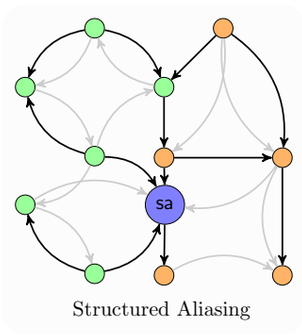
Reference Capabilities for Concurrency and Scalability

An Experience Report

Elias Castegren, Tobias Wrigstad

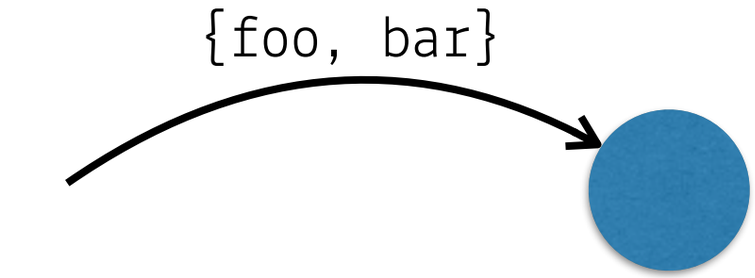
OCAP, Vancouver
October 24th 2017



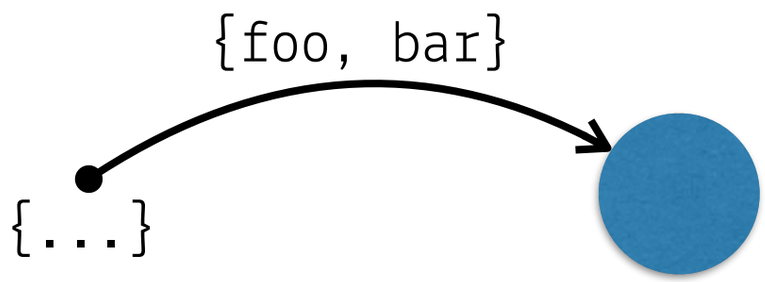
Structured Aliasing

What the CAP?

- Object capability
  - An unforgeable reference to an object
  - The permission to perform (one or more) operations on that object

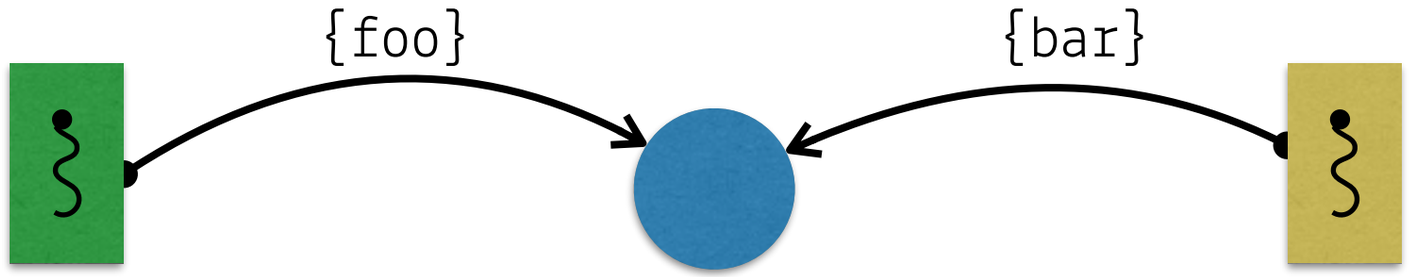


- Reference capability
  - An (unforgeable) object capability
  - The permission to perform (one or more) operations *on that reference*

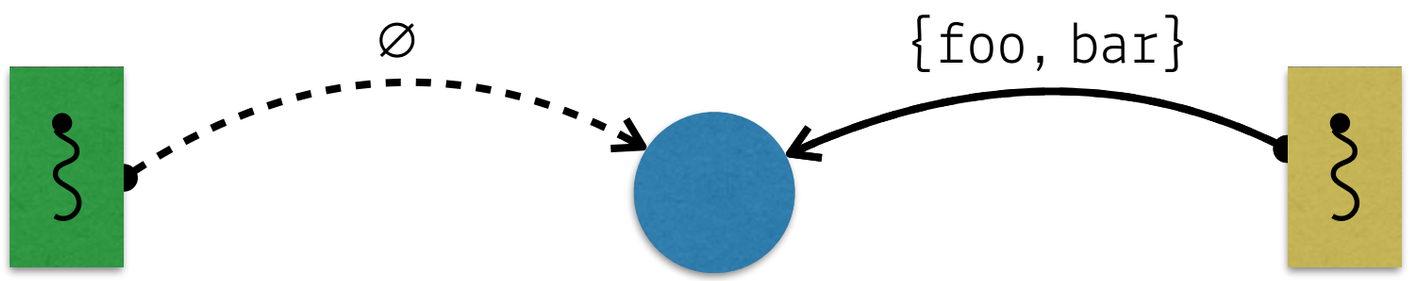


This Talk

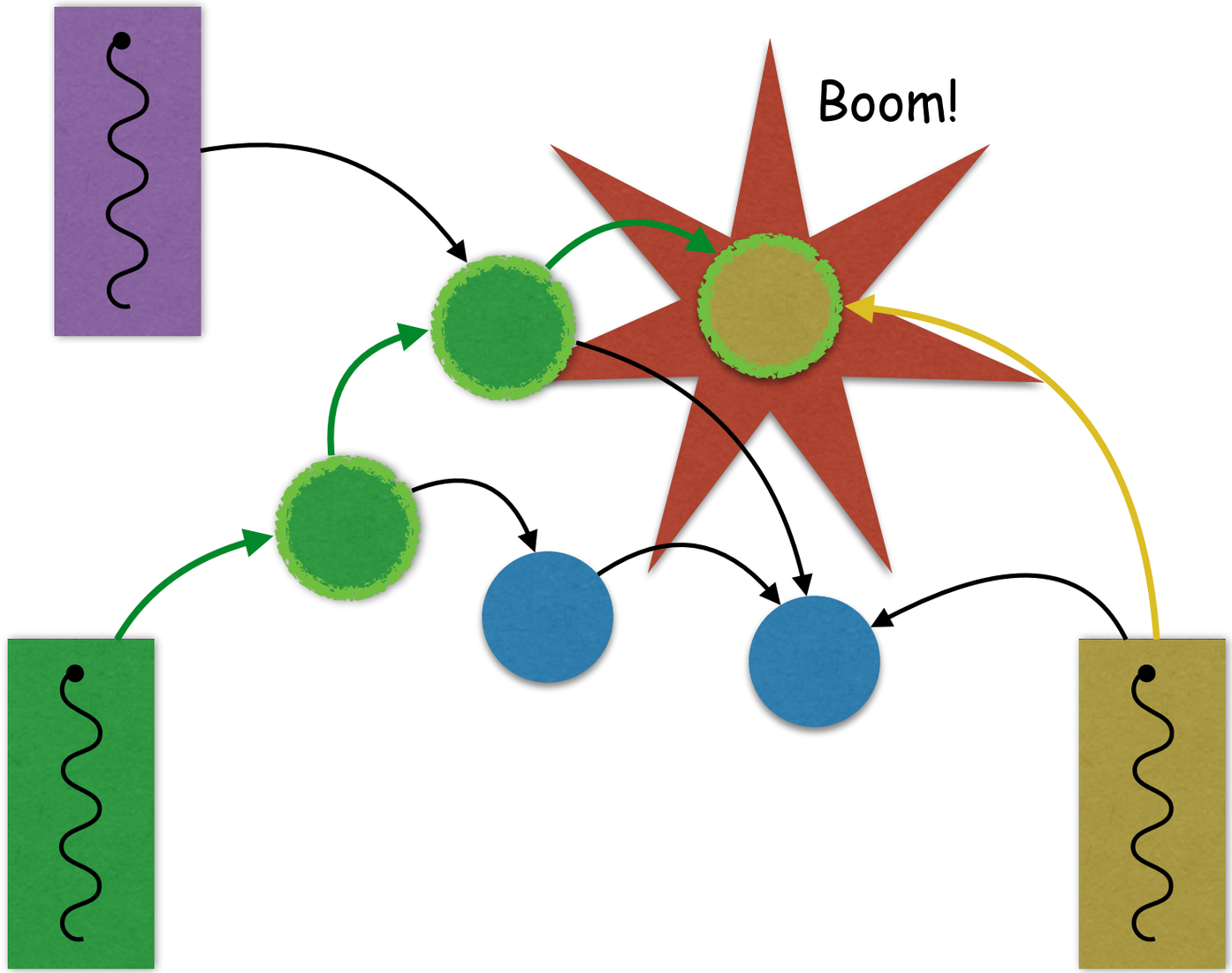
- Reference capabilities for data-race freedom



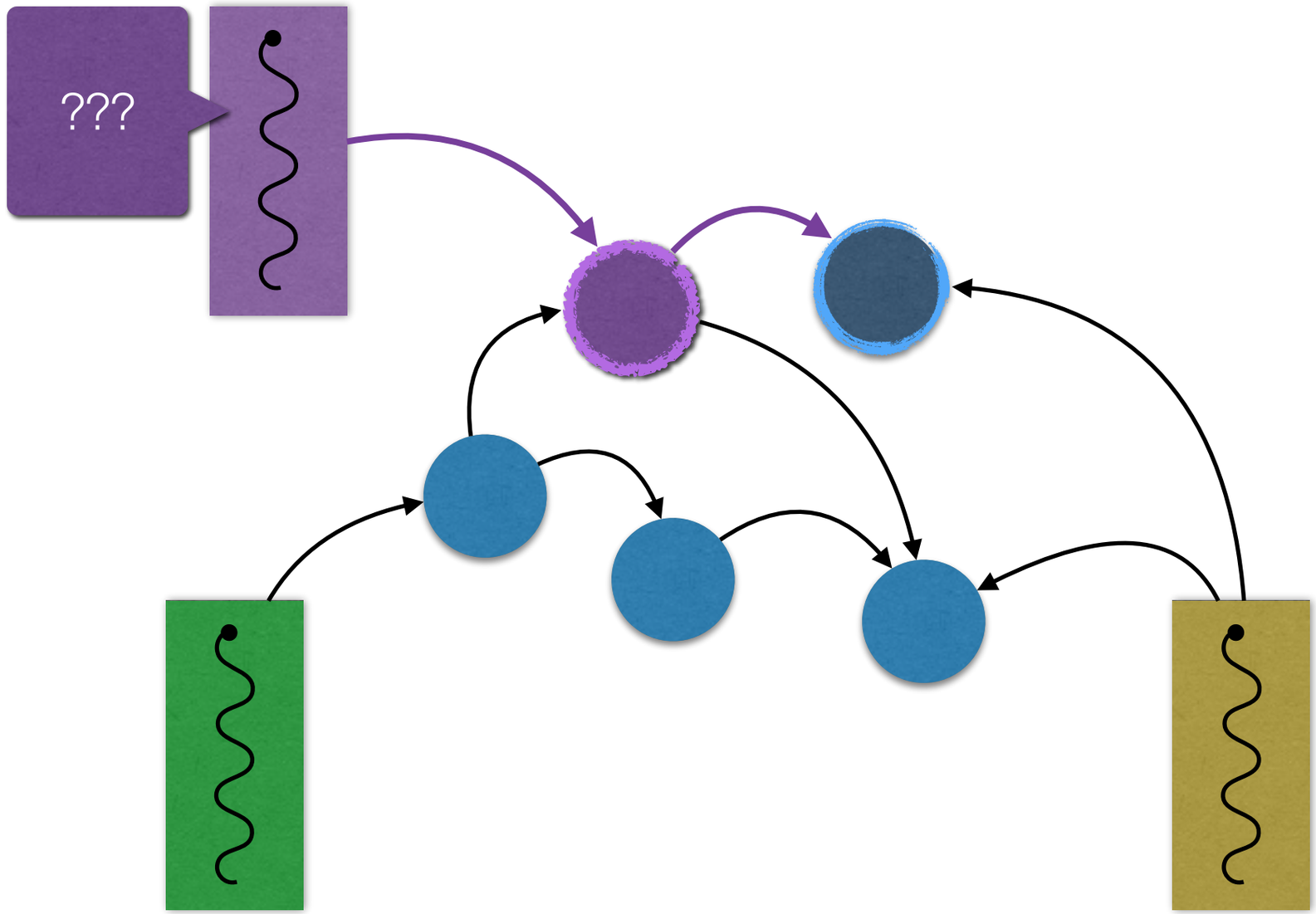
- Reference capabilities for atomic ownership transfer



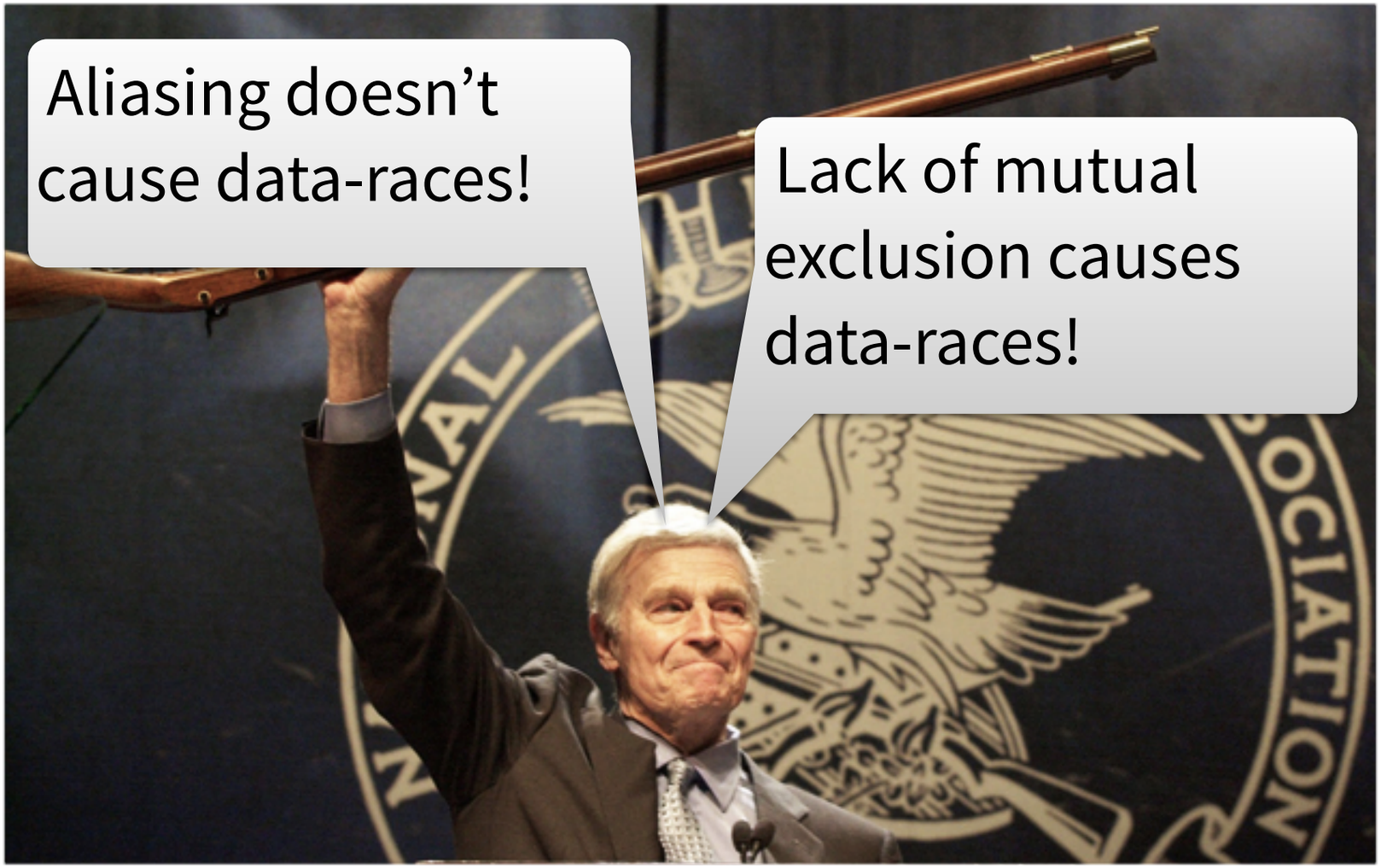
Warning: Aliasing May Cause Data-Races



Warning: Aliasing May Cause Data-Races

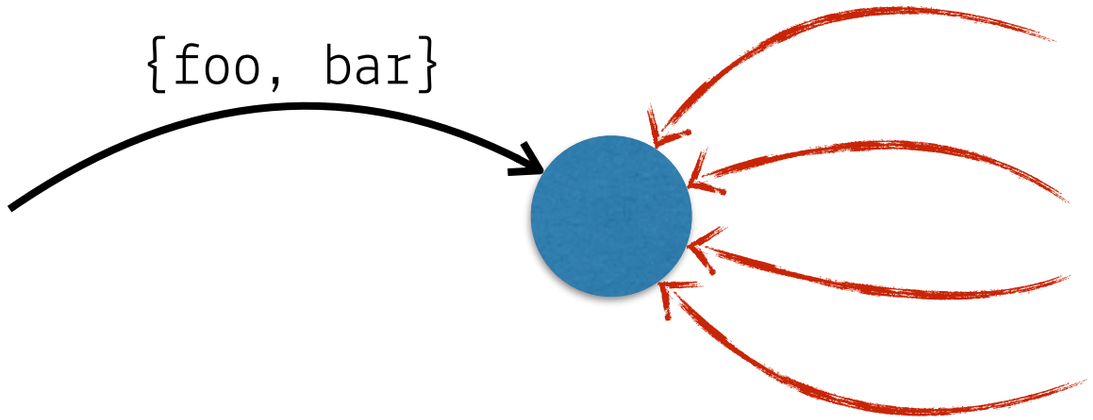


Warning: Aliasing May Cause Data-Races

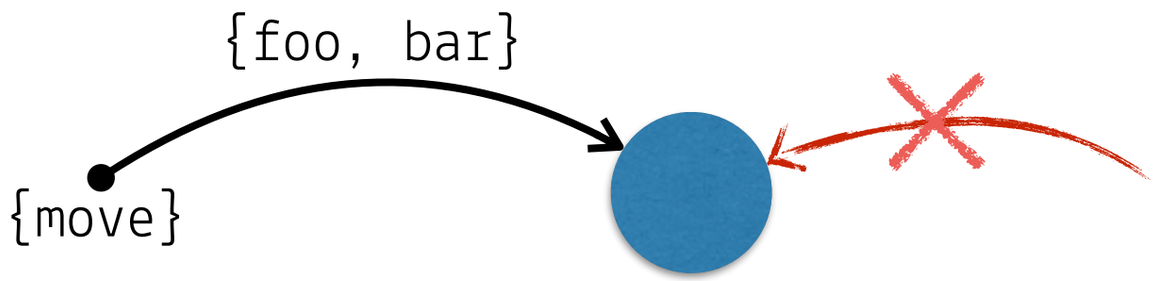


Object Capabilities Are Not Enough

- An object capability tells you something about what **you** can do with an object



- A reference capability implicitly tells you something about what **others** can do



Kappa [ECOOP ’16]

Reference Capabilities for Concurrency Control *

Elias Castegren[1] **and Tobias Wrigstad**[1]

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

Abstract

The proliferation of shared mutable state in object-oriented programming complicates software development as two seemingly unrelated operations may interact via an alias and produce unexpected results. In concurrent programming this manifests itself as data-races. Concurrent object-oriented programming further suffers from the fact that code that warrants synchronisation cannot easily be distinguished from code that does not. The burden is placed solely on the programmer to reason about alias freedom, sharing across threads and side-effects to deduce where and when to apply concurrency control, without inadvertently blocking parallelism.

This paper presents a reference capability approach to concurrent and parallel object-oriented programming where all uses of aliases are guaranteed to be data-race free. The static type of an alias describes its possible sharing without using explicit ownership or effect annotations. Type information can express non-interfering deterministic parallelism without dynamic concurrency control, thread-locality, lock-based schemes, and guarded-by relations giving multi-object atomicity to nested data structures. Unification of capabilities and traits allows trait-based reuse across multiple concurrency scenarios with minimal code duplication. The resulting system brings together features from a wide range of prior work in a unified way.

1998 ACM Subject Classification D.3.3 [Programming Languages] Language Constructs and Features – Classes and Objects

Keywords and phrases Type systems, Capabilities, Traits, Concurrency, Object-Oriented

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Shared mutable state is ubiquitous in object-oriented programming. Sharing can be more

Reference Capabilities for Concurrency Control

- A capability grants access to some object

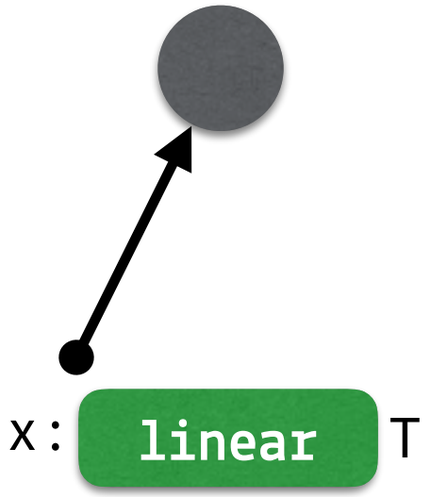
reference

- The type of a capability defines the interface to its object

- A capability assumes exclusive access

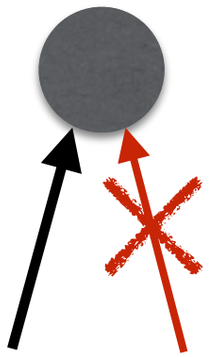
  Thread-safety ⇒ No data-races

- How thread-safety is achieved is controlled by the capability's *mode*

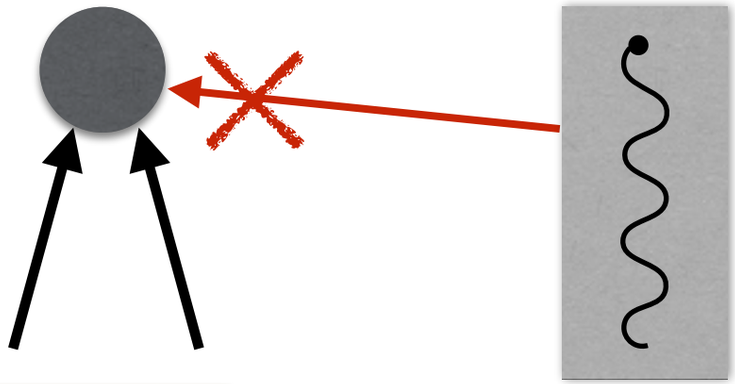


Reference Capabilities á la Mode

- *Exclusive* modes

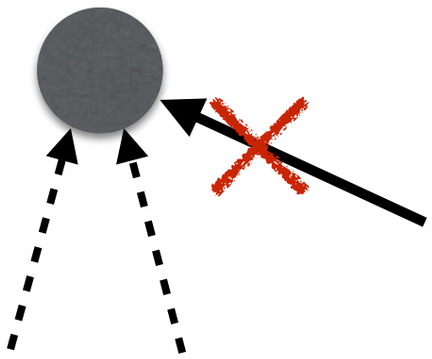


`linear` Globally unique



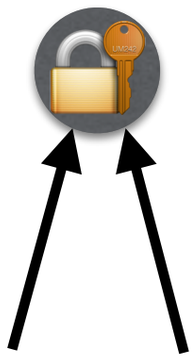
`local` Thread-local

- *Sharable* modes



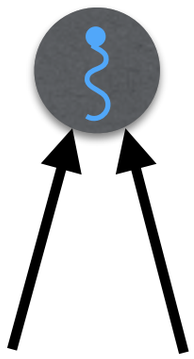
`read`

Precludes mutating aliases



`locked`

Implicit locking



`active`

Asynchronous actor

Reference Capabilities á la Mode

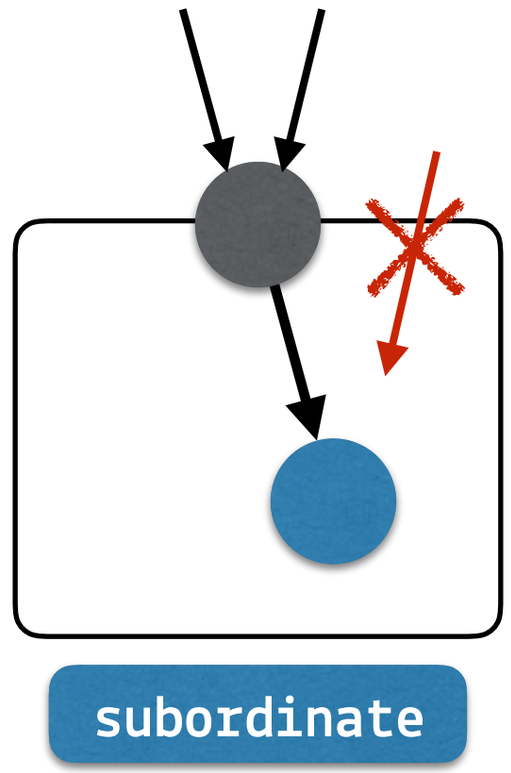
`linear`

`local`

`locked`

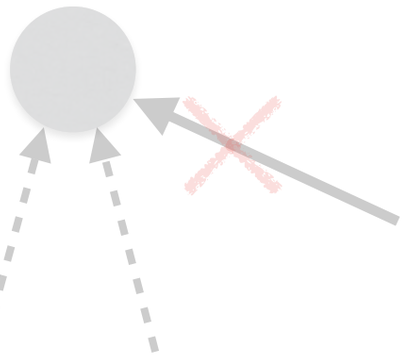
`active`

Guarantee
mutual exclusion



`subordinate`

Encapsulated

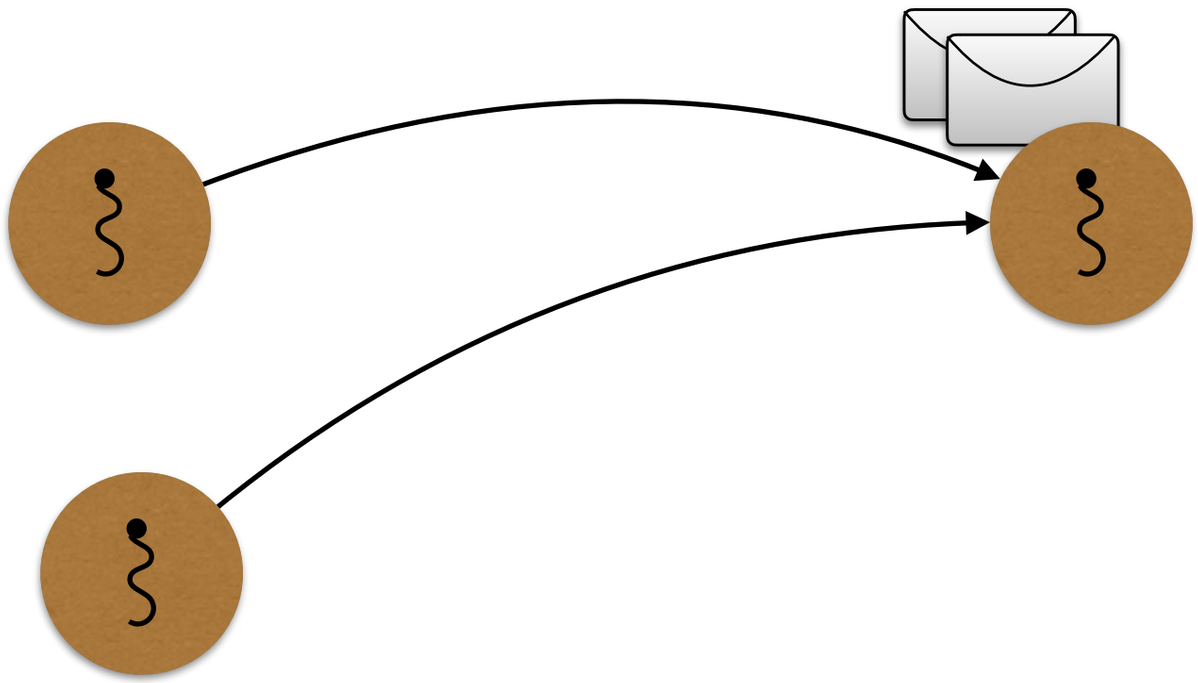


`read`

Precludes mutating
aliases

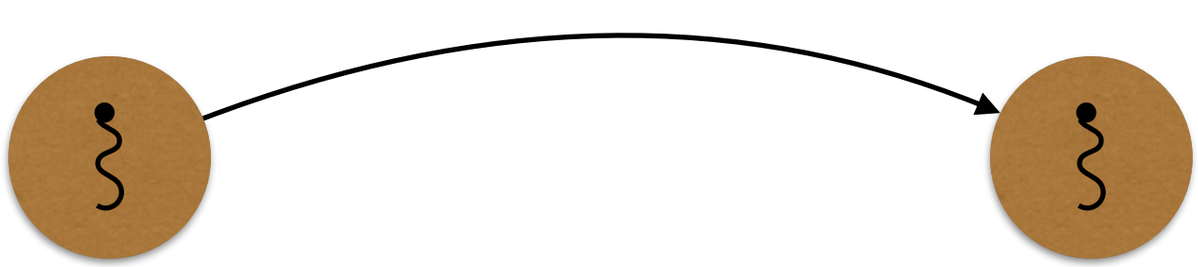
Encore

- Actor-based programming language



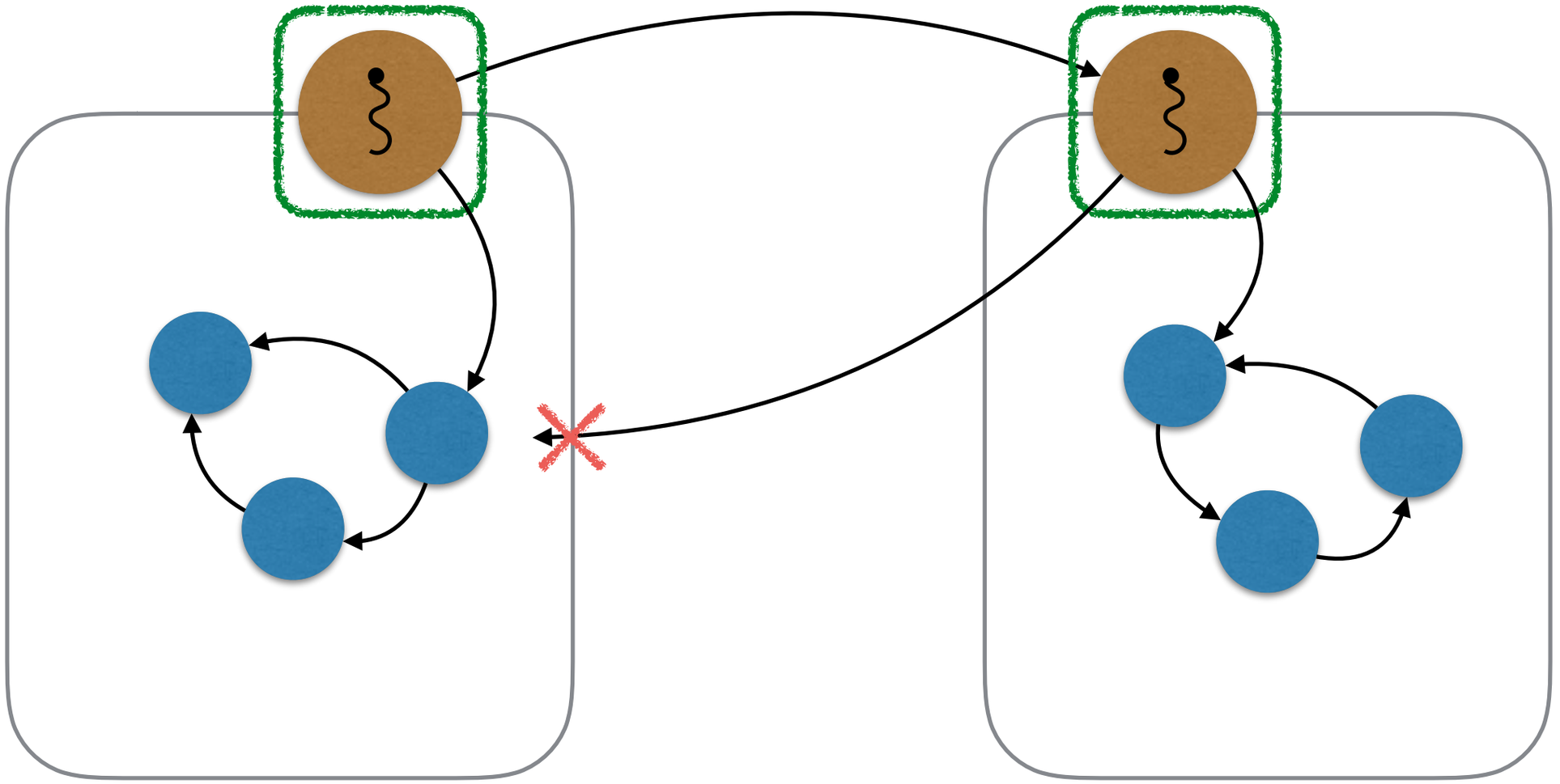
Encore

- Actor-based programming language



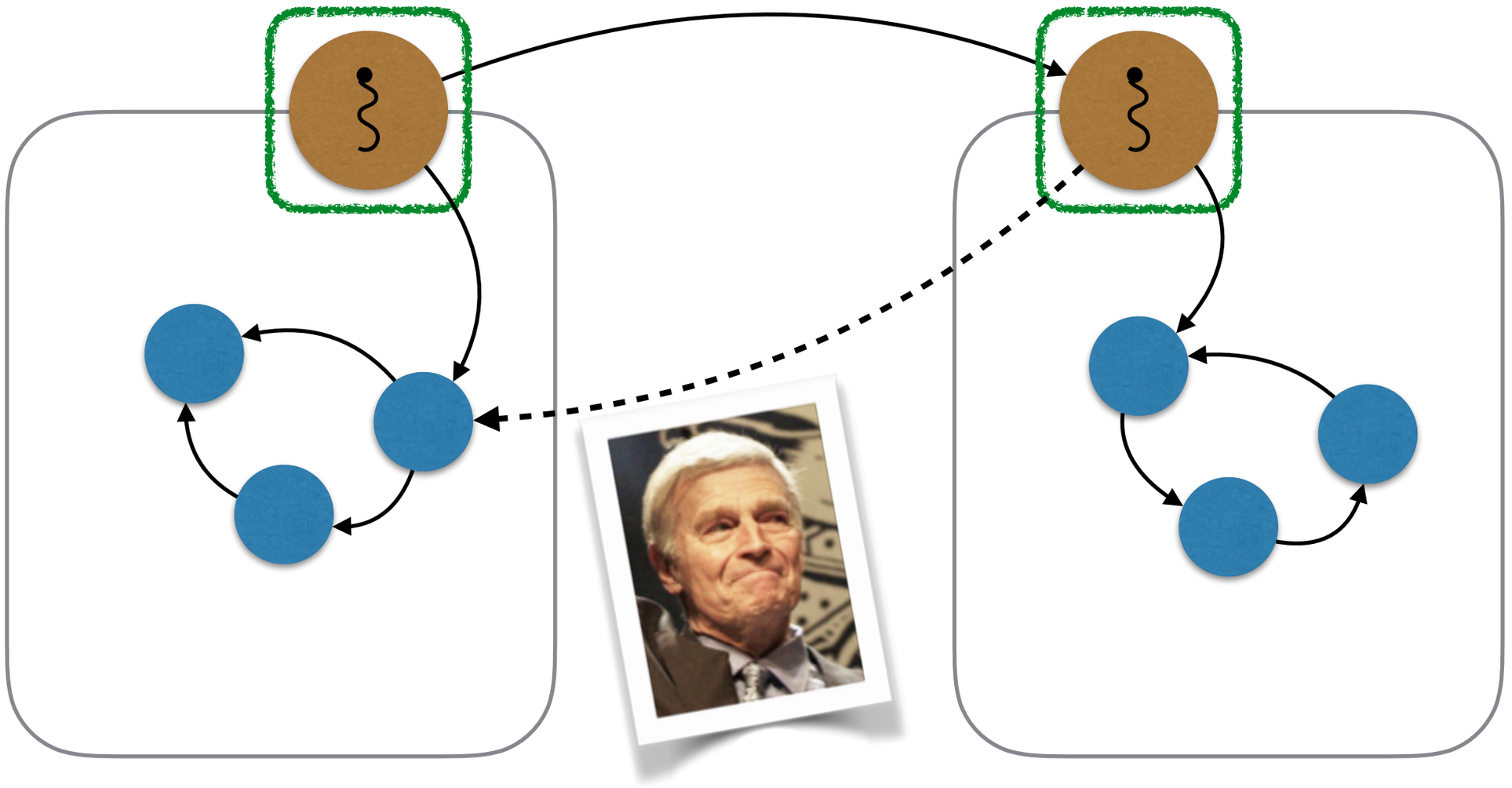
Encore

- Actor-based programming language
- Uses reference capabilities to enforce safe sharing between actors

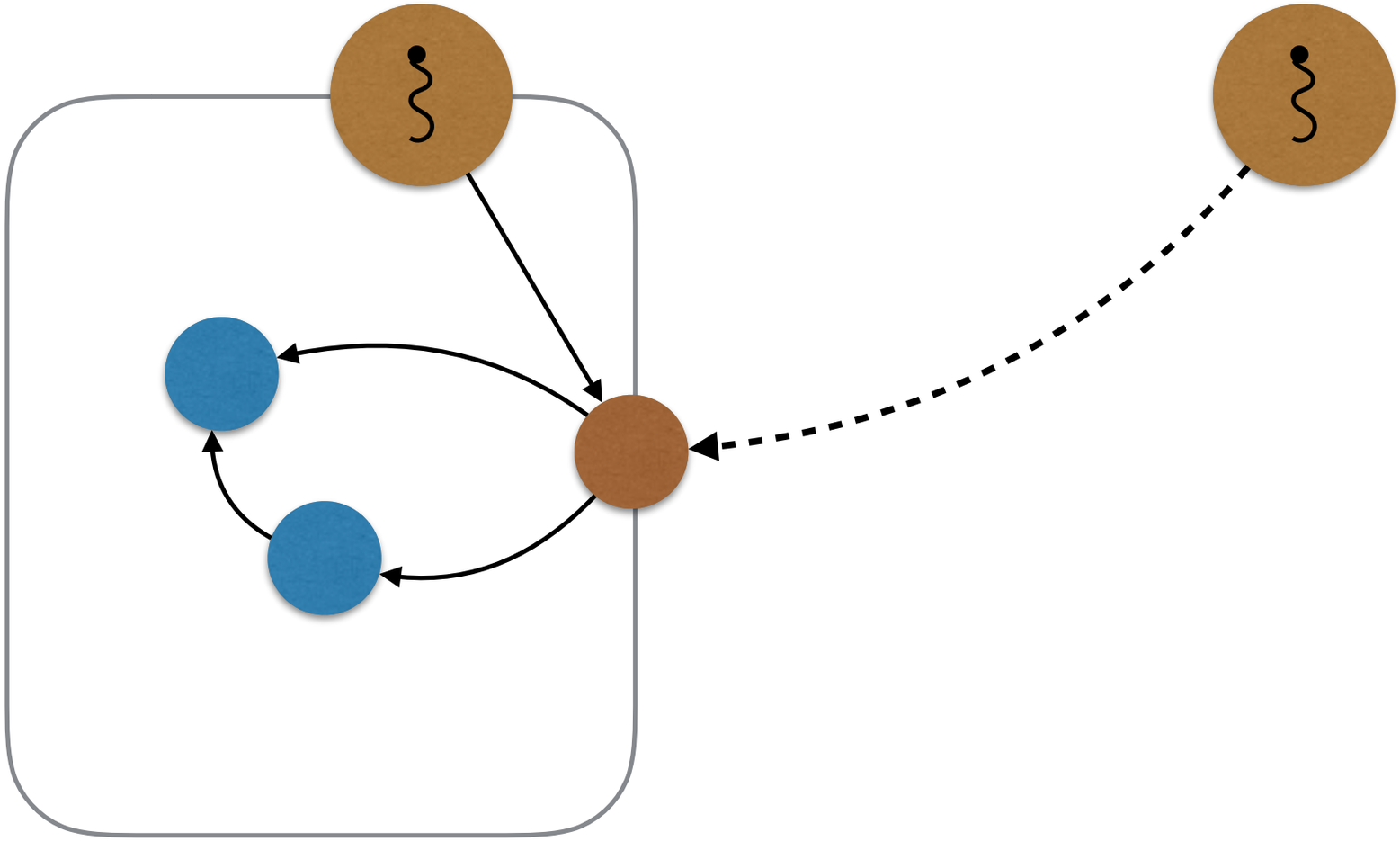


Encore

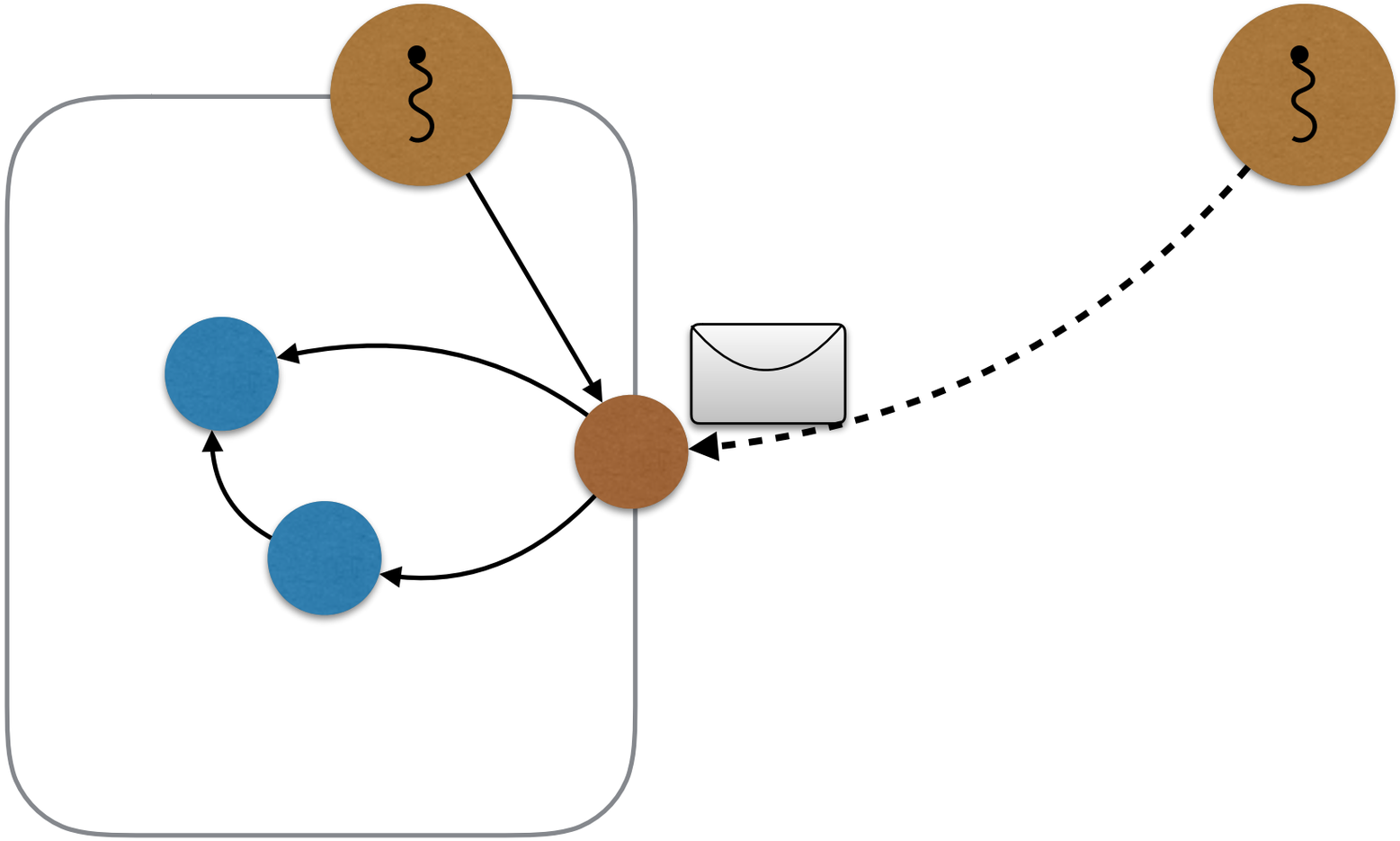
- Actor-based programming language
- Uses reference capabilities to enforce safe sharing between actors



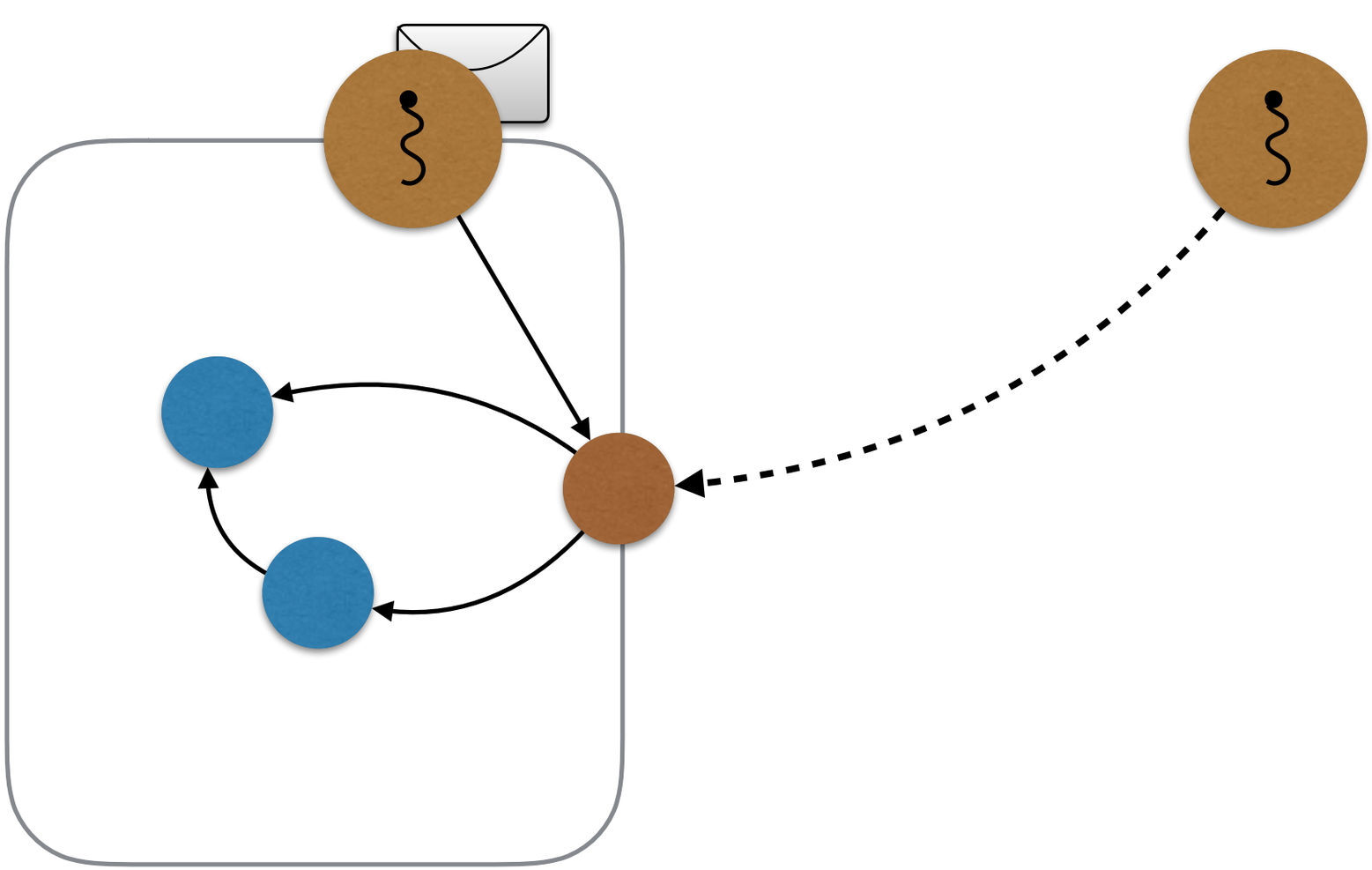
Switching to a Delegating Model



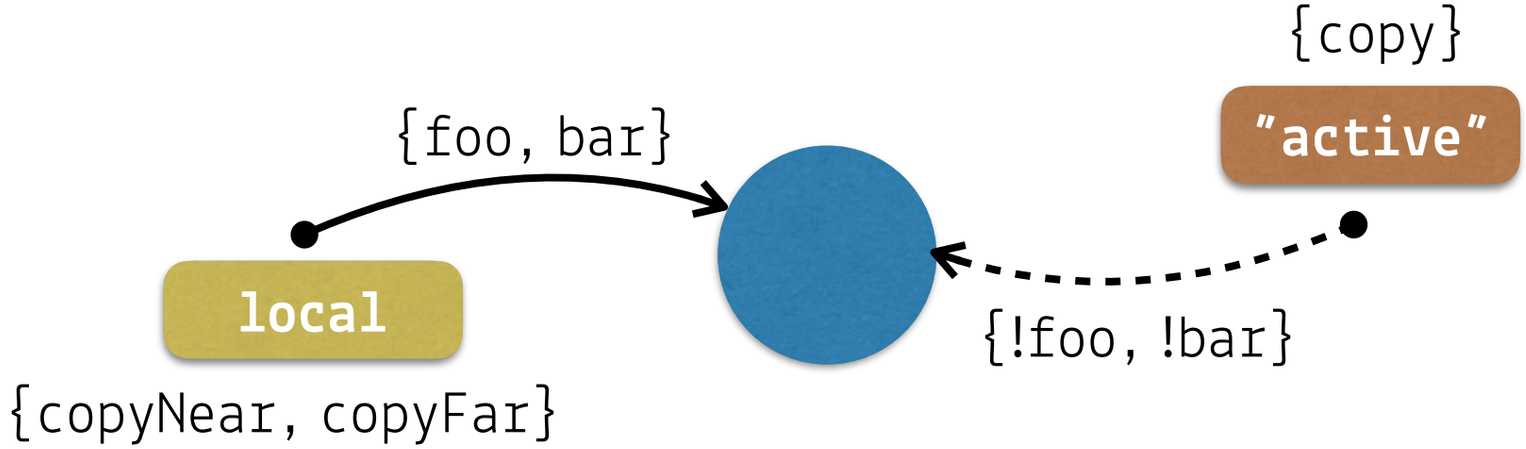
Switching to a Delegating Model



Switching to a Delegating Model

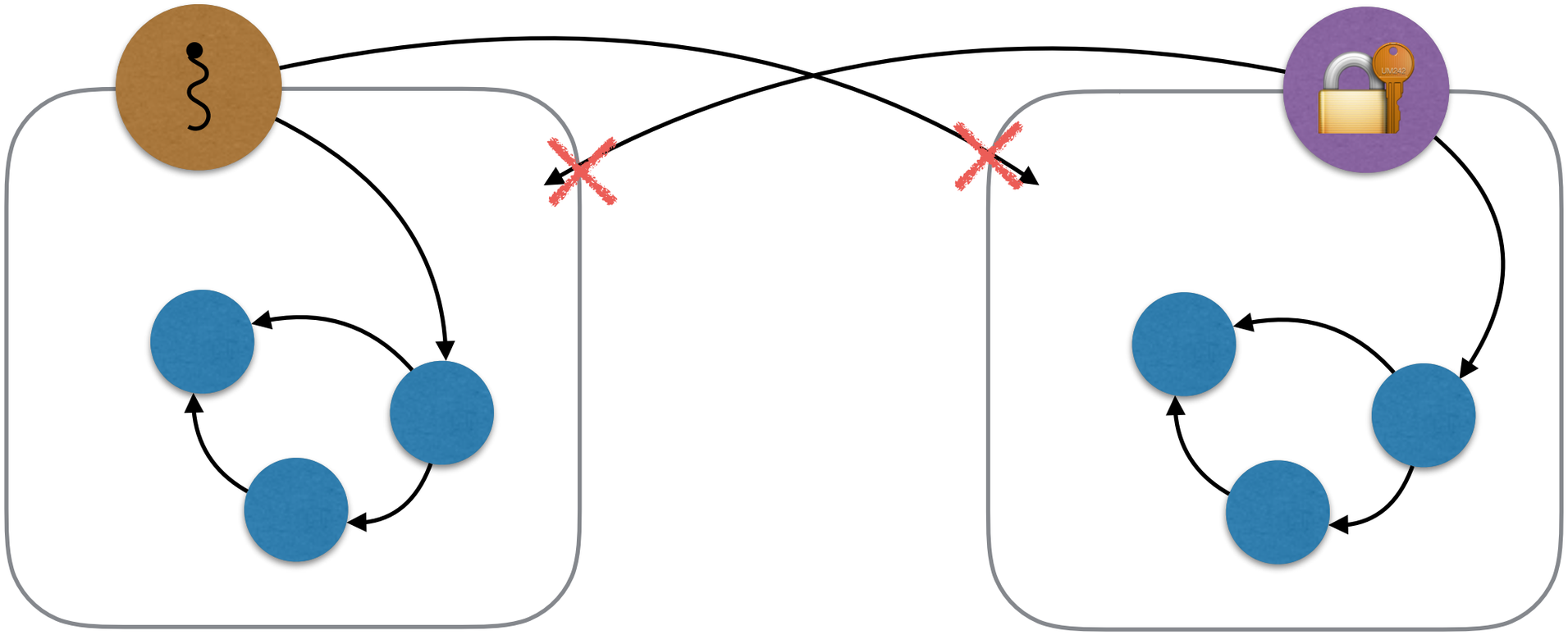


Far References with Reference Capabilities



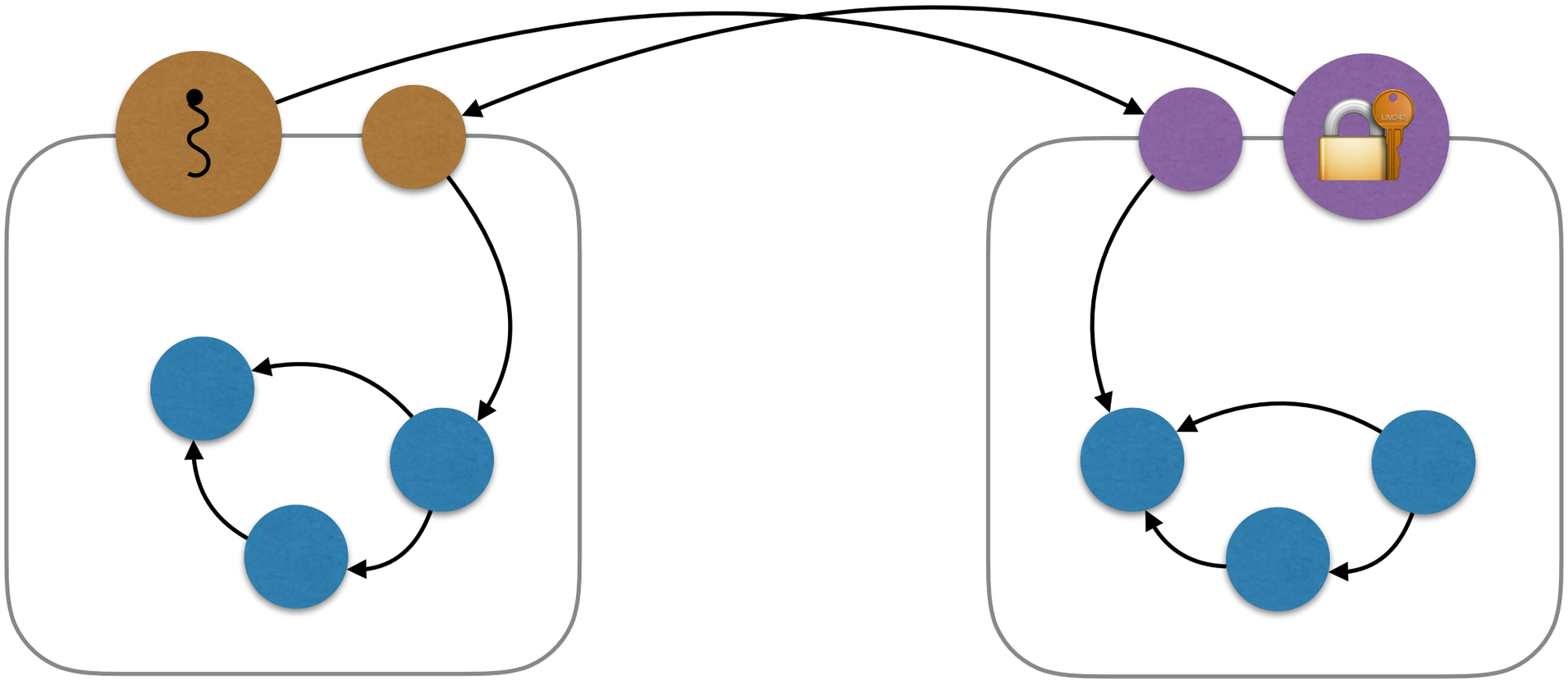
Taking Far References Further

- Safely breaking encapsulation

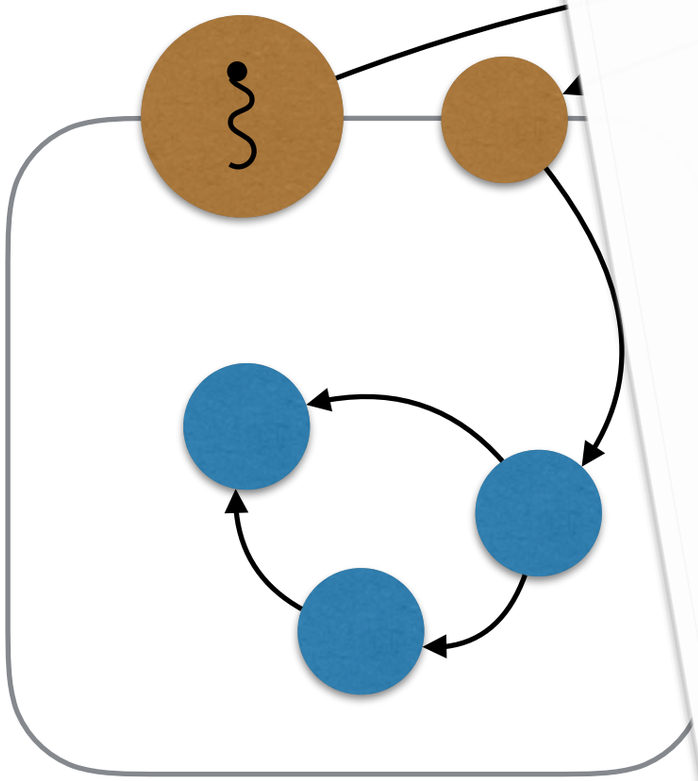


Taking Far References Further

- Safely breaking encapsulation



Actors without Borders [PLACES'17]



Actors without Borders: Amnesty for Imprisoned State

Elias Castegren Tobias Wrigstad
Uppsala University, Sweden

In concurrent systems, some form of synchronisation is typically needed to achieve data-race freedom, which is important for correctness and safety. In actor-based systems, messages are exchanged concurrently but executed sequentially by the receiving actor. By relying on isolation and non-sharing, an actor can access its own state without fear of data-races, and the internal behavior of an actor can be reasoned about sequentially.

However, actor isolation is sometimes too strong to express useful patterns. For example, letting the iterator of a data-collection alias the internal structure of the collection allows a more efficient implementation than if each access requires going through the interface of the collection. With full isolation, in order to maintain sequential reasoning the iterator must be made part of the collection, which bloats the interface of the collection and means that a client must have access to the whole data-collection in order to use the iterator.

In this paper, we propose a programming language construct that enables a relaxation of isolation but without sacrificing sequential reasoning. We formalise the mechanism in a simple lambda calculus with actors and passive objects, and show how an actor may leak parts of its internal state while ensuring that any interaction with this data is still synchronised.

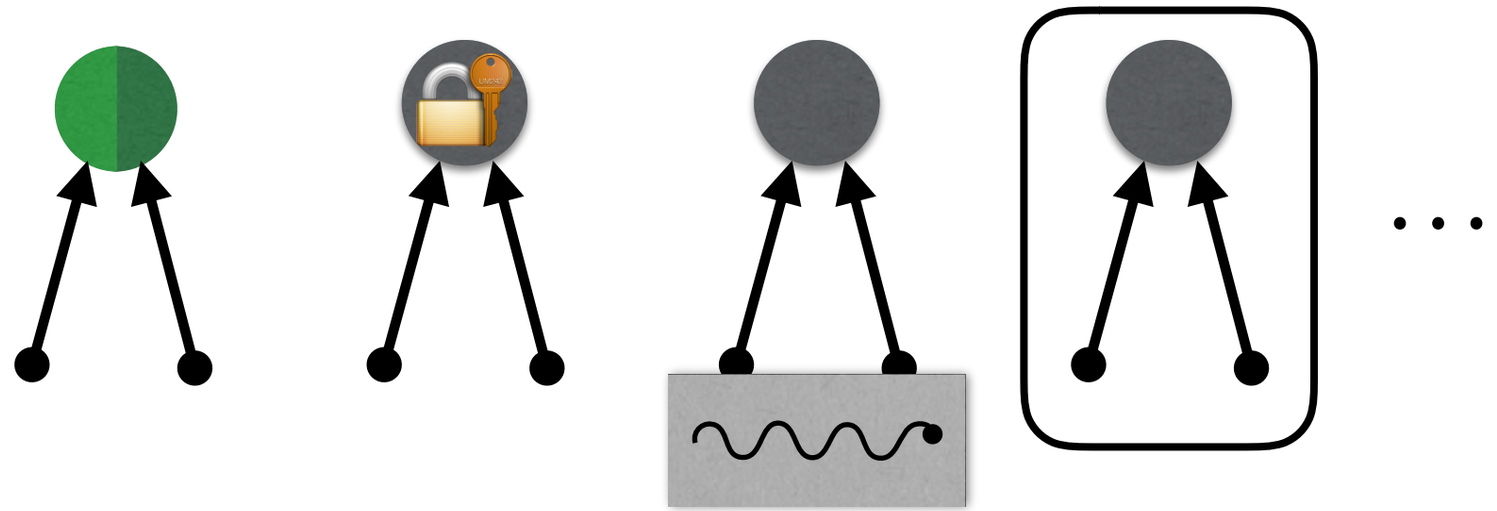
1 Introduction

Synchronisation is a key aspect of concurrent programs and different concurrency models ha
ization differently. Pessimistic models, like locks or the actor model [1] serialise co
regulated units, allowing sequential reasoning about internal beh
ity including also active objects, if a reference
inside of A are subject to dat
gate object

Safe Object Sharing

- Any two aliases are e.g. composable, synchronised, thread-local, encapsulated…

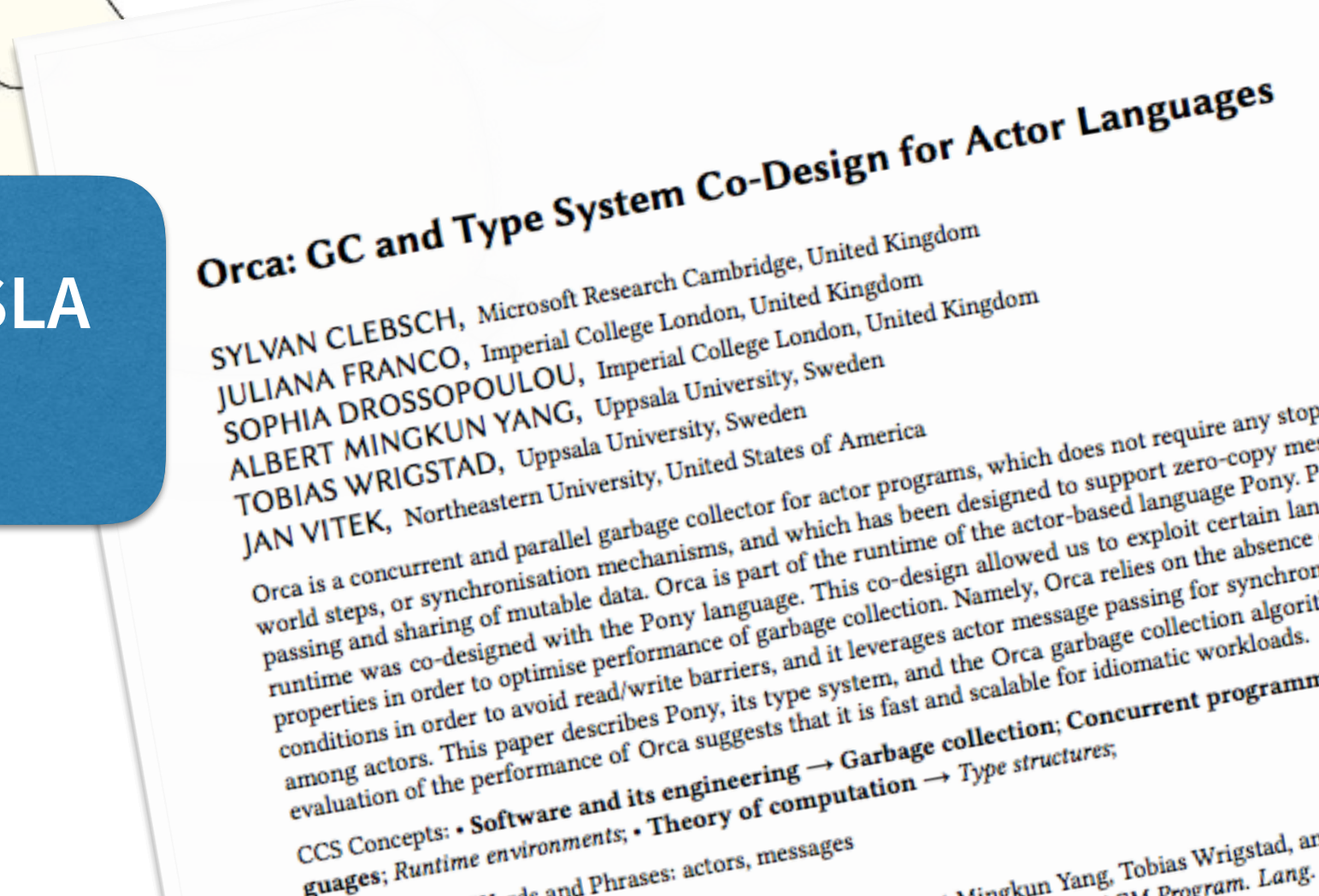
  ⟹ No data-races



- **Whoever controls the aliasing controls the concurrency!**

Pony [AGERE ’15] and Orca [OOPSLA’17]

Thursday@OOPSLA
11:15-11:37



Orca: GC and Type System Co-Design for Actor Languages

SYLVAN CLEBSCH, Microsoft Research Cambridge, United Kingdom
JULIANA FRANCO, Imperial College London, United Kingdom
SOPHIA DROSSOPOULOU, Imperial College London, United Kingdom
ALBERT MINGKUN YANG, Uppsala University, Sweden
TOBIAS WRIGSTAD, Uppsala University, Sweden
JAN VITEK, Northeastern University, United States of America

Orca is a concurrent and parallel garbage collector for actor programs, which does not require any stop
world steps, or synchronisation mechanisms, and which has been designed to support zero-copy me
passing and sharing of mutable data. Orca is part of the runtime of the actor-based language Pony. P
runtime was co-designed with the Pony language. This co-design allowed us to exploit certain lan
properties in order to optimise performance of garbage collection. Namely, Orca relies on the absence
conditions in order to avoid read/write barriers, and it leverages actor message passing for synchron
among actors. This paper describes Pony, its type system, and the Orca garbage collection algorit
evaluation of the performance of Orca suggests that it is fast and scalable for idiomatic workloads.

CCS Concepts: • **Software and its engineering → Garbage collection**; **Concurrent programm**
guages; *Runtime environments*; • **Theory of computation** → *Type structures*;

ds and Phrases: actors, messages

Mingkun Yang, Tobias Wrigstad, an

Program. Lang.

Recap

- Object capabilities specify the permitted operations on an **object**
- Reference capabilities specify the permitted operations on a **reference**
- Strictly following the reference capability model gives **global guarantees**:
  - Encapsulation
  - Thread-locality
  - Uniqueness
  - Absence of mutable aliases
  - …

Refcap > Ocap? Refcap ⇒ Ocap? Refcap ⊆ Ocap?

- Reference capabilities can be a *compliment* to object capabilities
- Reference capabilities allow specifying how object capabilities may be further shared
  - Shallow/transitive permission
  - Local/distributed permission

- Can object capabilities ”emulate” reference capabilities?
  - Only partially
- Can a reference capability be revoked?
  - Weak references

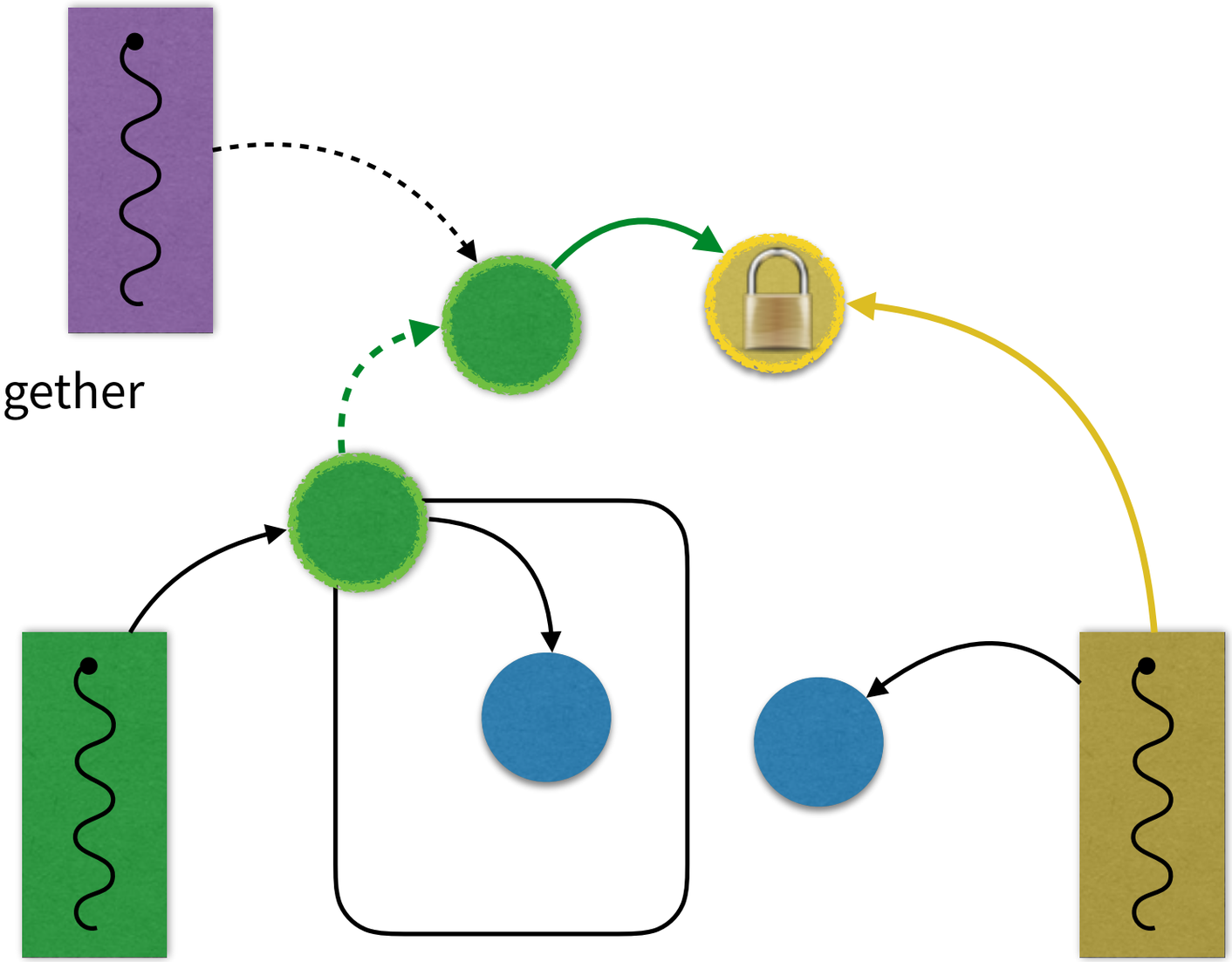
eliasc.github.io

@CartesianGlee

EliasC

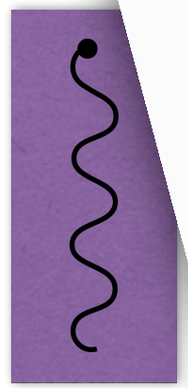
Ensuring Mutual Exclusion

- Relying on locks...
- Relying on isolation…
- Relying on immutability…
- …bans shared mutable state altogether



Ensuring Mutual Exclusion

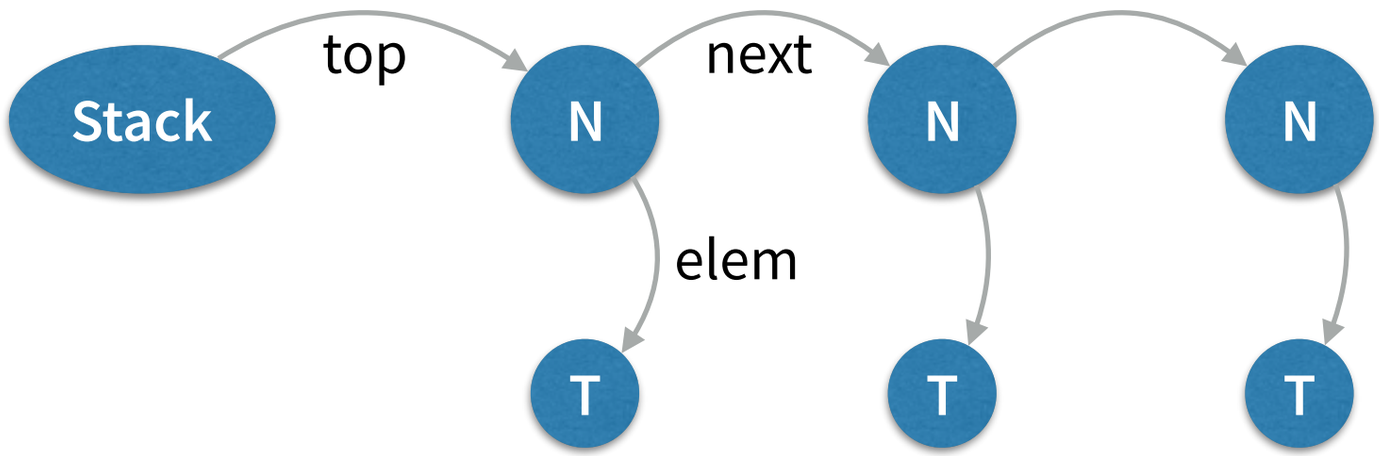
- Relying on locks...
- Relying on isolation…
- Relying on immutability…
- …bans shared mutable state altogether



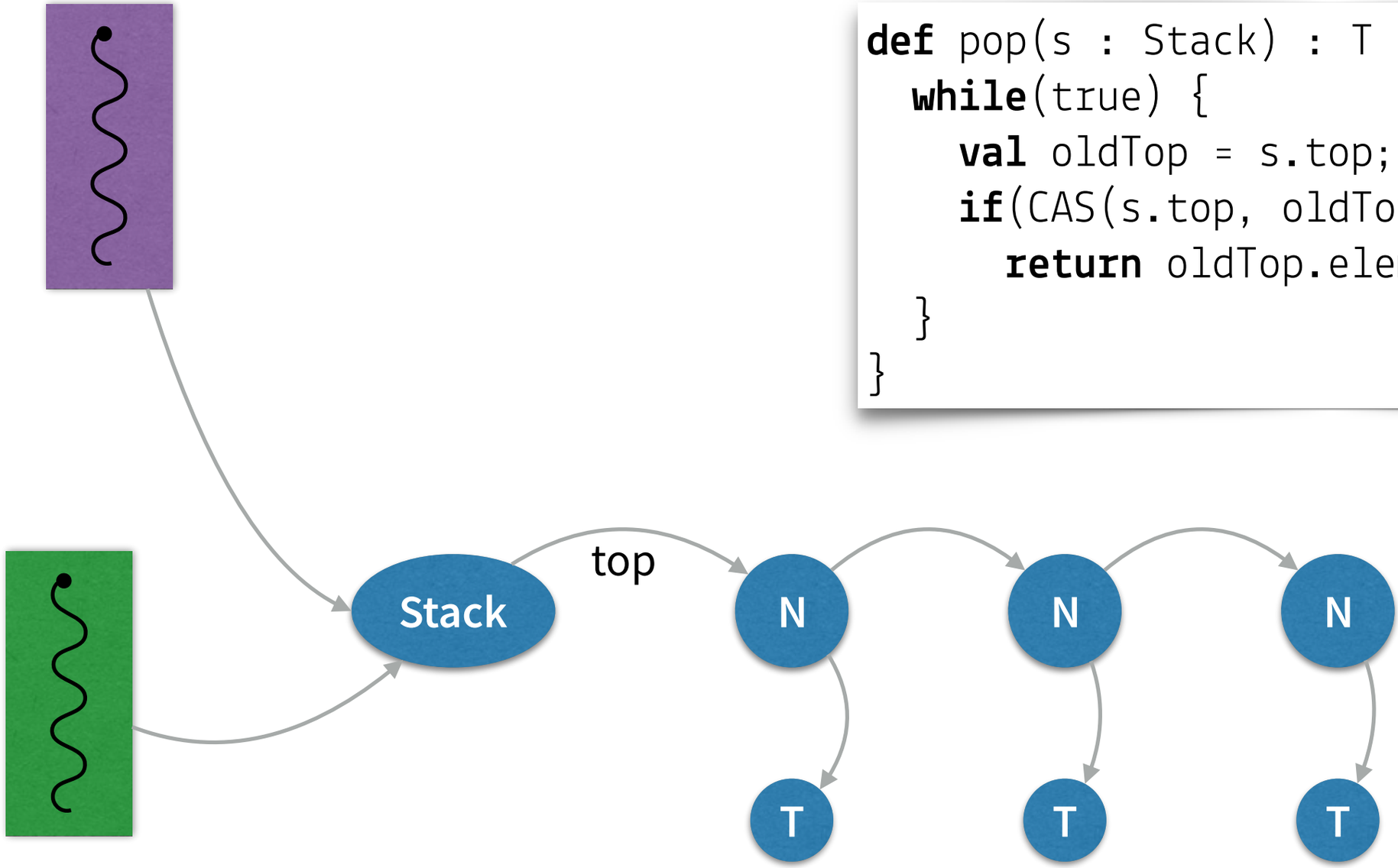
Example: A Treiber Stack

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

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



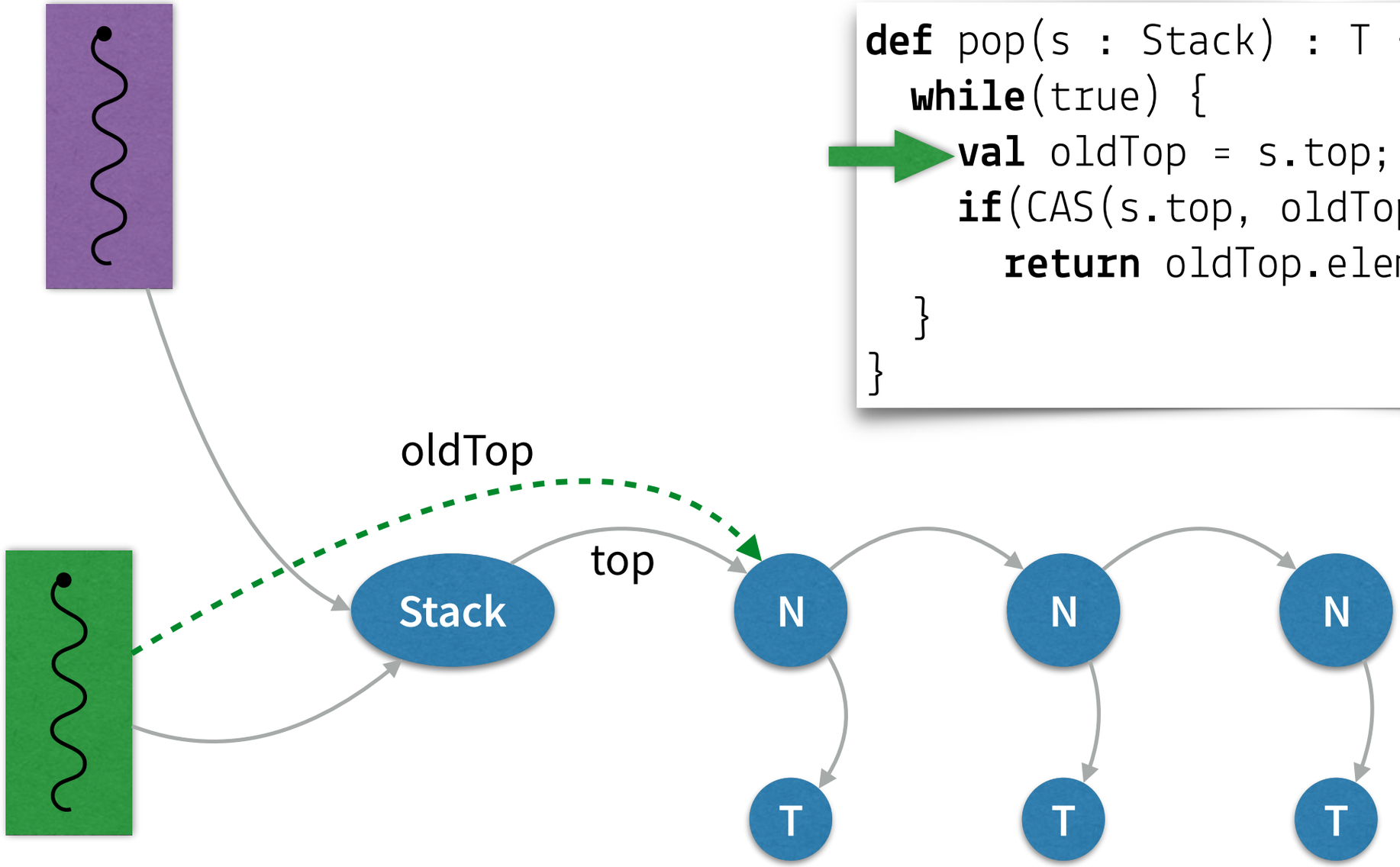
Example: A Treiber Stack



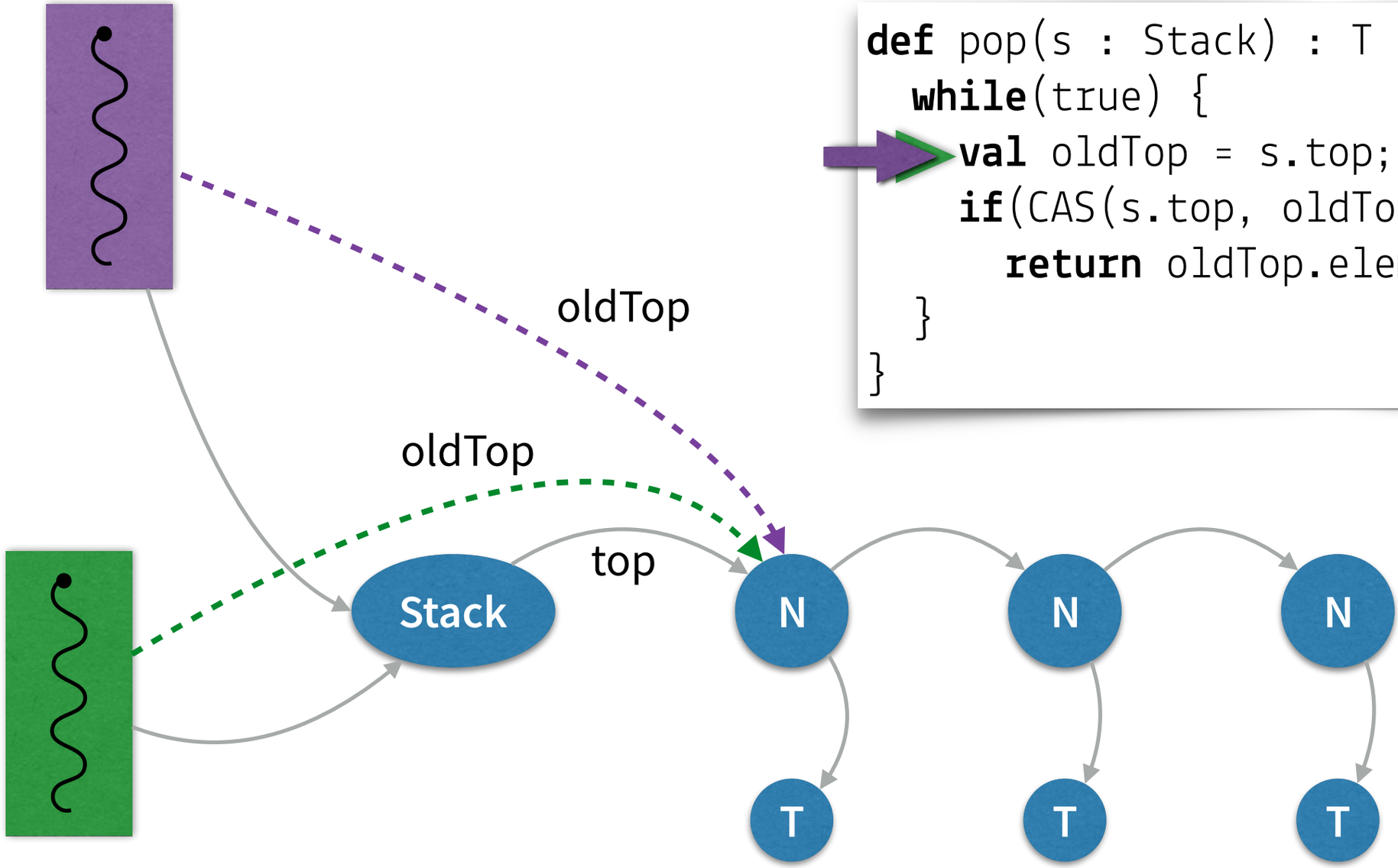

```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAS(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```

Example: A Treiber Stack

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

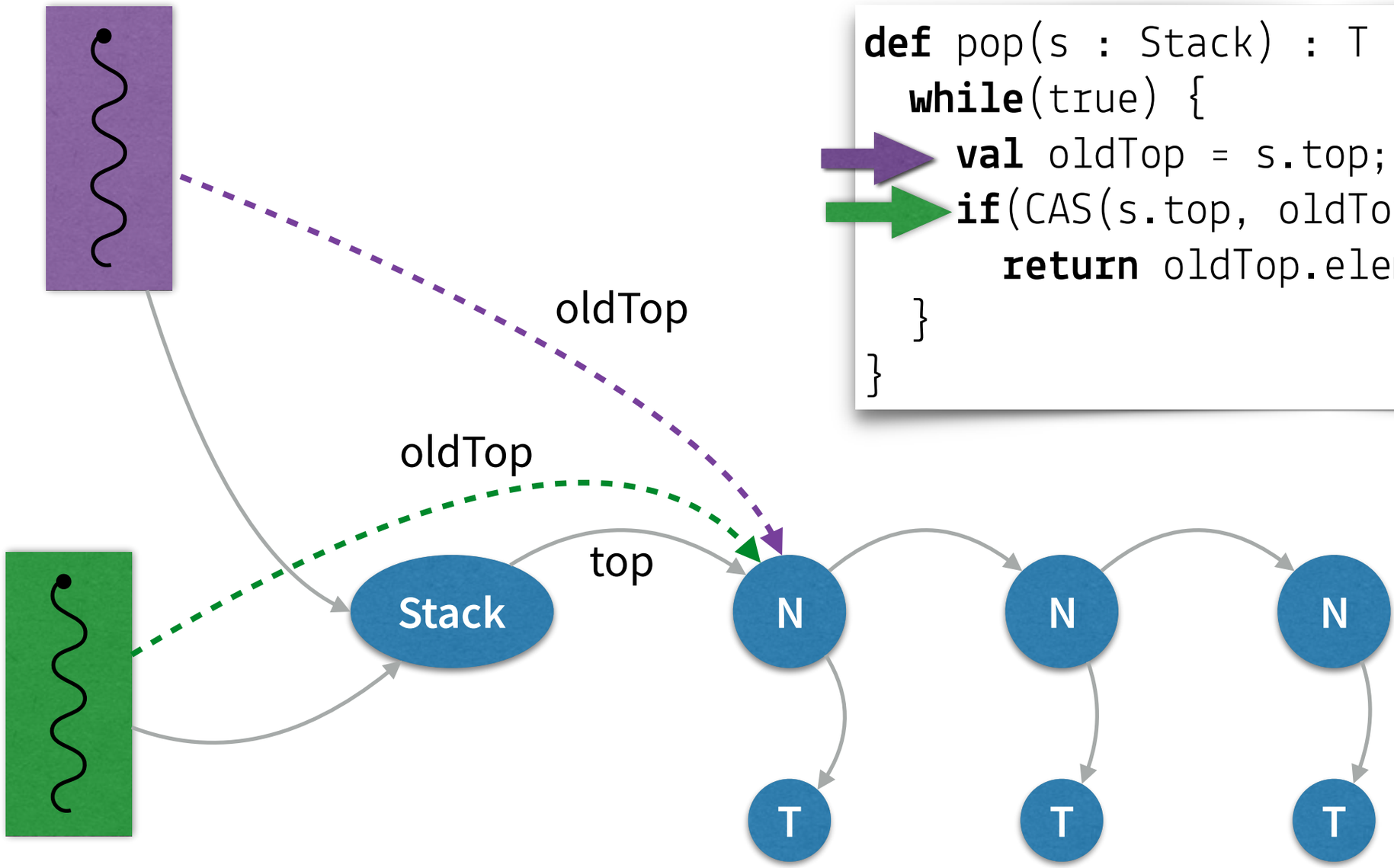


Example: A Treiber Stack



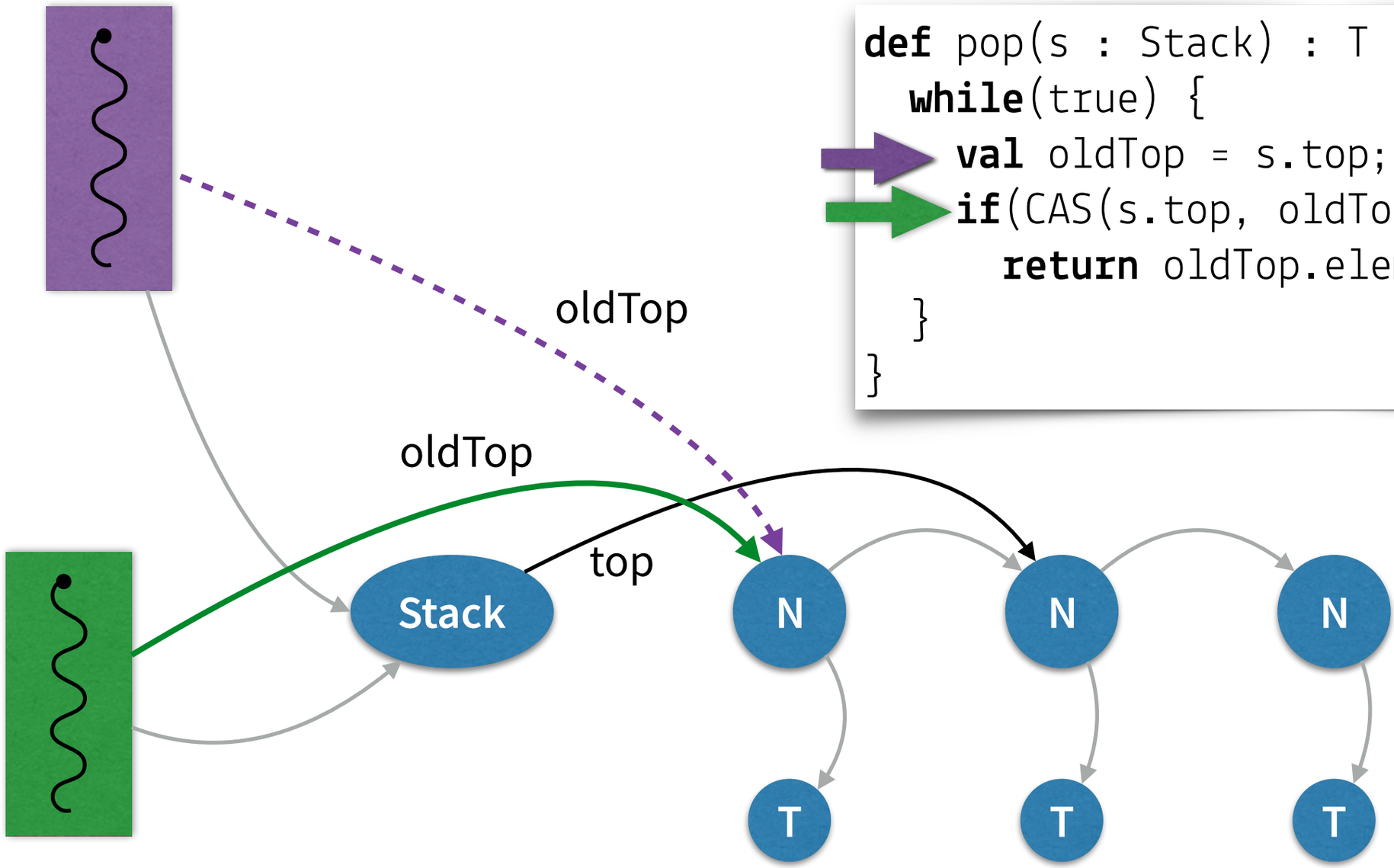

```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAS(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```

Example: A Treiber Stack




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

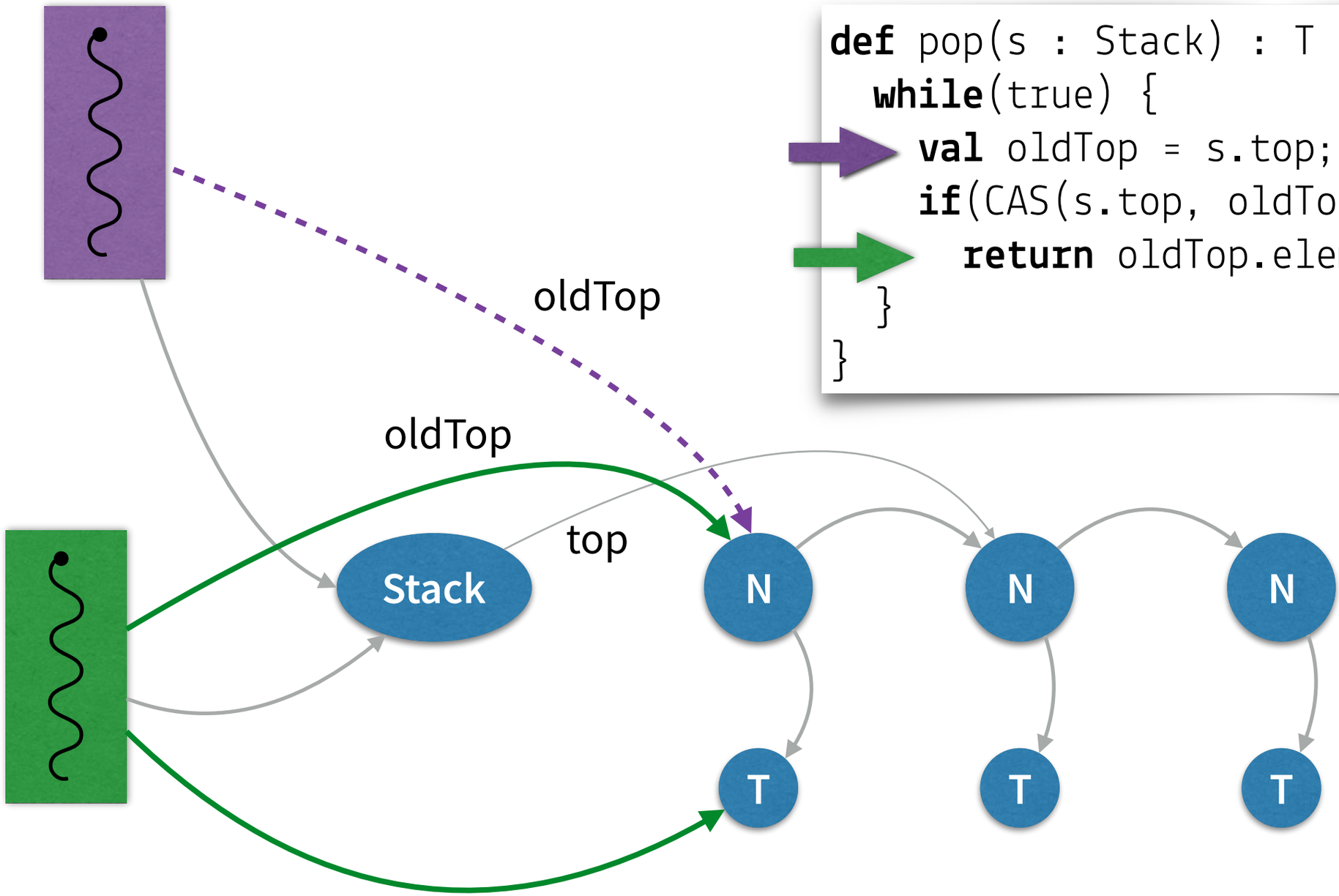
Example: A Treiber Stack




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

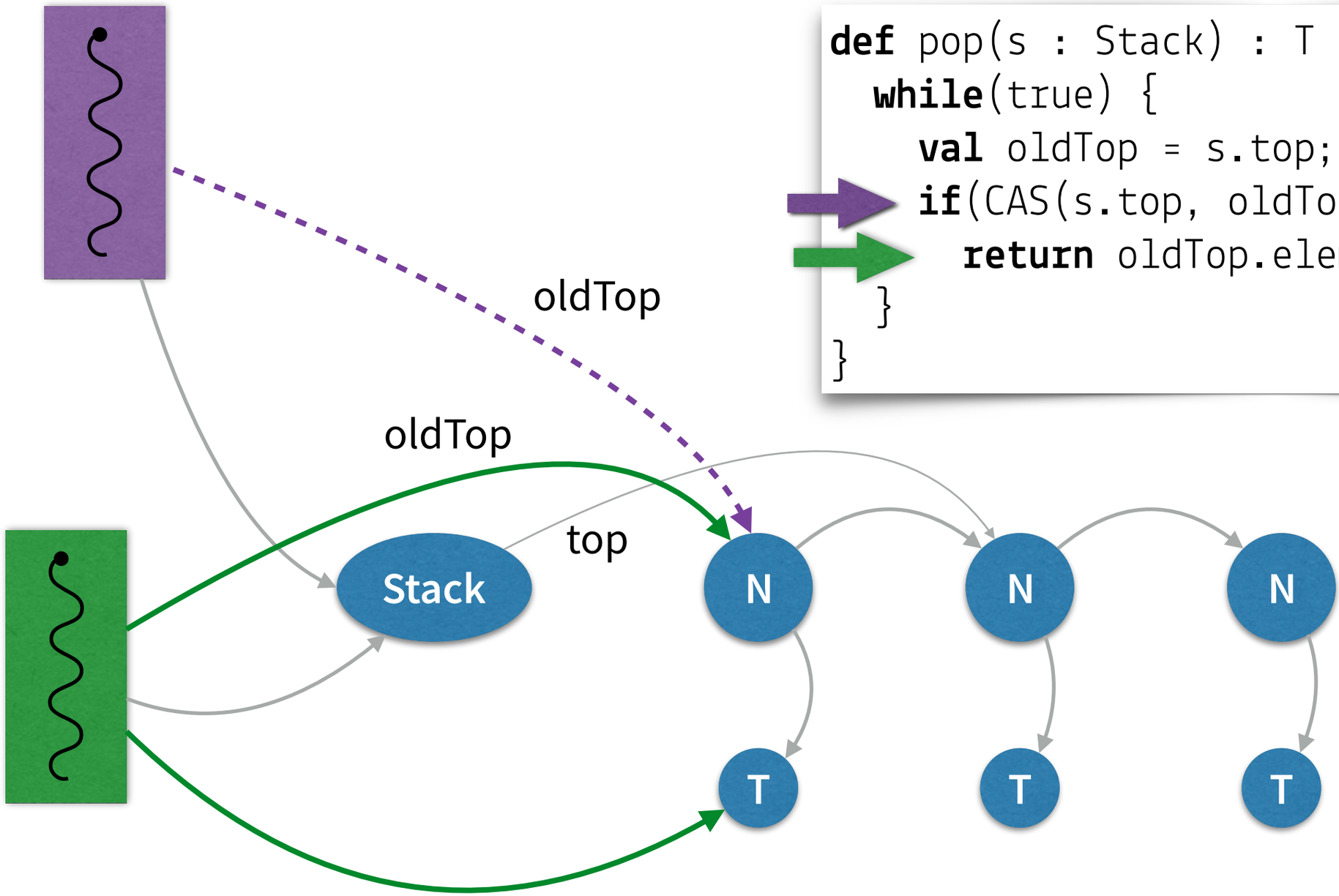
Example: A Treiber Stack

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



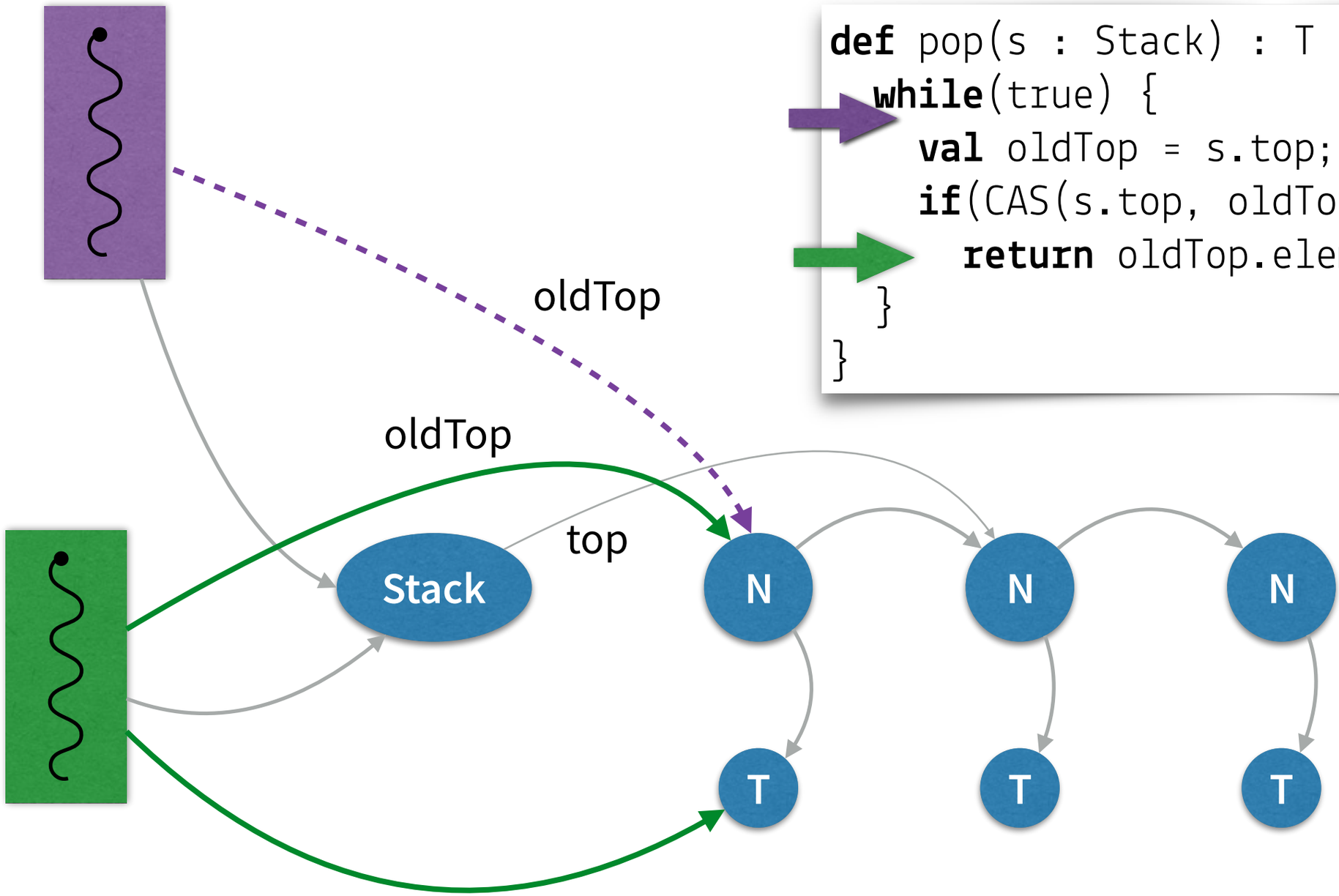
Example: A Treiber Stack

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

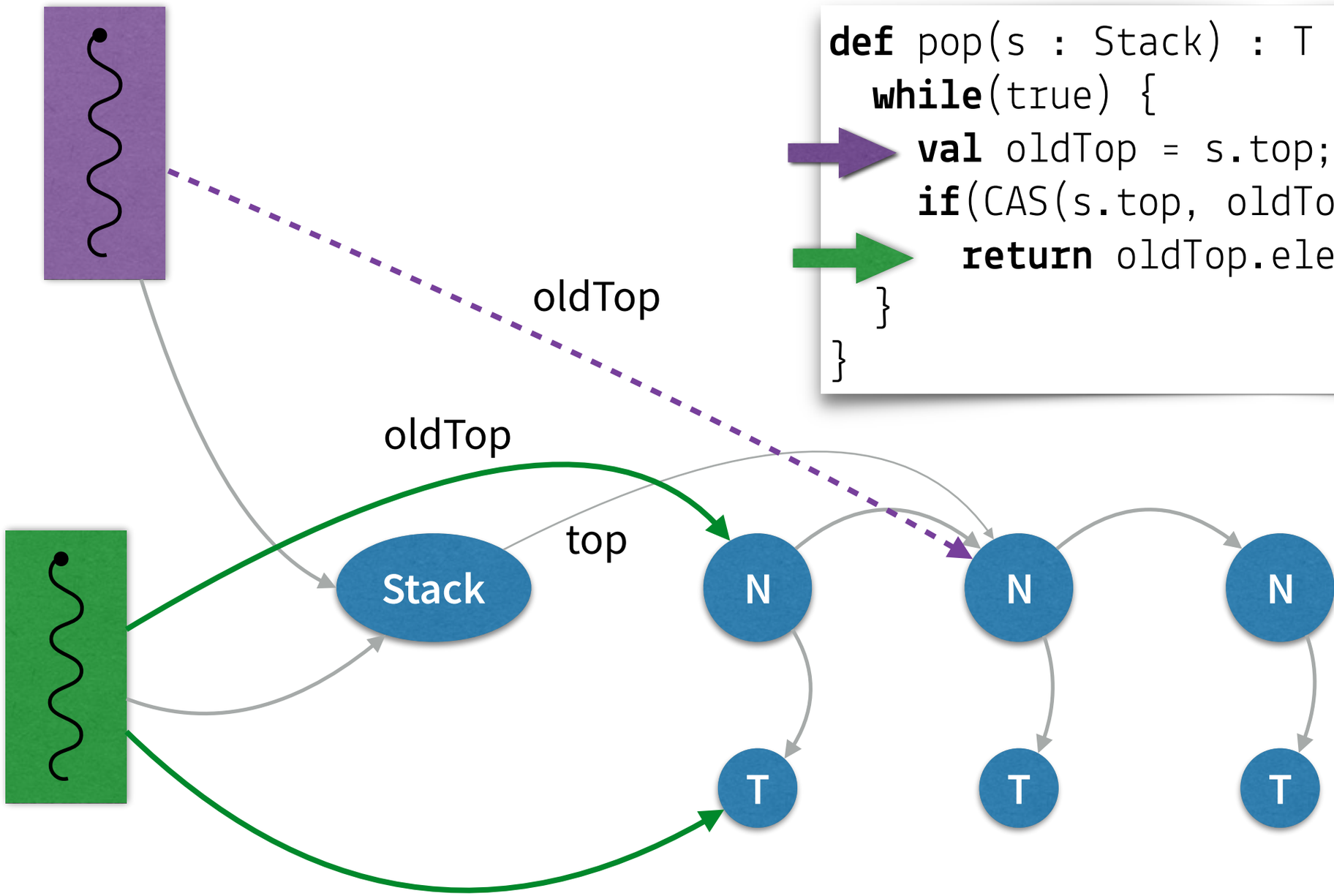


Example: A Treiber Stack

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

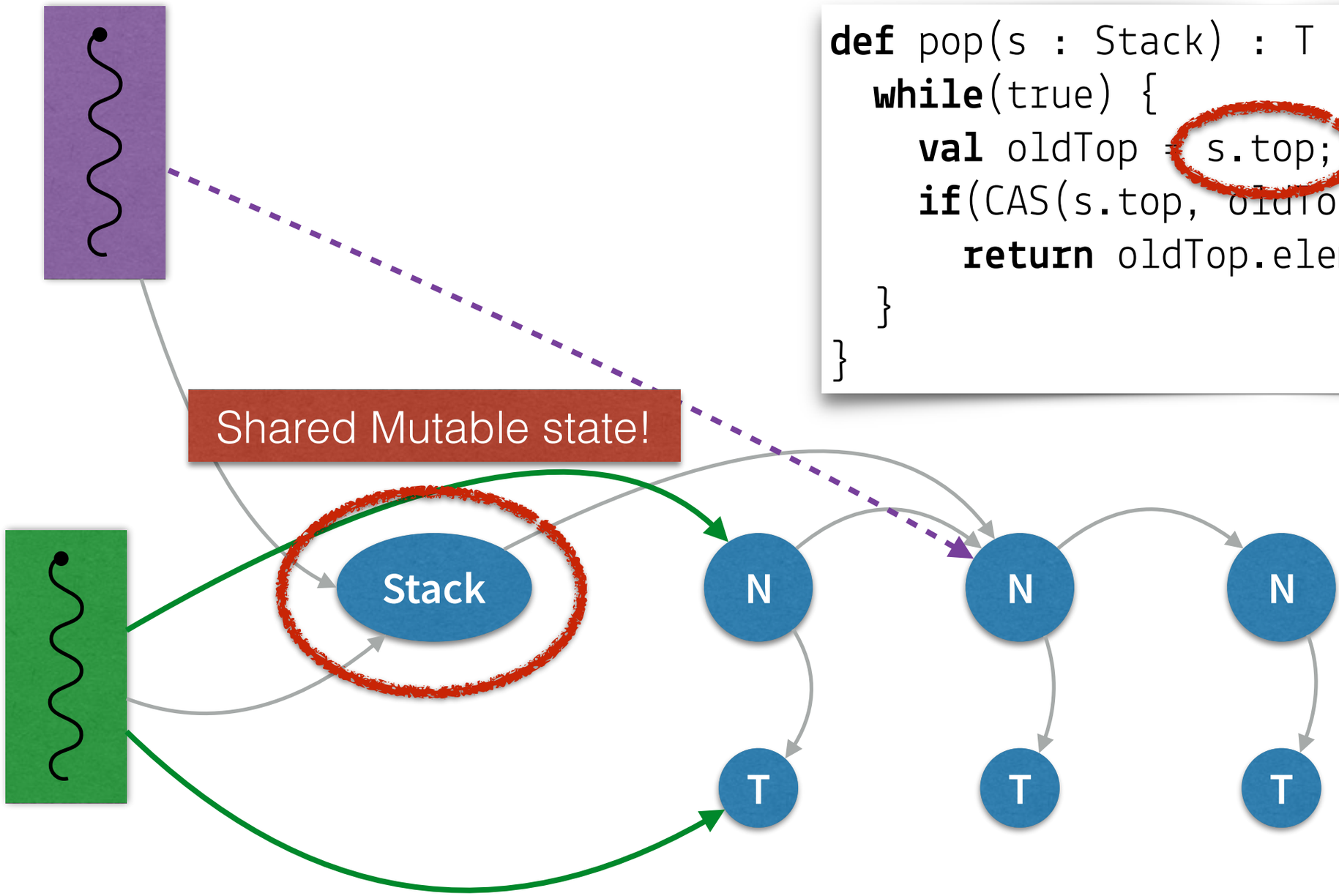


Example: A Treiber Stack



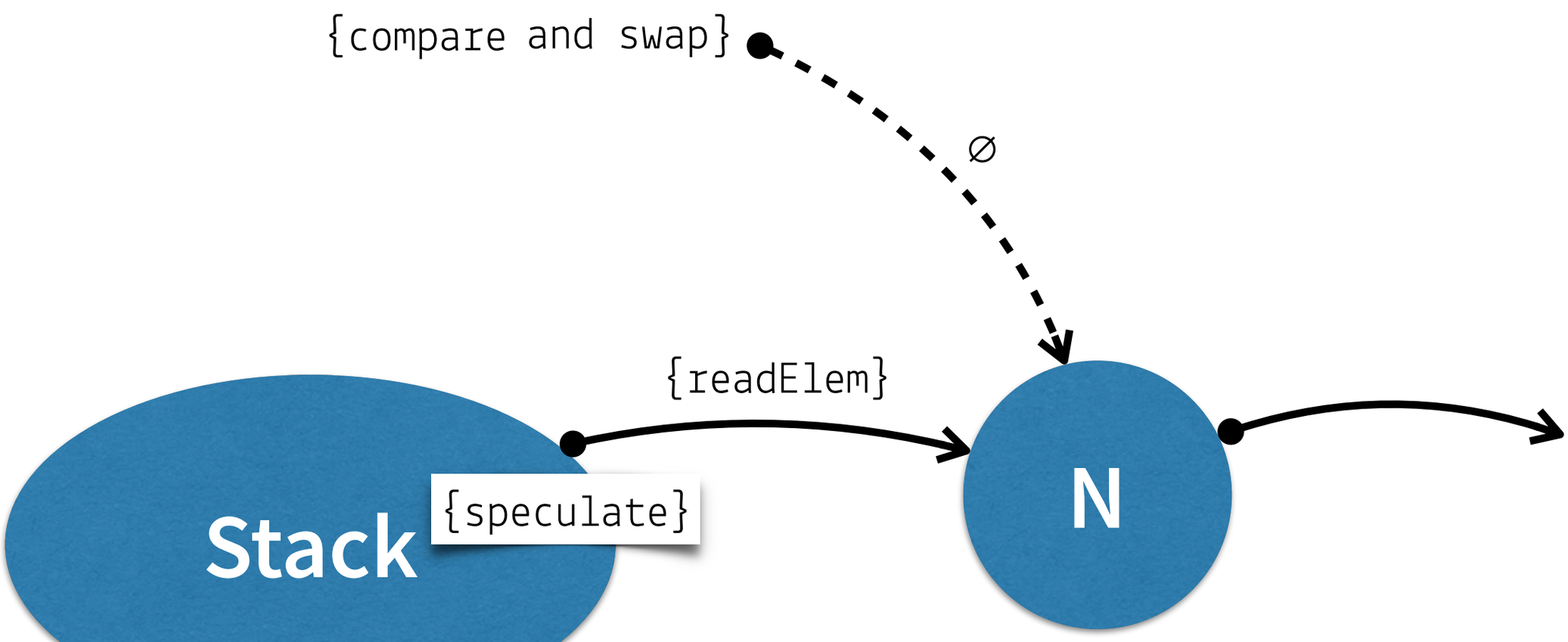

```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAS(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```

Fine-Grained Concurrency ≠ Mutual Exclusion



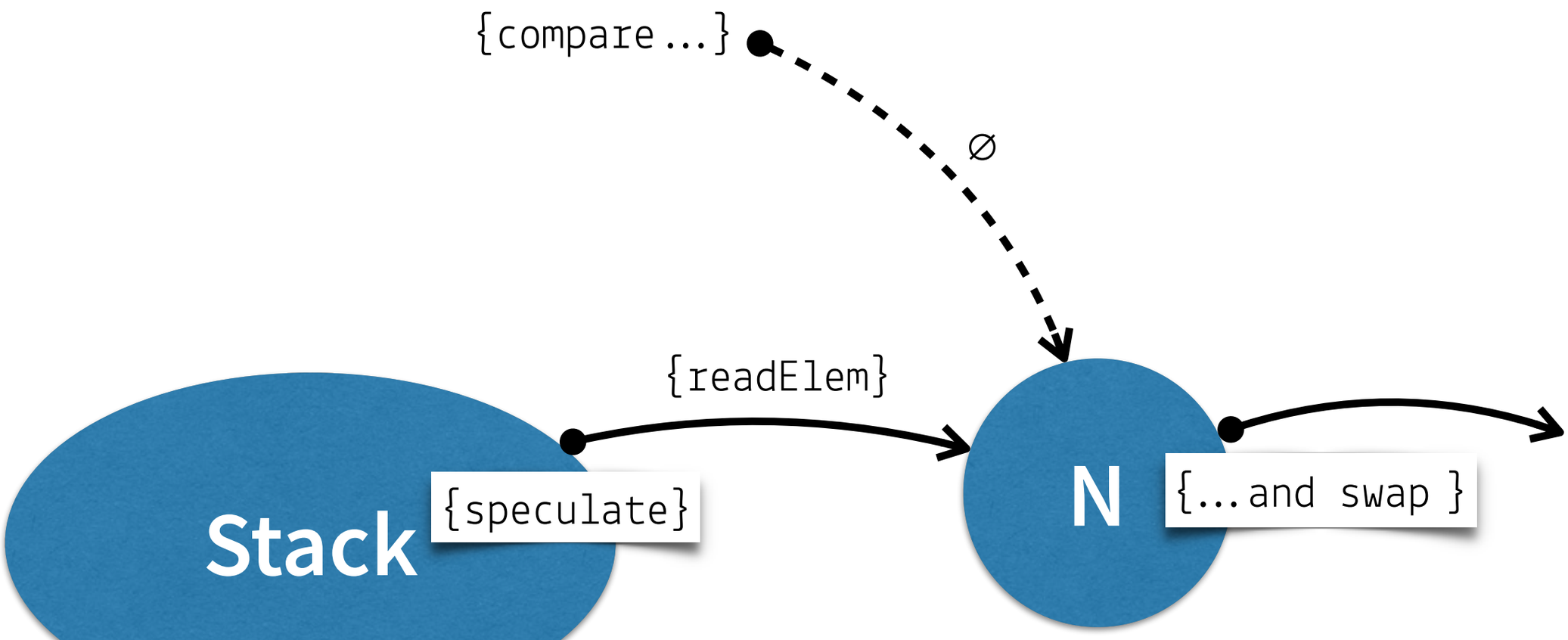

```
def pop(s : Stack) : T {
  while(true) {
    val oldTop = s.top;
    if(CAS(s.top, oldTop, oldTop.next))
      return oldTop.elem;
  }
}
```

Treiber Stack with Reference Capabilities




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

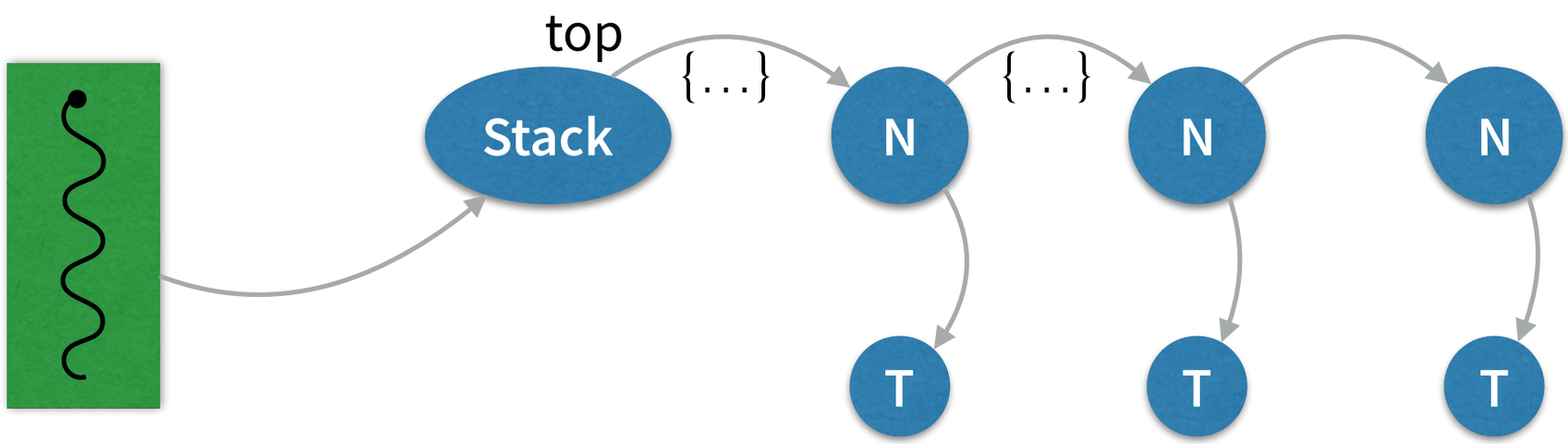
Treiber Stack with Reference Capabilities




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

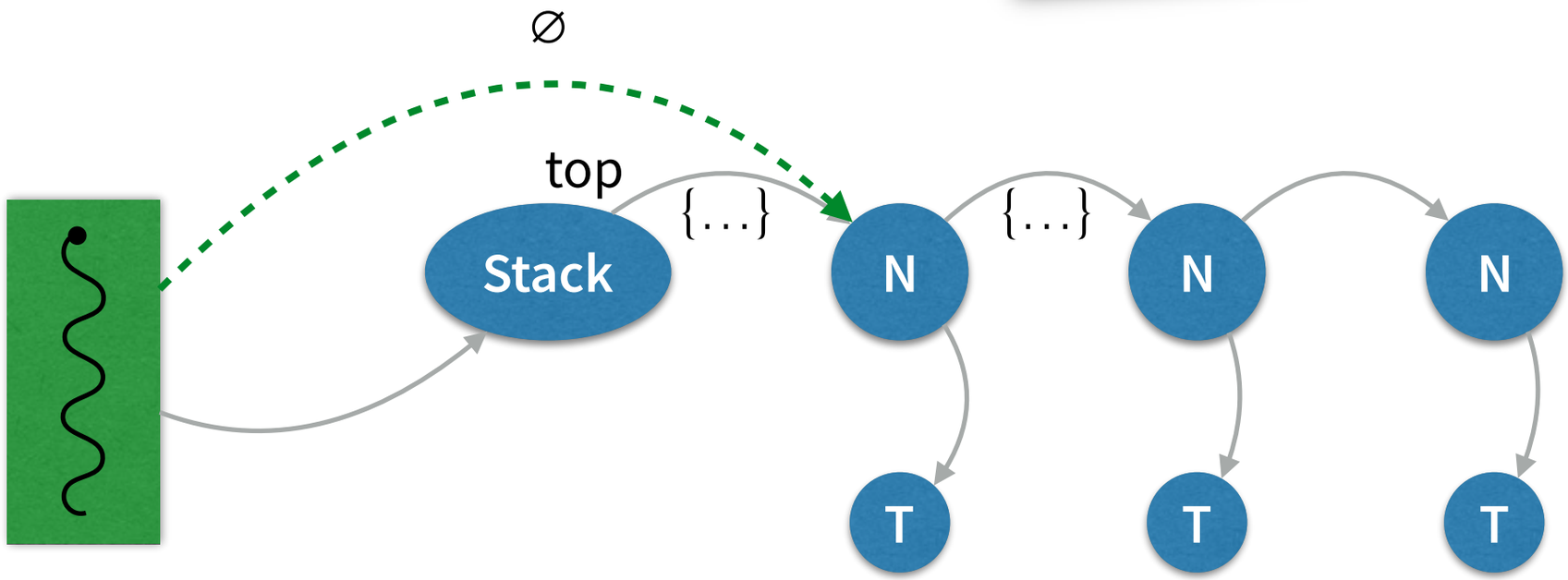
Capability Transfer with Compare-and-Swap

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



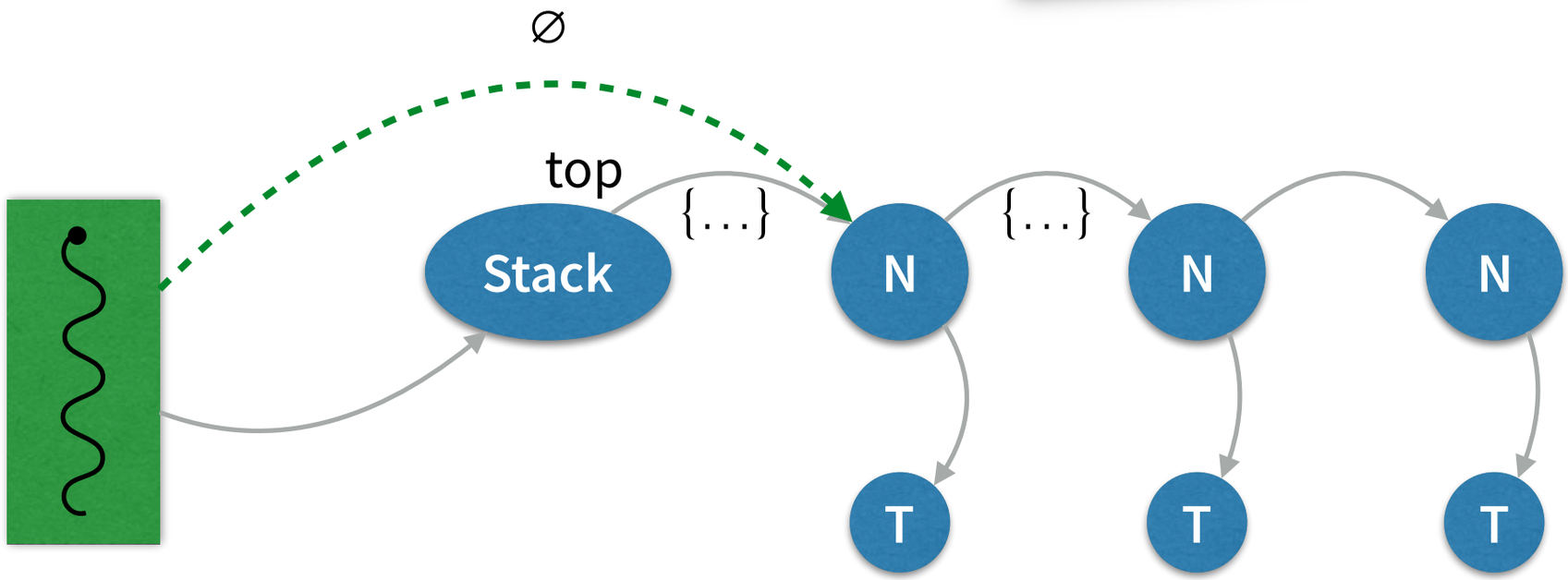
Capability Transfer with Compare-and-Swap

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



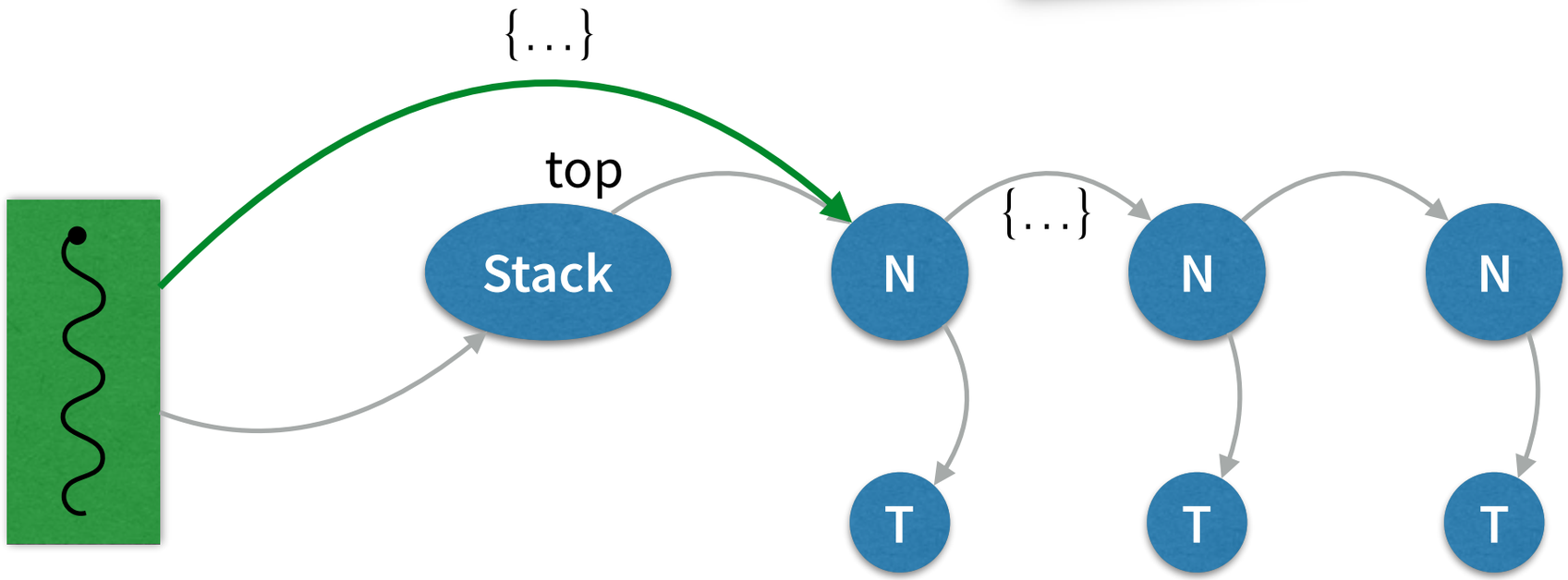
Capability Transfer with Compare-and-Swap

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



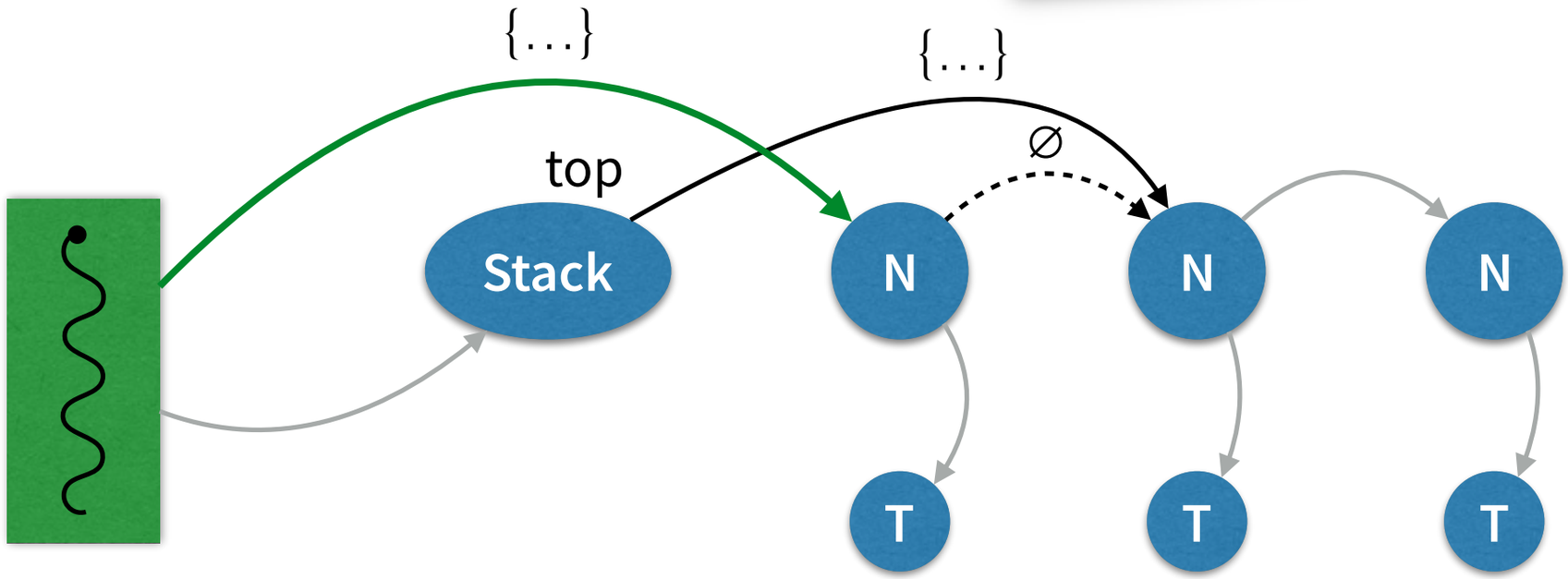
Capability Transfer with Compare-and-Swap

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



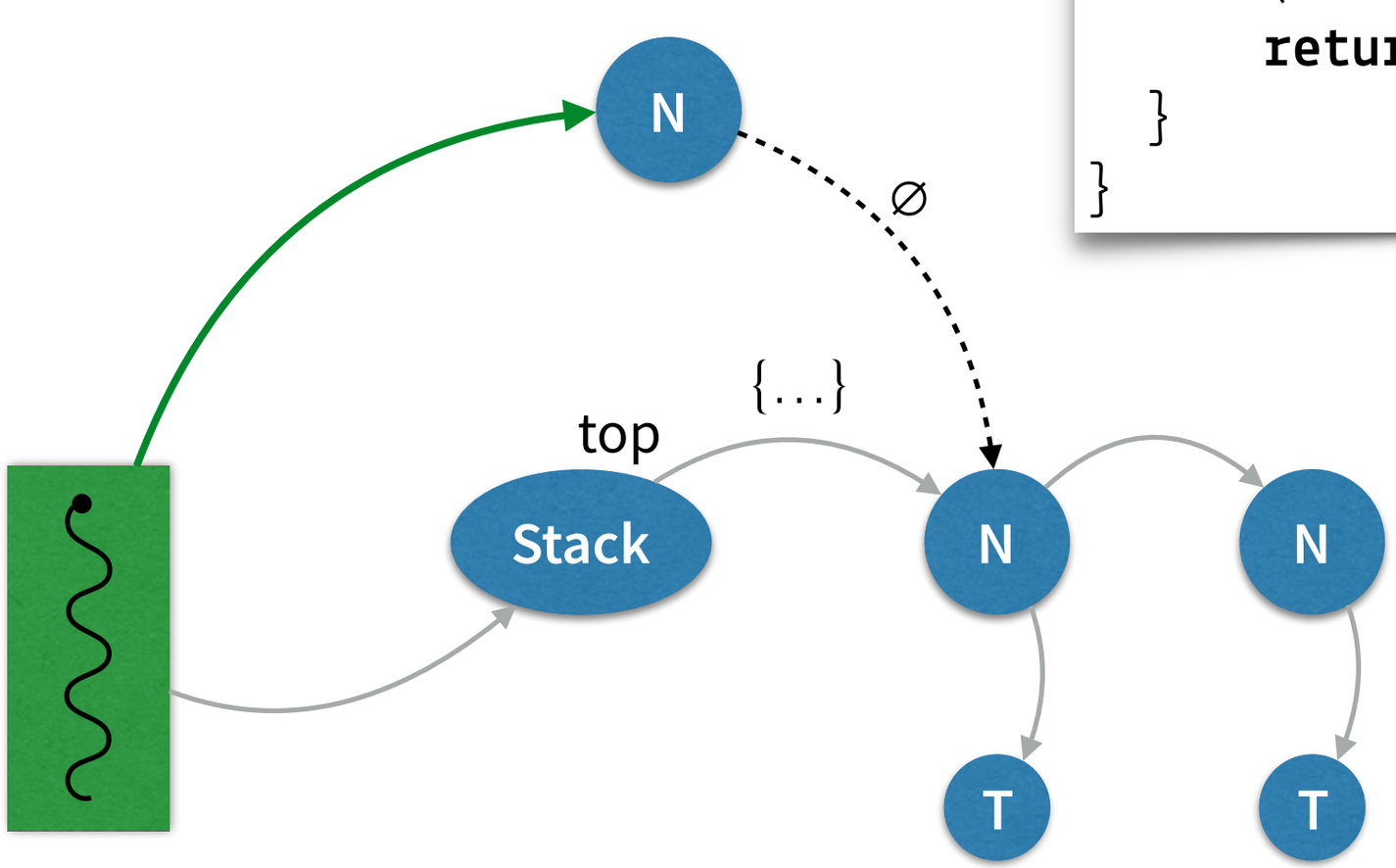
Capability Transfer with Compare-and-Swap

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



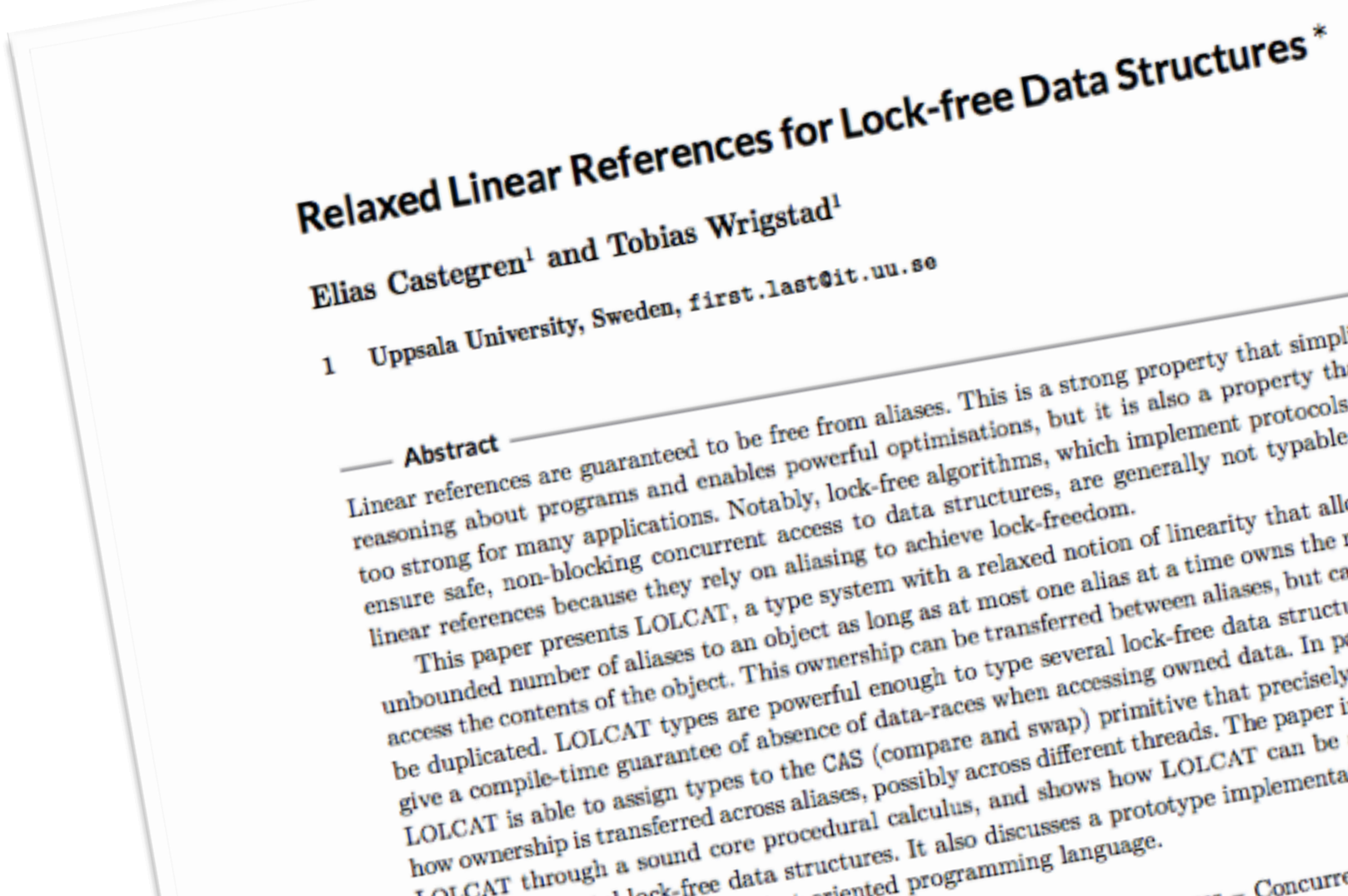
Capability Transfer with Compare-and-Swap

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



LOLCAT [ECOOP ’17]

- Linear Ownership:
  - At most one capability may access an object’s mutable fields
- Three fundamental lock-free data-structures
  - Treiber Stack
  - Michael—Scott Queue
  - Tim Harris List
- No uncontrolled data-races!



Relaxed Linear References for Lock-free Data Structures*

Elias Castegren[1] and Tobias Wrigstad[1]

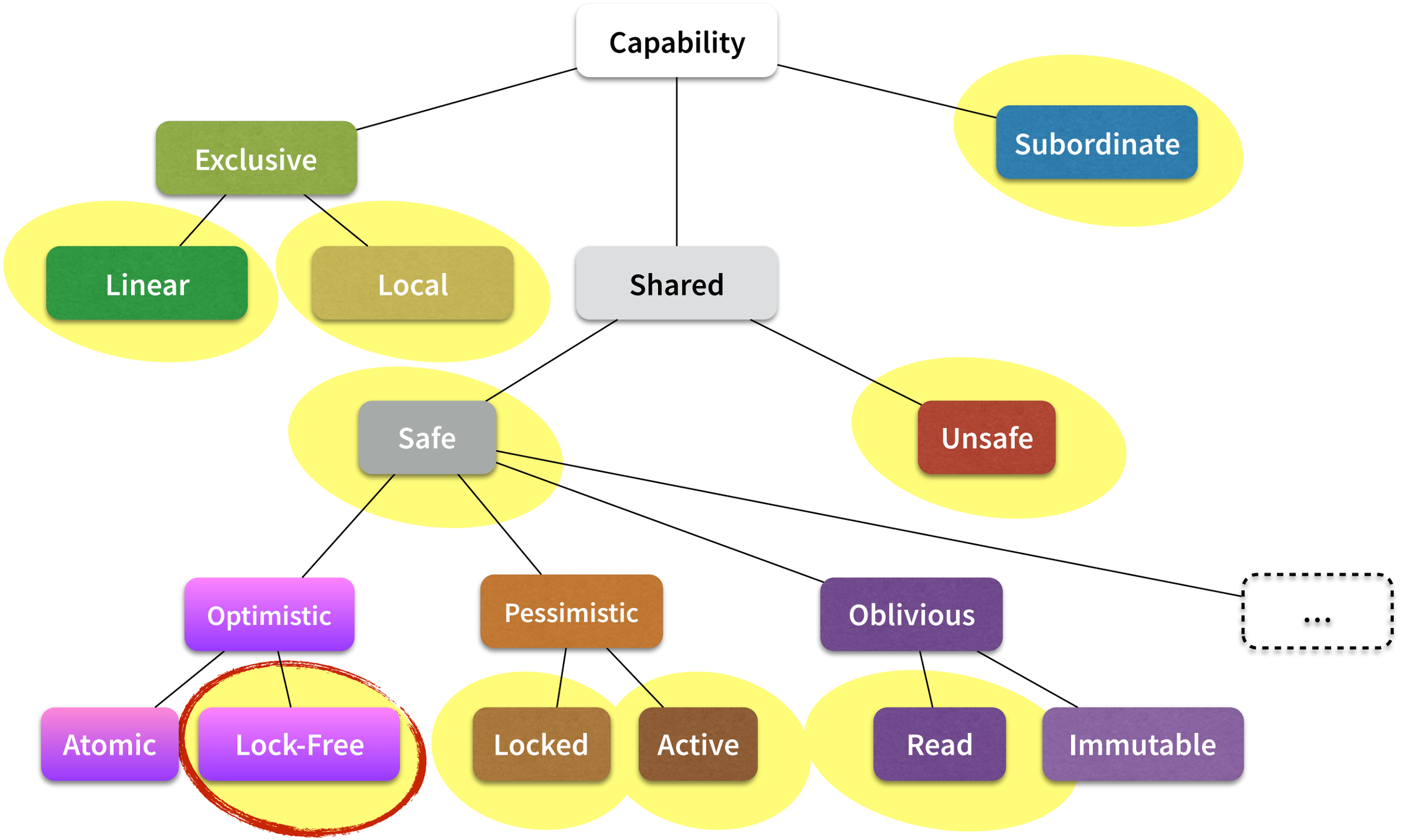
1 Uppsala University, Sweden, first.last@it.uu.se

Abstract

Linear references are guaranteed to be free from aliases. This is a strong property that simplifies
reasoning about programs and enables powerful optimisations, but it is also a property that is
too strong for many applications. Notably, lock-free algorithms, which implement protocols that
ensure safe, non-blocking concurrent access to data structures, are generally not typable with
linear references because they rely on aliasing to achieve lock-freedom.

This paper presents LOLCAT, a type system with a relaxed notion of linearity that allows an
unbounded number of aliases to an object as long as at most one alias at a time owns the right to
access the contents of the object. This ownership can be transferred between aliases, but can not
be duplicated. LOLCAT types are powerful enough to type several lock-free data structures and
give a compile-time guarantee of absence of data-races when accessing owned data. In particular,
LOLCAT is able to assign types to the CAS (compare and swap) primitive that precisely describes
how ownership is transferred across aliases, possibly across different threads. The paper introduces
LOLCAT through a sound core procedural calculus, and shows how LOLCAT can be applied to
lock-free data structures. It also discusses a prototype implementation for a concurrent
object-oriented programming language.

A Taxonomy of Reference Capabilities



eliasc.github.io

@CartesianGlee

EliasC