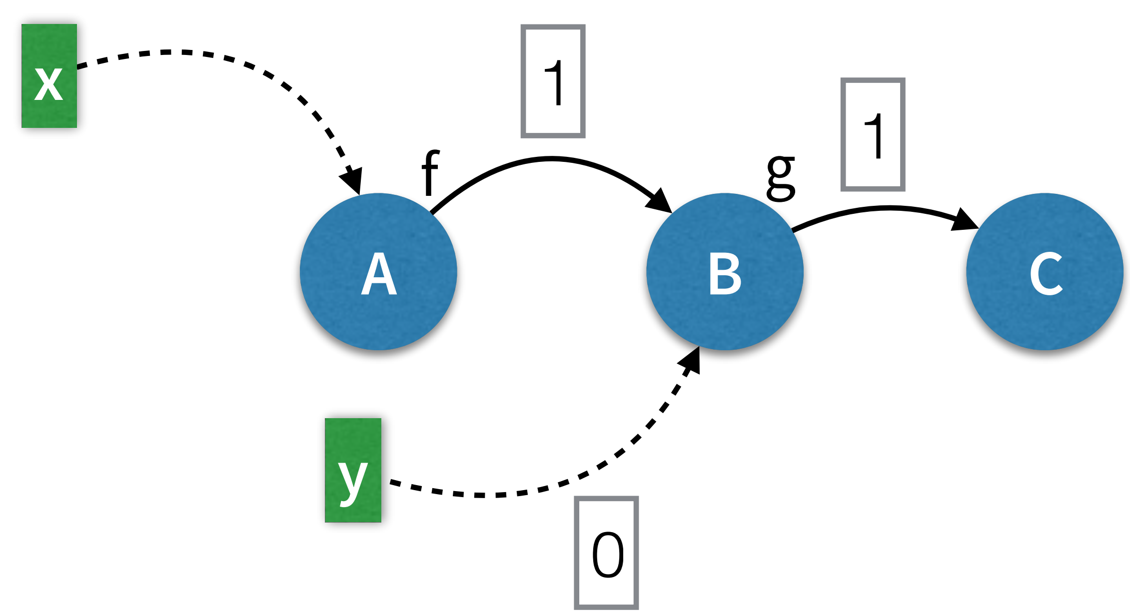

```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAT(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```

# Atomic Ownership Transfer with ~~CAS~~ CAT

`CAT(x.f, y, z)` — ”Transfer ownership of `x.f` to `y` and of `z` to `x.f`”

Unlink from a chain of objects:

`CAT(x.f, y, y.g)` — ”Transfer ownership of `x.f` to y and of `y.g` to `x.f`”

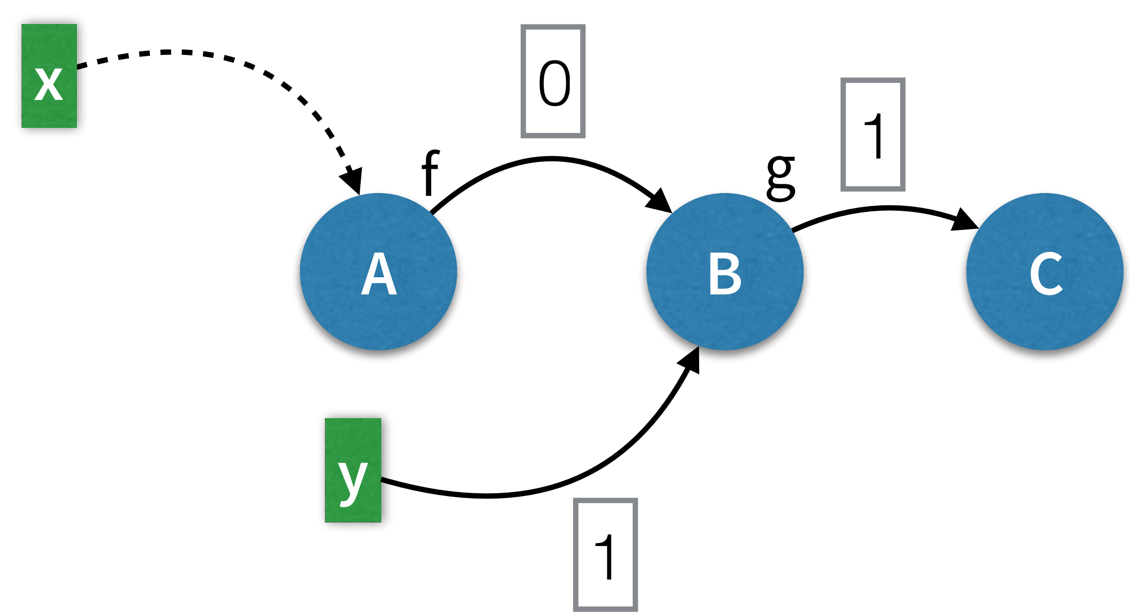


# Atomic Ownership Transfer with ~~CAS~~ CAT

`CAT(x.f, y, z)` — "Transfer ownership of `x.f` to `y` and of `z` to `x.f`"

Unlink from a chain of objects:

`CAT(x.f, y, y.g)` — "Transfer ownership of `x.f` to `y` and of `y.g` to `x.f`"

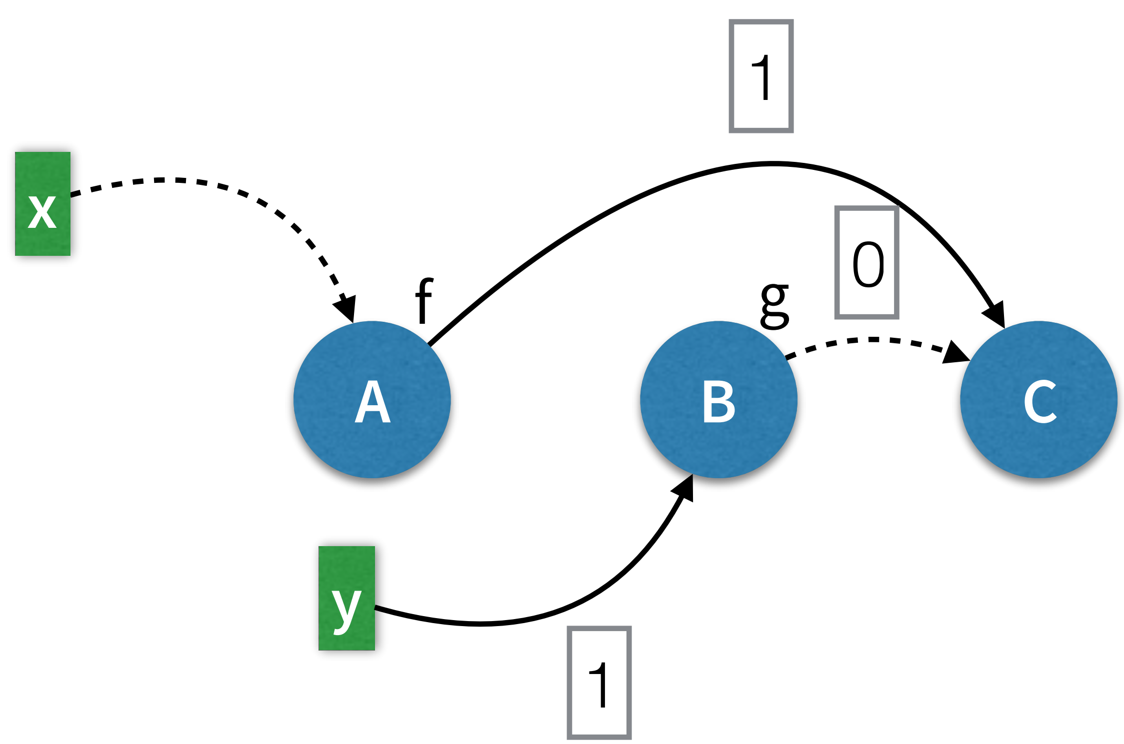


# Atomic Ownership Transfer with ~~CAS~~ CAT

`CAT(x.f, y, z)` — "Transfer ownership of `x.f` to `y` and of `z` to `x.f`"

Unlink from a chain of objects:

`CAT(x.f, y, y.g)` — "Transfer ownership of `x.f` to y and of `y.g` to `x.f`"



# Atomic Ownership Transfer with ~~CAS~~ CAT

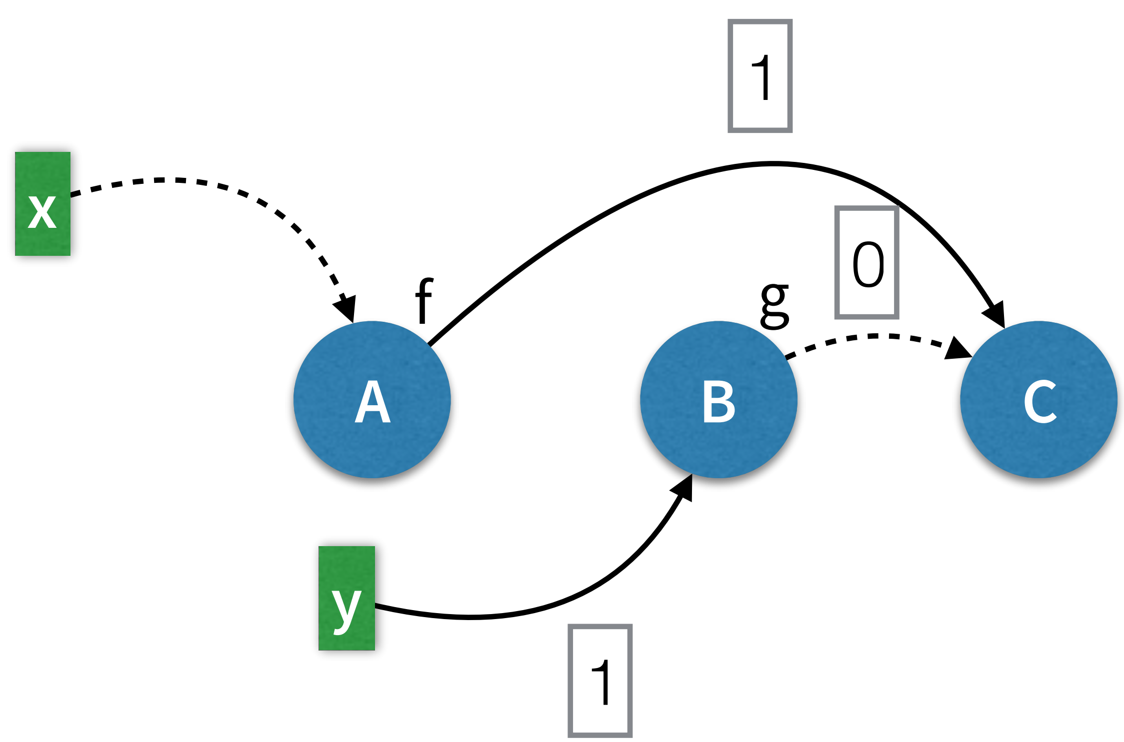
`CAT(x.f, y, z)` — "Transfer ownership of `x.f` to `y` and of `z` to `x.f`"

Unlink from a chain of objects:

`CAT(x.f, y, y.g)` — "Transfer ownership of `x.f` to `y` and of `y.g` to `x.f`"

Link into a chain of objects:

`CAT(x.f, y.g, y)` — "Transfer ownership of `x.f` to `y.g` and of `y` to `x.f`"



# Atomic Ownership Transfer with ~~CAS~~ CAT

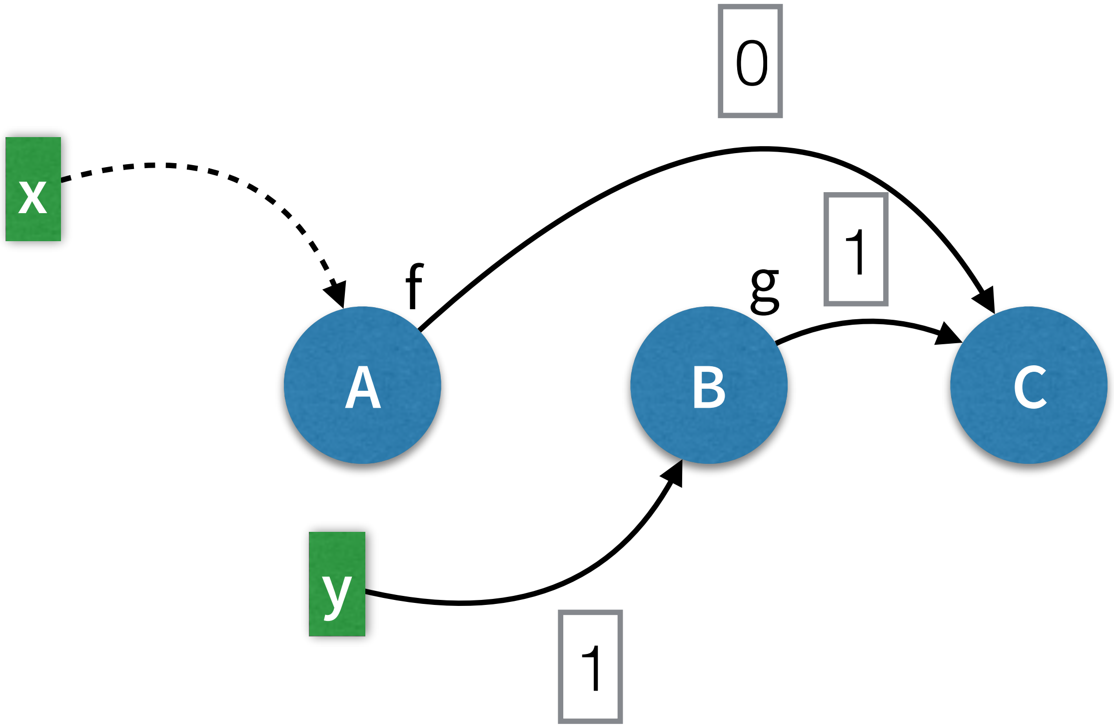
`CAT(x.f, y, z)` — ”Transfer ownership of `x.f` to `y` and of `z` to `x.f`”

Unlink from a chain of objects:

`CAT(x.f, y, y.g)` — ”Transfer ownership of `x.f` to `y` and of `y.g` to `x.f`”

Link into a chain of objects:

`CAT(x.f, y.g, y)` — ”Transfer ownership of `x.f` to `y.g` and of `y` to `x.f`”



# Atomic Ownership Transfer with ~~CAS~~ CAT

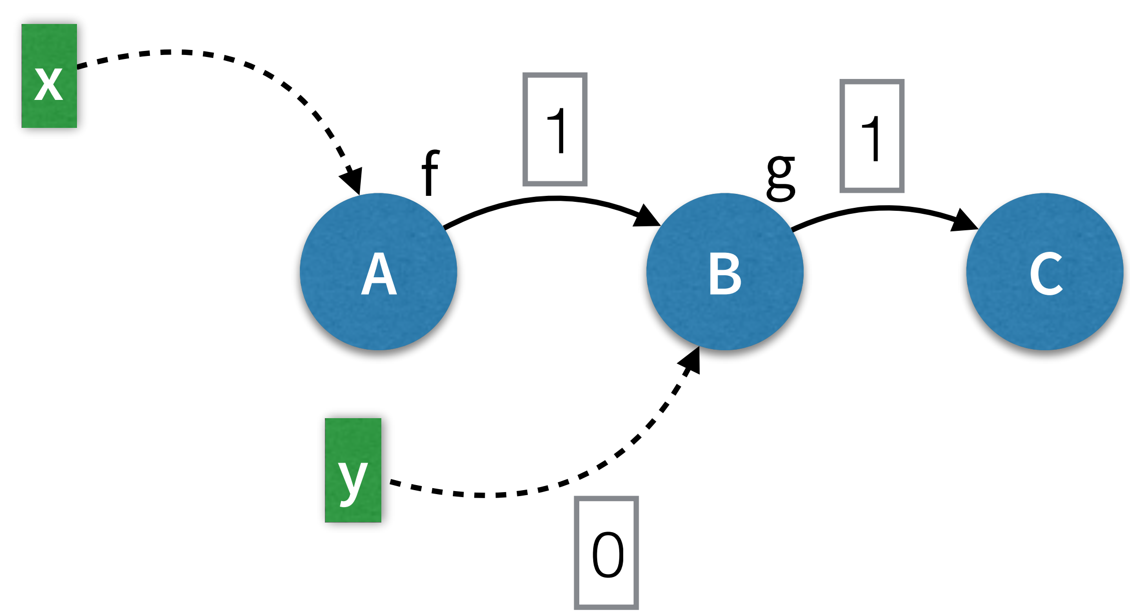
`CAT(x.f, y, z)` — "Transfer ownership of `x.f` to `y` and of `z` to `x.f`"

Unlink from a chain of objects:

`CAT(x.f, y, y.g)` — "Transfer ownership of `x.f` to `y` and of `y.g` to `x.f`"

Link into a chain of objects:

`CAT(x.f, y.g, y)` — "Transfer ownership of `x.f` to `y.g` and of `y` to `x.f`"



# Atomic Ownership Transfer with ~~CAS~~ CAT

`CAT(x.f, y, z)` — ”Transfer ownership of `x.f` to `y` and of `z` to `x.f`”

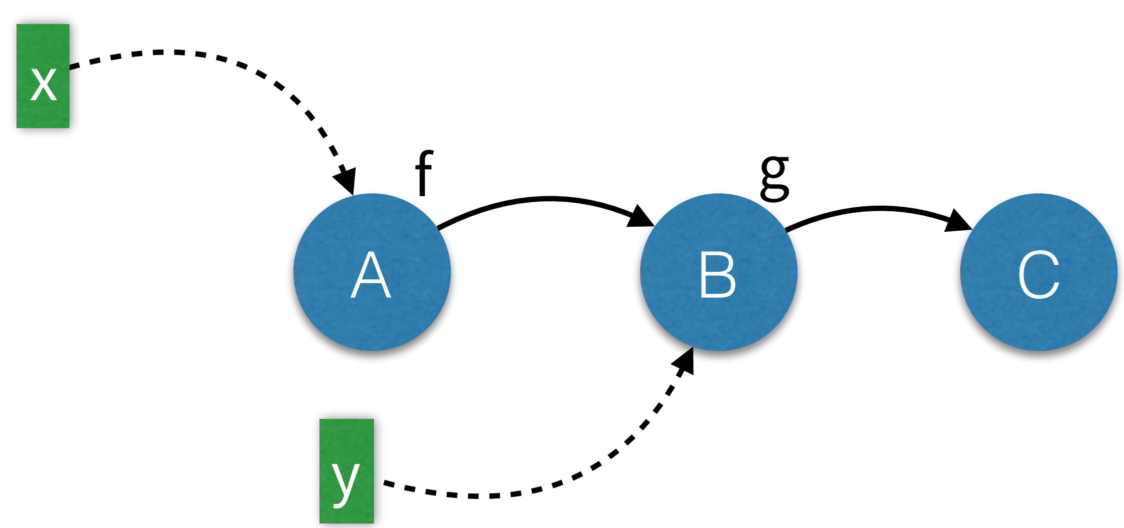
Unlink from a chain of objects:

`CAT(x.f, y, y.g)` — ”Transfer ownership of `x.f` to `y` and of `y.g` to `x.f`”

Link into a chain of objects:

`CAT(x.f, y.g, y)` — ”Transfer ownership of `x.f` to `y.g` and of `y` to `x.f`”

Three kinds of CATs



# Linear Ownership with Lock-free CAT

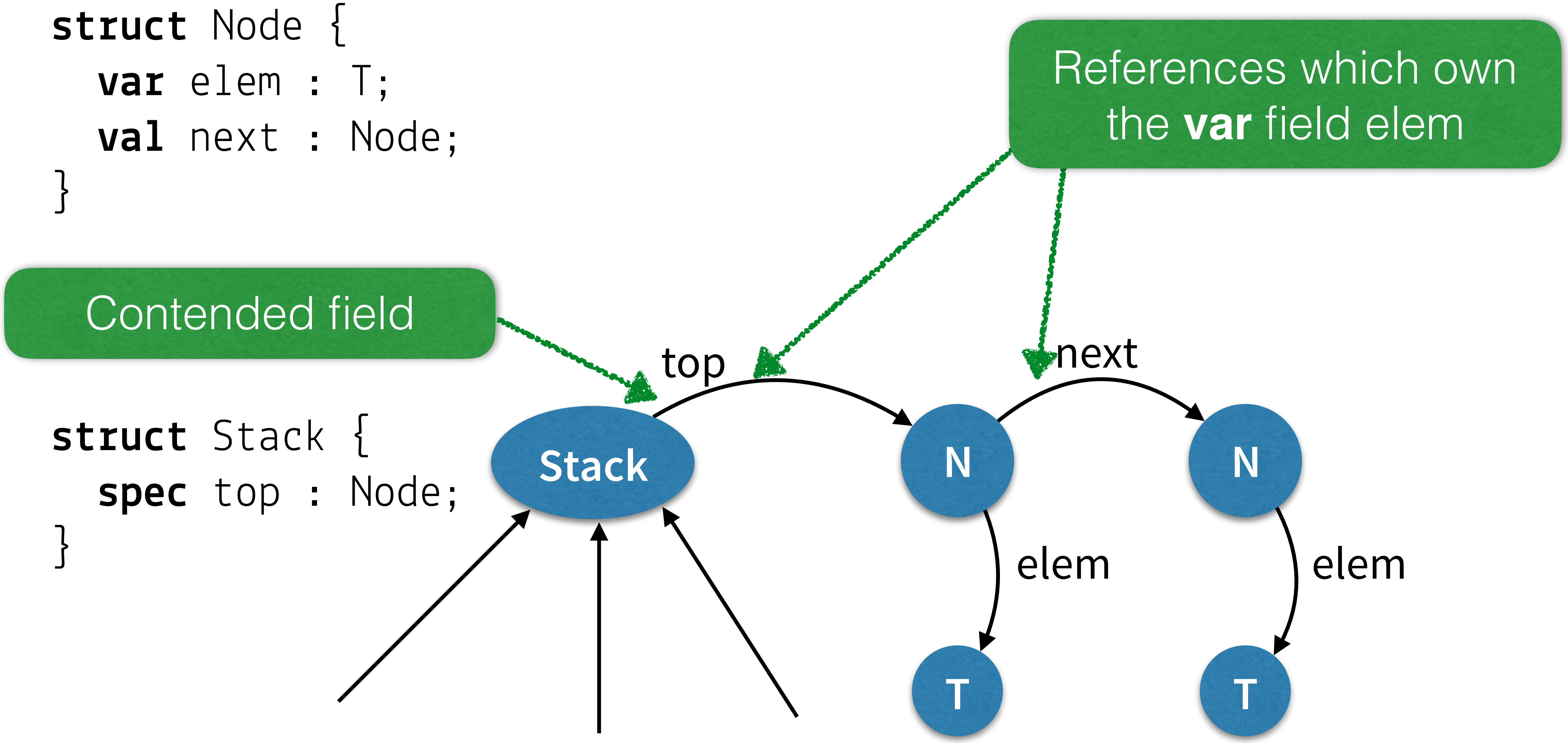
# LOLCAT

# Ownership in LOLCAT

- References do not own objects, but rather the (mutable) fields of the objects

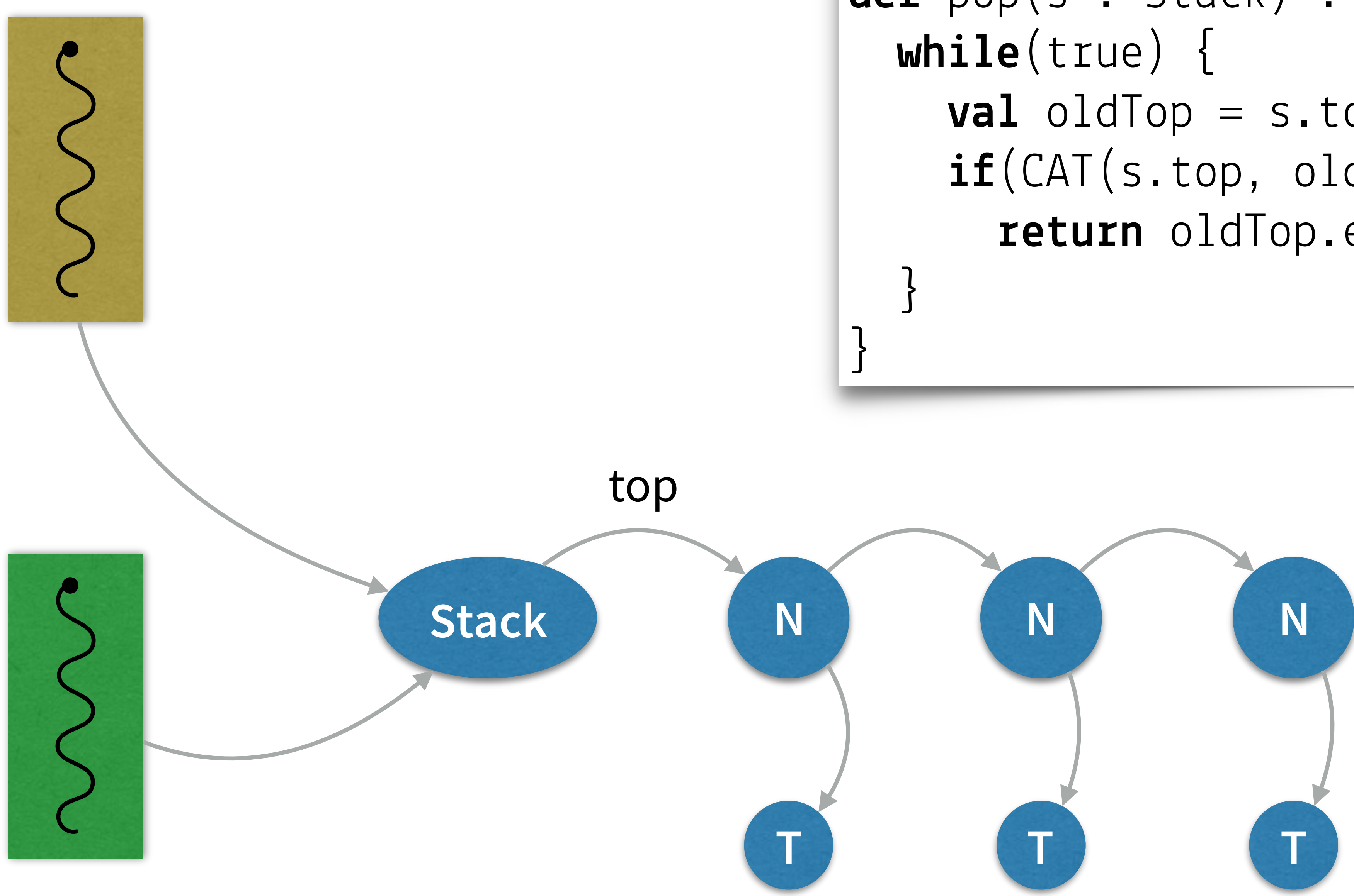
```
struct Node {
  var elem : T;
  val next : Node;
}
```

```
struct Stack {
  spec top : Node;
}
```



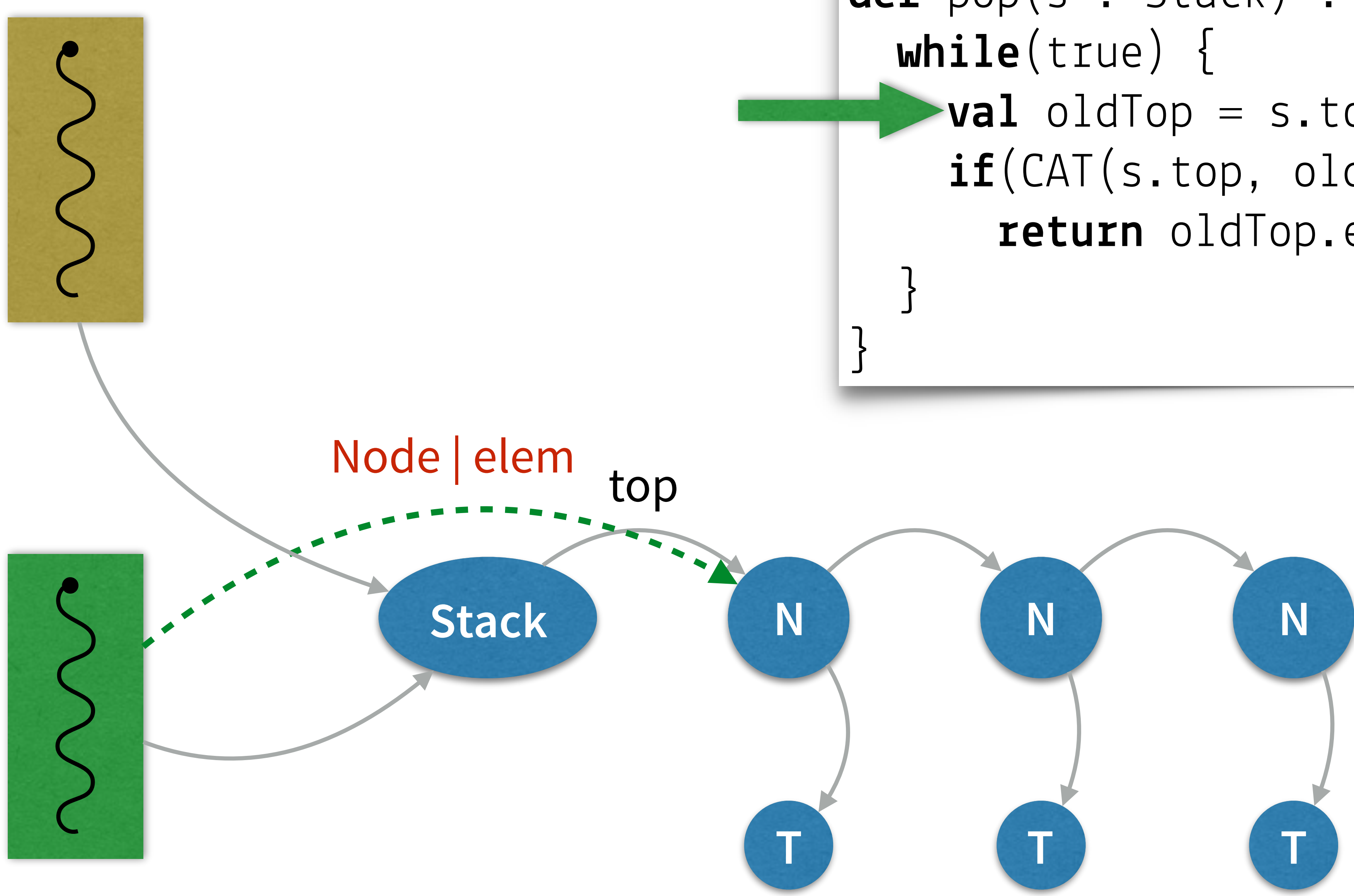
# Ownership Tracked using Flow-sensitive Types

```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAT(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```



# Ownership Tracked using Flow-sensitive Types

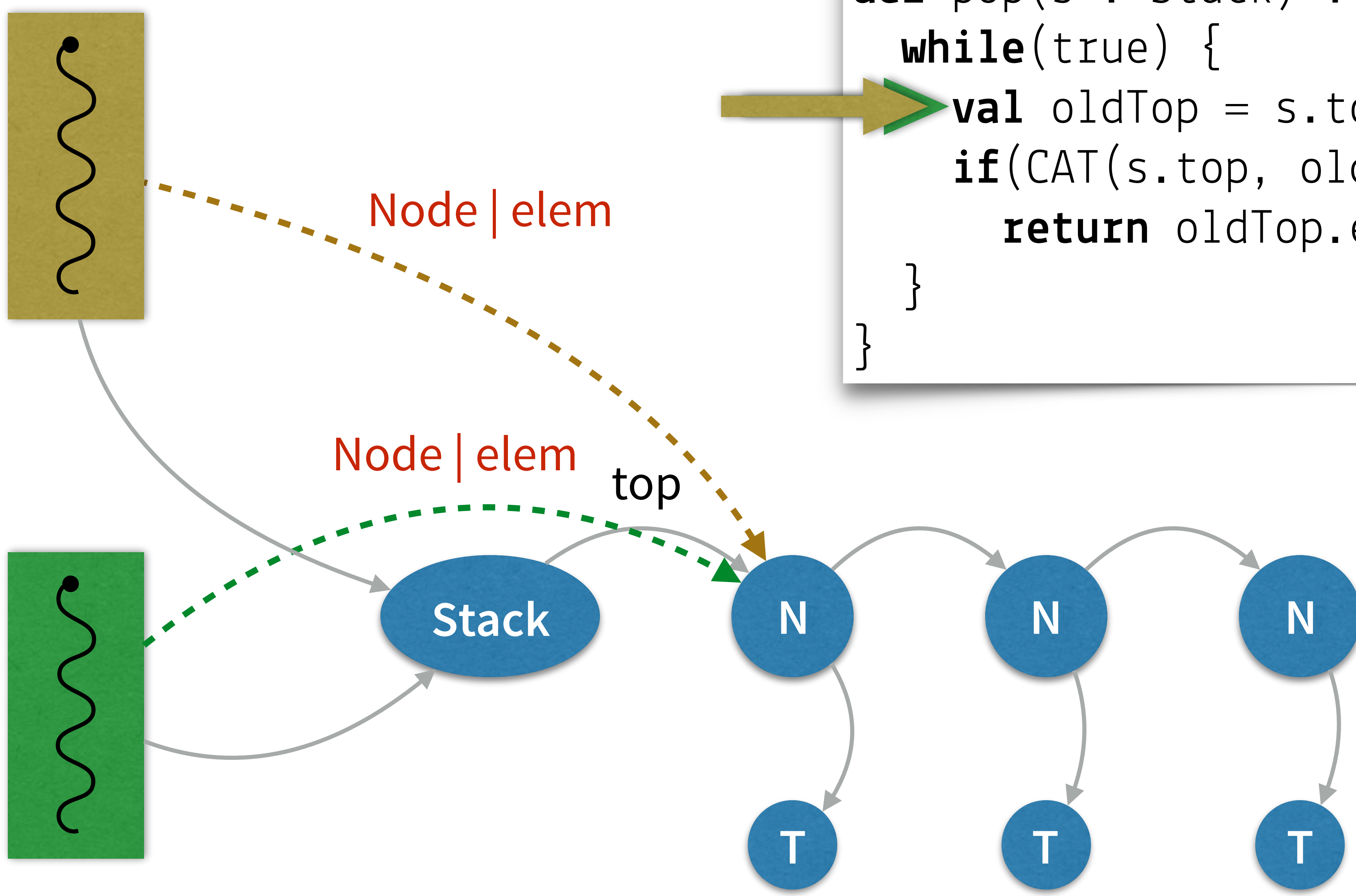
```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;   Node | elem
    if(CAS(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```



# Ownership Tracked using Flow-sensitive Types

```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAT(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```

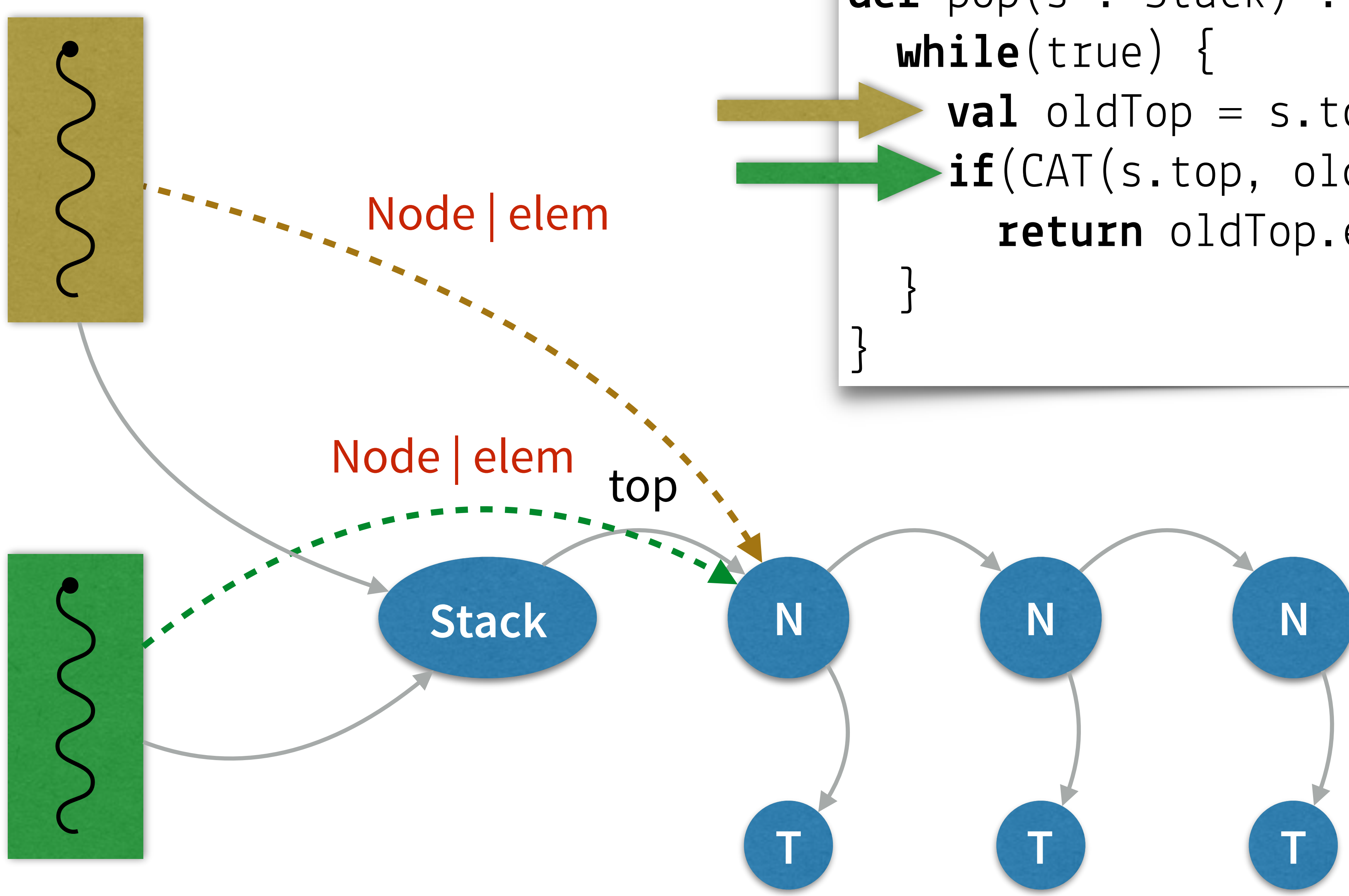
Node | elem



# Ownership Tracked using Flow-sensitive Types

```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAT(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```

Node | elem

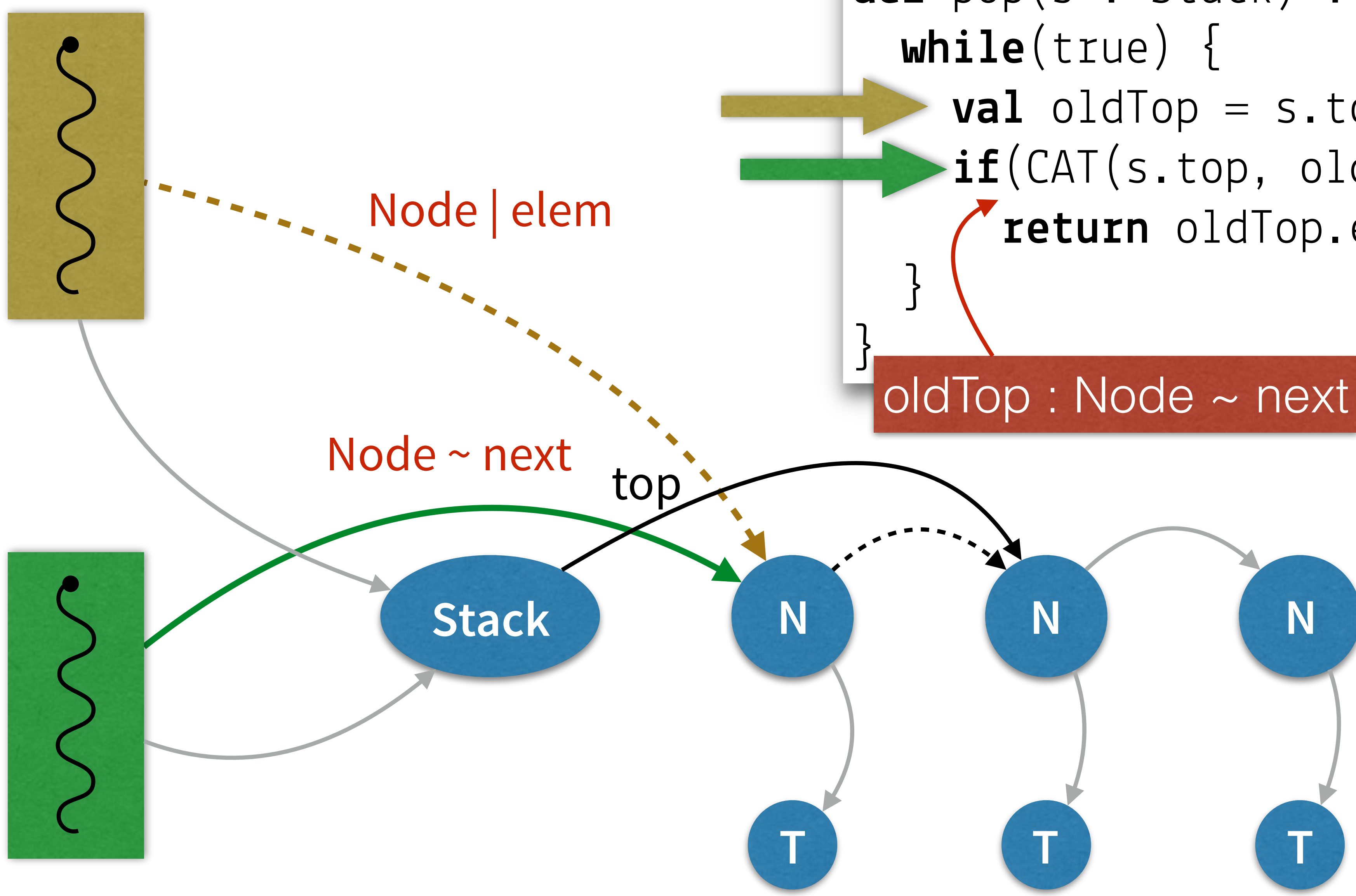


# Ownership Tracked using Flow-sensitive Types

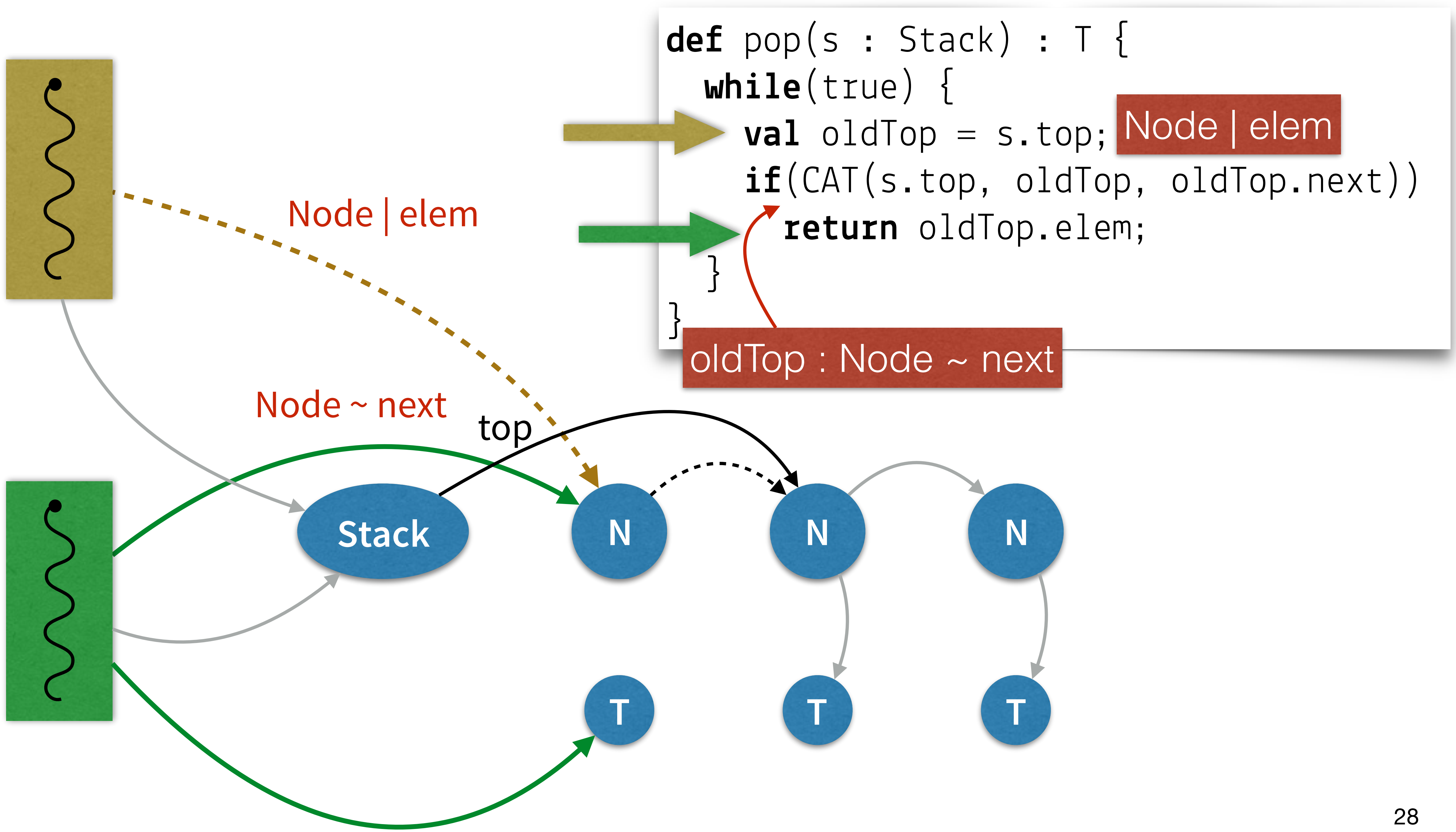
```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAT(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```

Node | elem

oldTop : Node ~ next



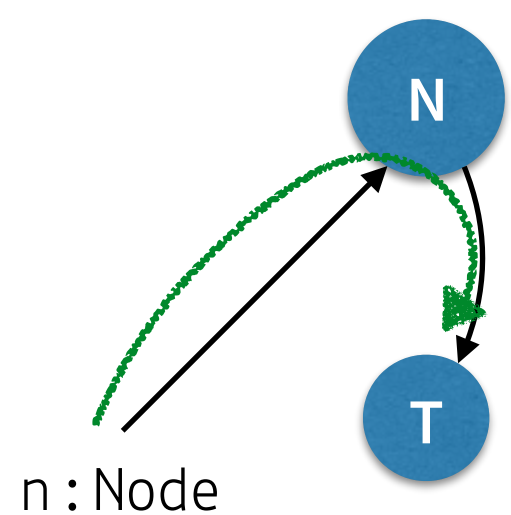
# Ownership Tracked using Flow-sensitive Types



# Restricted References

- A reference of type `Node` can access the full node

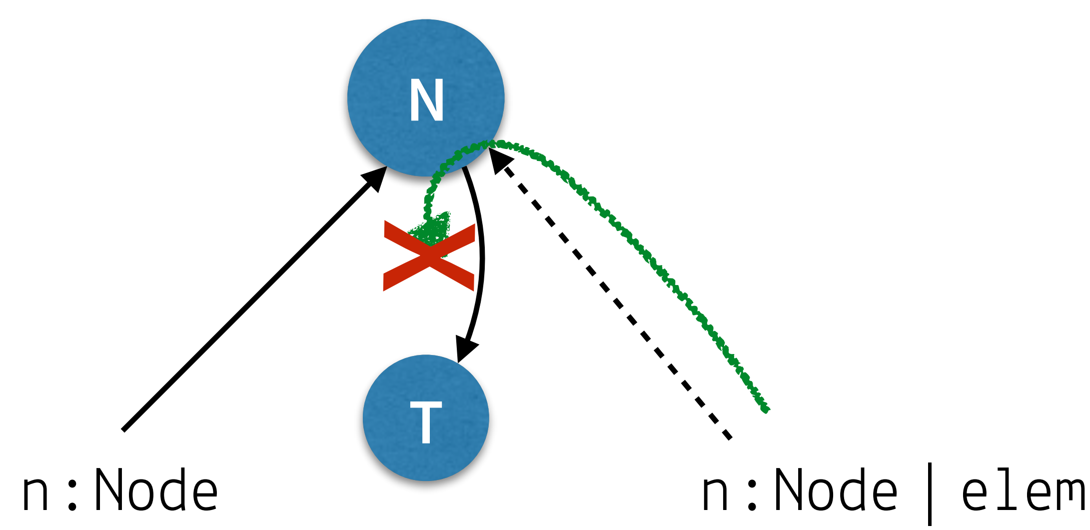
```
struct Node {
  var elem : T;
  val next : Node;
}
```



# Restricted References

- A reference of type `Node` can access the full node
- A reference of type `Node | elem` cannot access the **var** field elem

```
struct Node {
  var elem : T;
  val next : Node;
}
```

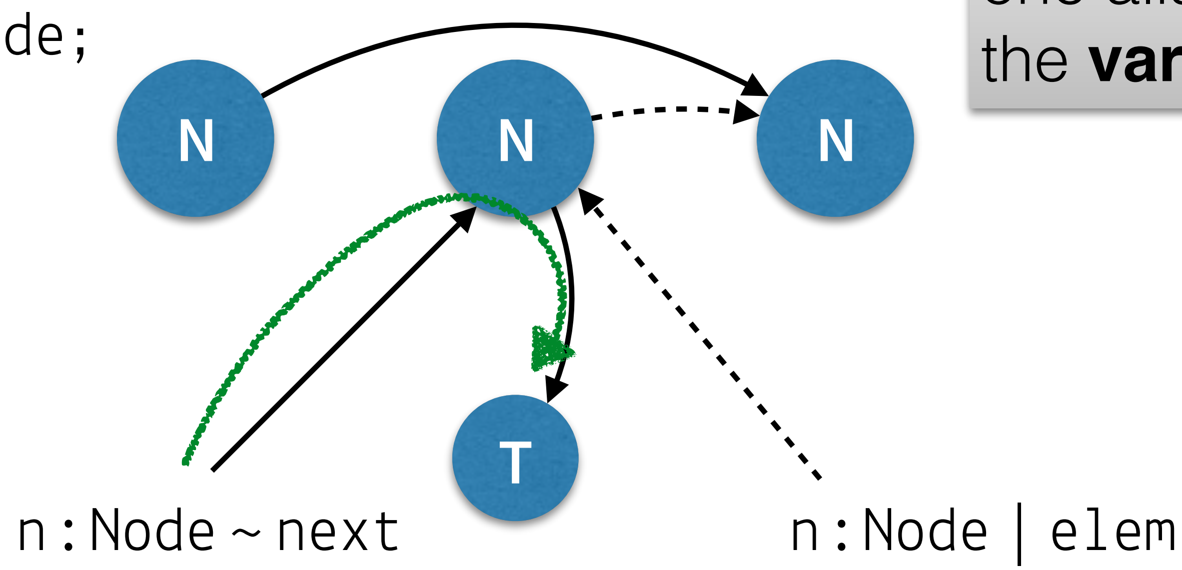


# Restricted References

- A reference of type `Node` can access the full node
- A reference of type `Node | elem` cannot access the **var** field elem
- A reference of type `Node ~ next` can access the **var** field `elem`, but has a **val** field `next` without ownership

```
struct Node {
  var elem : T;
  val next : Node;
}
```

**Main invariant:**
For all aliases of a Node, at most one alias has ownership of the **var** field `elem`



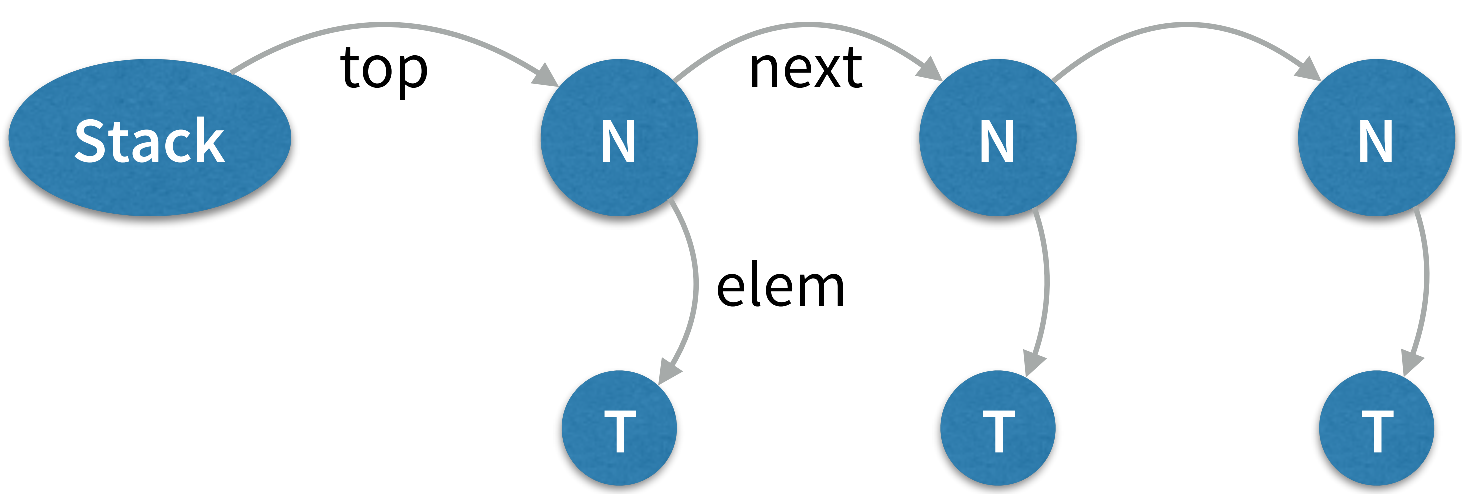
# A Lock-free Stack

```
struct Node {
  var elem : T;
  val next : Node;
}

struct Stack {
  var top : Node;
}

def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAS(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}

def push(s : Stack, x : T) : void {
  val newTop = new Node(x);
  newTop.next = s.top;
  while(true) {
    if(CAS(s.top, newTop.next, newTop))
      return;
    newTop.next = s.top;
  }
}
```



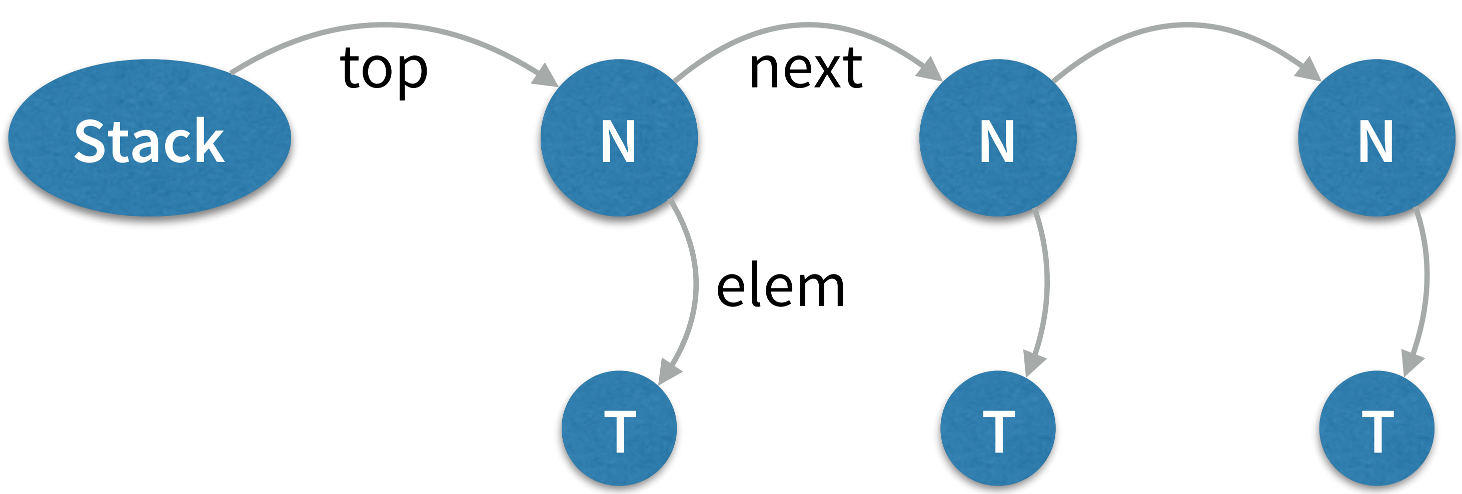
# A Lock-free Stack in LOLCAT

```
struct Node {
  var elem : T;
  val next : Node;
}
```

```
struct Stack {
  spec top : Node;
}
```

```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAT(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```

```
def push(s : Stack, x : T) : void {
  val newTop = new Node(x);
  newTop.next = s.top;
  while(true) {
    if(CAT(s.top, newTop.next, newTop))
      return;
    newTop.next = s.top;
  }
}
```

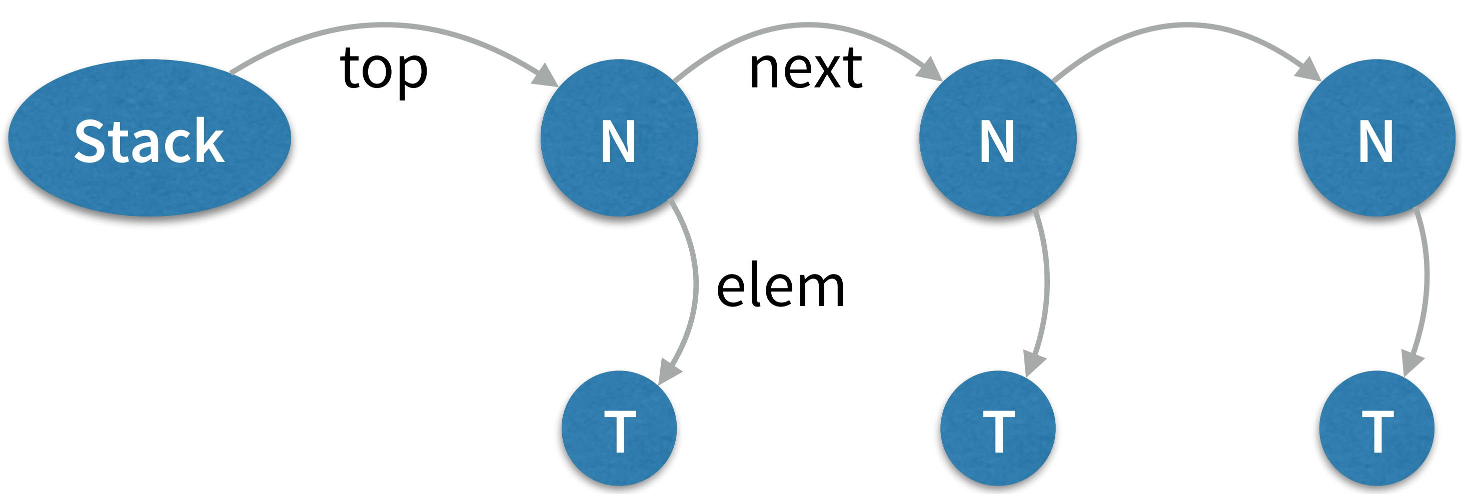


# A Lock-free Stack in LOLCAT (more explicit)

```
struct Node {          struct Stack {
  var elem : T;          spec top : Node;
  val next : Node;     }
}
```

```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = speculate s.top;
    if(CAT(s.top, oldTop, oldTop.next))
      return consume oldTop.elem;
  }
}
```

```
def push(s : Stack, x : T) : void {
  val newTop = new Node(consume x);
  newTop.next = speculate s.top;
  while(true) {
    if(CAT(s.top, newTop.next, newTop))
      return;
    newTop.next = speculate s.top;
  }
}
```



# Also in the Paper

- LOLCAT formalized and proven sound
  - — Linear ownership: at most one alias may access the **`var`** fields of an object
- Three fundamental lock-free data-structures
  - — Treiber Stack
  - — Michael—Scott Queue
  - — Tim Harris List
- Prototype implementation in Encore
  - — OO support


Relaxed Linear References for Lock-free Data Structures *

Elias Castegren[1] and Tobias Wrigstad[1]

1 Uppsala University, Sweden, first.last@it.uu.se

Abstract

1 Introduction

# Not in the Paper

- Additional examples (implemented in Encore)
  - Skip list [Fomitchev & Ruppert]
  - BST [Ellen et al.]
  - Lazy list-based set [Heller et al.]
  - Spin-locks
- Type-Assisted Automatic Garbage Collection for Lock-Free Data Structures [Yang & Wrigstad, ISMM'17]
  - Pluggable concurrent garbage collector
  - Relies on LOLCAT types to track when objects may be collected

# Summary

- Relaxed linearity: unbounded aliasing with linear ownership
  - Access to mutable (**`var`**) fields is always exclusive
  - Compare-and-swap for atomic ownership transfer
  - Controlled sharing of certain fields (**`val`** and **`spec`**)
  - Guarantee absence of (uncontrolled) data-races in lock-free data structures
- LOLCAT provides meaningful types for common patterns in lock-free programming
  - Speculation (`Node | elem`)
  - Publication (`CAT(x.f, y, z)`)
  - Acquisition (`Node ~ next`)
  - Types inferred (modulo struct declarations)
- Prototype implementation in Encore
  - Open source: try it yourself!


*http://www.kristianstadik.se/www/wordpress/wp-content/uploads/2016/06/midsommar.jpg*

# Summary

- Relaxed linearity: unbounded aliasing with linear ownership
  - Access to mutable (**`var`**) fields is always exclusive
  - Compare-and-swap for atomic ownership transfer
  - Controlled sharing of certain fields (**`val`** and **`spec`**)
  - Guarantee absence of (uncontrolled) data-races in lock-free data structures
- LOLCAT provides meaningful types for common patterns in lock-free programming
  - Speculation (`Node | elem`)
  - Publication (`CAT(x.f, y, z)`)
  - Acquisition (`Node ~ next`)
  - Types inferred (modulo struct declarations)
- Prototype implementation in Encore
  - Open source: try it yourself!