Proof methods for concurrent programs

1. shared memory, Hoare logic, separation logic

Francesco Zappa Nardelli

INRIA Paris-Rocquencourt, MOSCOVA project-team

francesco.zappa_nardelli@inria.fr

http://moscova.inria.fr/~zappa/teaching/mpri/2010/

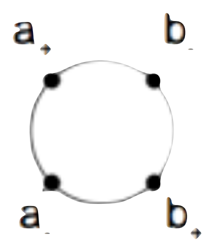
Concurrency, in theory

Example: 2-way Buffers

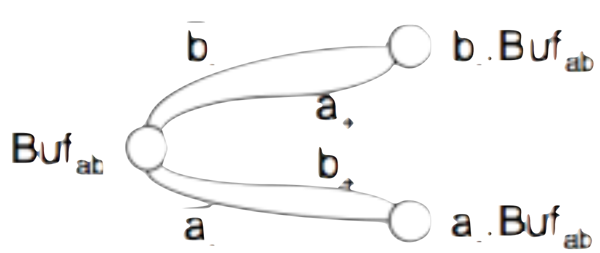
1-place 2-way buffer:

$Buf_{ab} == a_+.\bar{b}_-.Buf_{ab} + b_+.\bar{a}_-.Buf_{ab}$

Flow graph:



LTS:



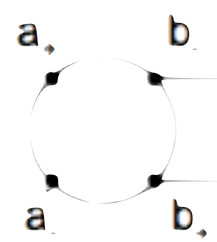
$Buf_{bc} ==$

$\quad Buf_{ab}[c_+/b_+, c_-/b_-, b_-/a_+, b_+/a_-]$

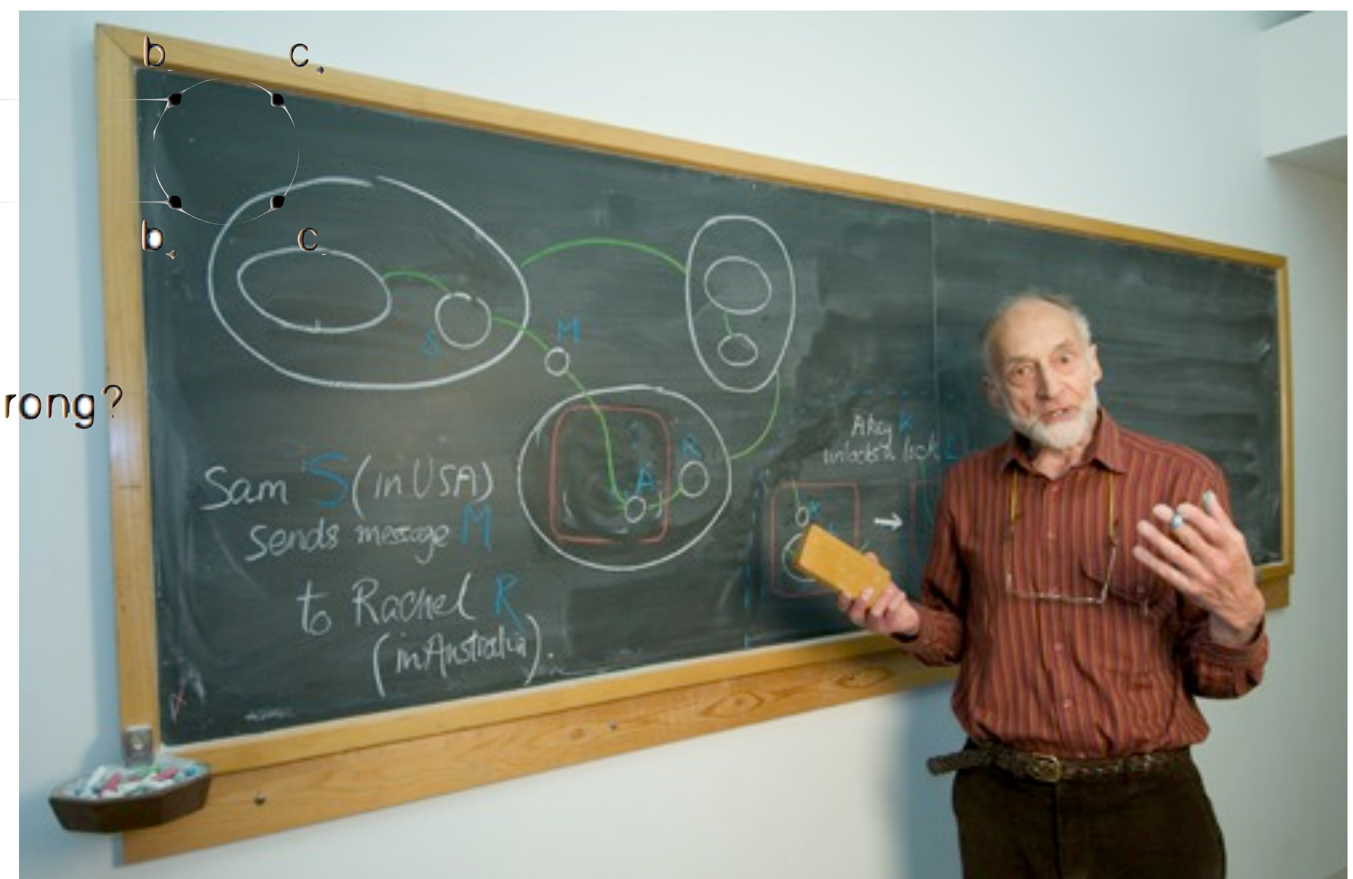
(Obs: Simultaneous substitution!)

$Sys = (Buf_{ab} \mid Buf_{bc})\backslash\{b_+, b_-\}$

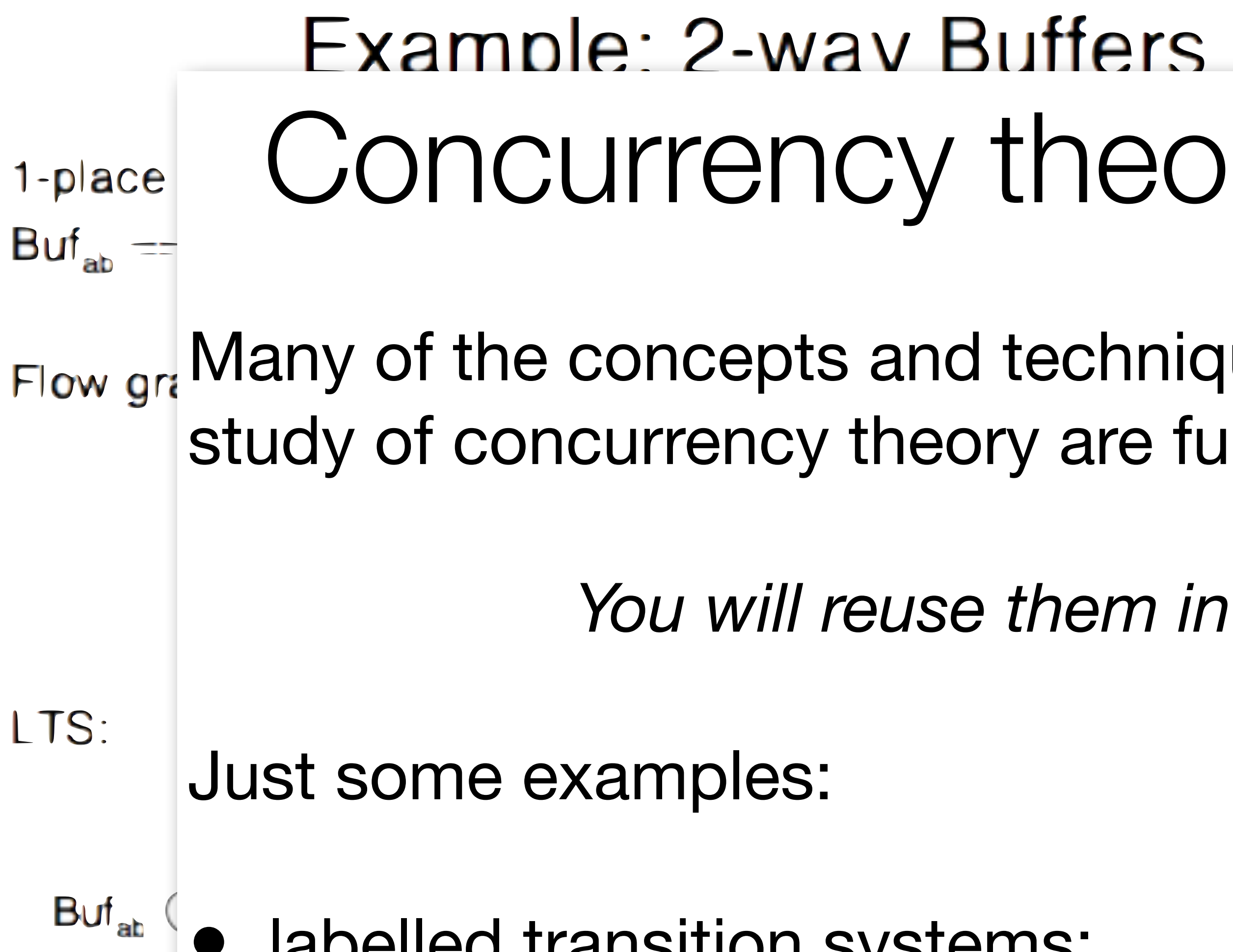
Intention:

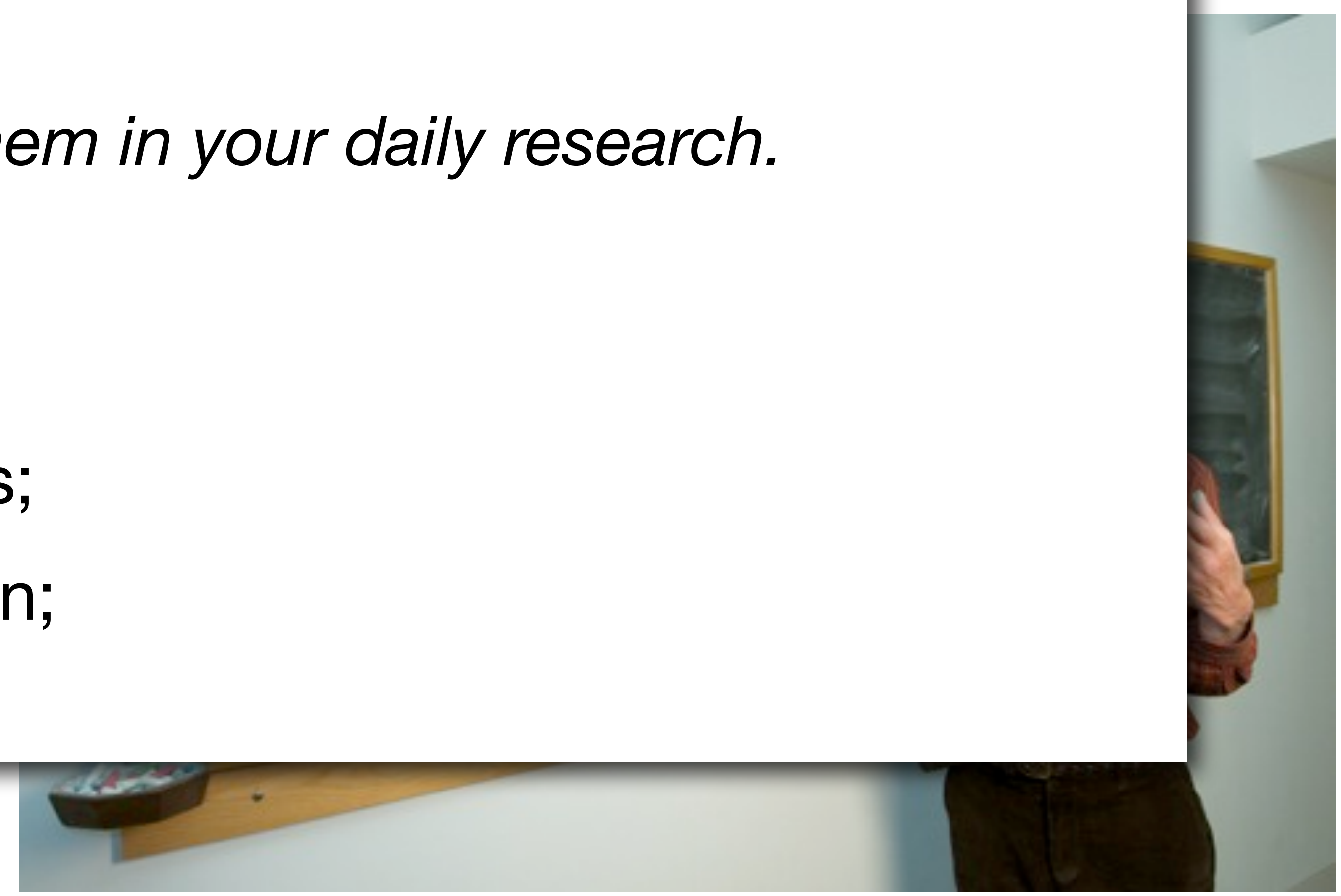


What went wrong?



Concurrency, in theory





Concurrency theory is fundamental

Many of the concepts and techniques developed in 25 years of study of concurrency theory are fundamental.

You will reuse them in your daily research.

Just some examples:

- labelled transition systems;
- simulation and bisimulation;
- contextual equivalences.

Concurrency, in practice

```
void __lockfunc _##op##_lock(locktype##_t *lock)
{
        for (;;) {
                preempt_disable();
                if (likely(_raw_##op##_trylock(lock)))
                        break;
                preempt_enable();

                if (!(lock)->break_lock)
                        (lock)->break_lock = 1;
                while (!op##_can_lock(lock) && (lock)->break_lock)
                        _raw_##op##_relax(&lock->raw_lock);
        }
        (lock)->break_lock = 0;
}
```

excerpt from Linux spinlock.c

Concurrency, in practice

```
void __lockfunc _##op##_lock(locktype##_t *lock)
{
        for (;;) {
                preempt_disable();
                if (likely(_raw_##op##_trylock(lock)))
                        break;
                preempt_enable();

                if (!(lock)->break_lock)
                        (lock)->break_lock = 1;
                while (!op##_can_lock(lock) && (lock)->break_lock)
                        _raw_##op##_relax(&lock->raw_lock);
        }
        (lock)->break_lock = 0;
}
```

excerpt from Linux spinlock.c

```
/**
 * LazyInitRace
 *
 * Race condition in lazy initialization
 *
 * @author Brian Goetz and Tim Peierls
 */

@NotThreadSafe
public class LazyInitRace {
    private ExpensiveObject instance = null;

    public ExpensiveObject getInstance() {
        if (instance == null)
            instance = new ExpensiveObject();
        return instance;
    }
}

class ExpensiveObject { }
```

excerpt from
www.javaconcurrencyinpractice.com

Concurrency, in practice

```
ResourceResponse response;
unsigned long identifier = std::numeric_limits<unsigned long>::max();
if (document->frame())
    identifier = document->frame()->loader()->loadResourceSynchronously(request, storedCredentials, error, response, data

// No exception for file:/// resources, see <rdar://problem/4962298>.
// Also, if we have an HTTP response, then it wasn't a network error in fact.
if (!error.isNull() && !request.url().isLocalFile() && response.httpStatusCode() <= 0) {
    client.didFail(error);
    return;
}

// FIXME: This check along with the one in willSendRequest is specific to xhr and
// should be made more generic.
if (sameOriginRequest && !document->securityOrigin()->canRequest(response.url())) {
    client.didFailRedirectCheck();
    return;
}

client.didReceiveResponse(response);

const char* bytes = static_cast<const char*>(data.data());
int len = static_cast<int>(data.size());
client.didReceiveData(bytes, len);

client.didFinishLoading(identifier);
```

excerpt from WebKit

excerpt from
www.javaconcurrencyinpractice.com

```
        return instance;
    }
}

class ExpensiveObject { }
```

Concurrency, in practice

in practice

sequential code, interaction via shared memory, some OS calls.

Libraries may provide some abstractions (e.g. message passing).
However, somebody must still implement these libraries. And...

Programming is hard:
subtle algorithms, awful corner cases.

Testing is hard:
some behaviours are observed rarely and difficult to reproduce.

Warm-up: let's implement a shared stack.

excerpt from
www.javaconcurrencyinpractice.com

Setup

A program is composed by *threads* that communicate by writing and reading in a *shared memory.* No assumptions about the relative speed of the threads.

In this example we will use a *mild variant* of the *C programming language*:

- local variables: `x`, `y`, ... (allocated on the stack, local to each thread)
- global variables: `Top`, `H`, ... (allocated on the heap, shared between threads)
- data structures: arrays `H[i]`, records `n = t->tl`, ...
- an atomic *compare-and-swap* operation (e.g. CMPXCHG on x86):

```
bool CAS (value_t *addr, value_t exp, value_t new) {
  atomic {
    if (*addr == exp) then { *addr = new; return true; }
    else return false;
  }}
```

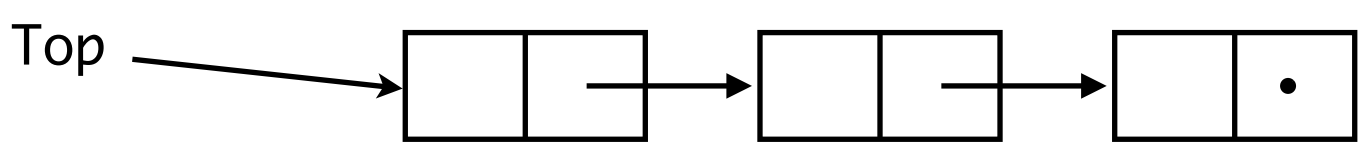
A stack

We implement a stack using a list living in the heap:

- each entry of the stack is a record of two fields:

```
typedef struct entry { value hd; entry *tl } entry
```

- the top of the stack is pointed by `Top`.



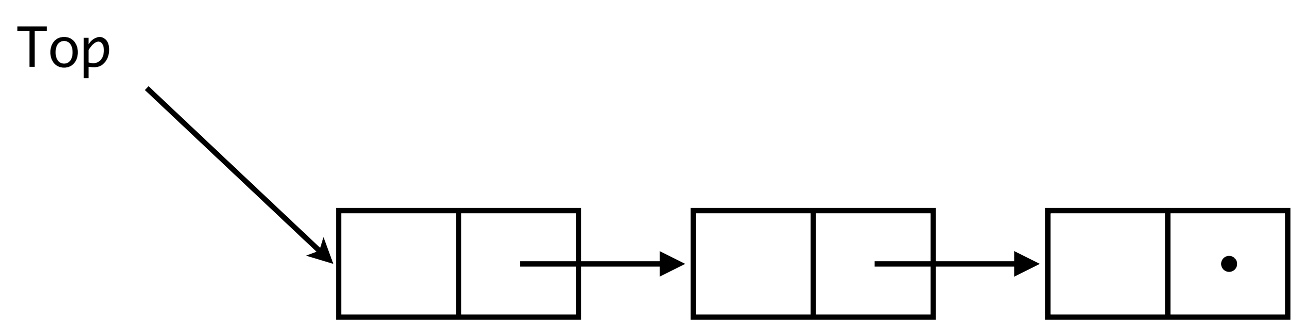

```
pop () {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```

A sequential stack: demo

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

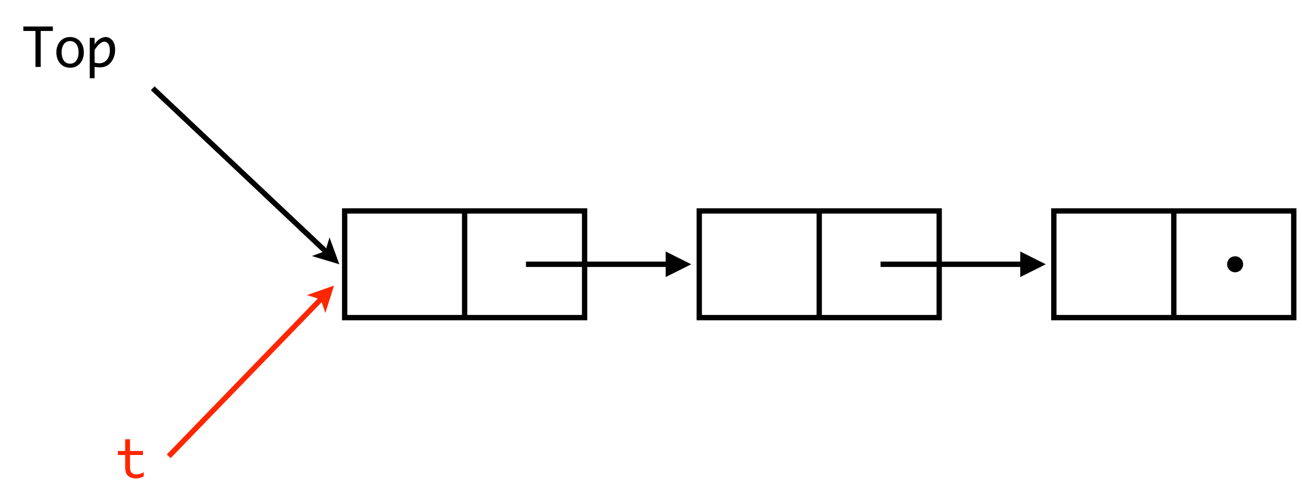
```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```



A sequential stack: pop ( )

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

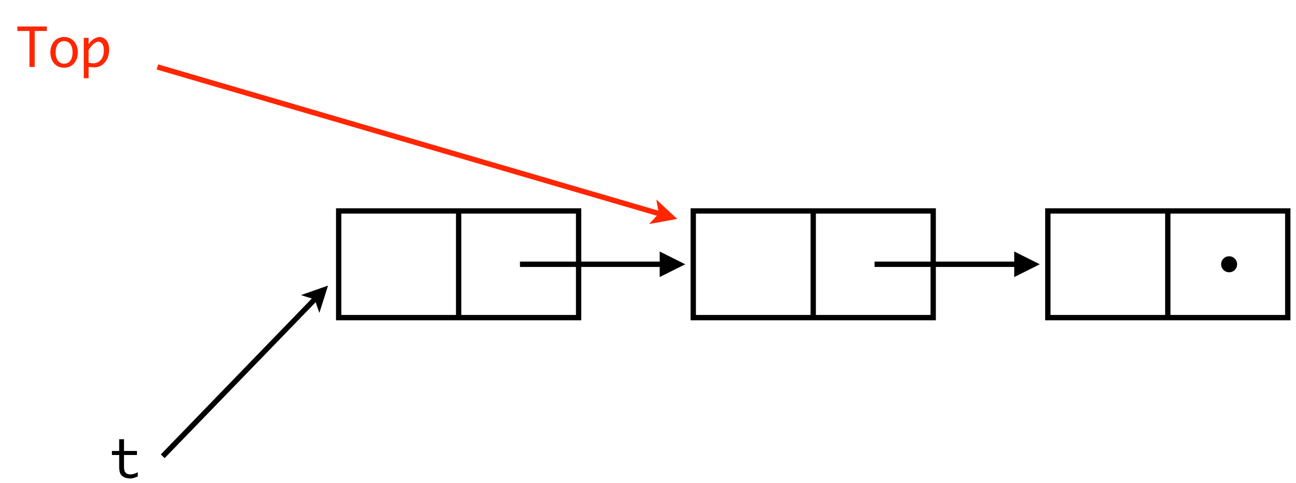
```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```



A sequential stack: pop ( )

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

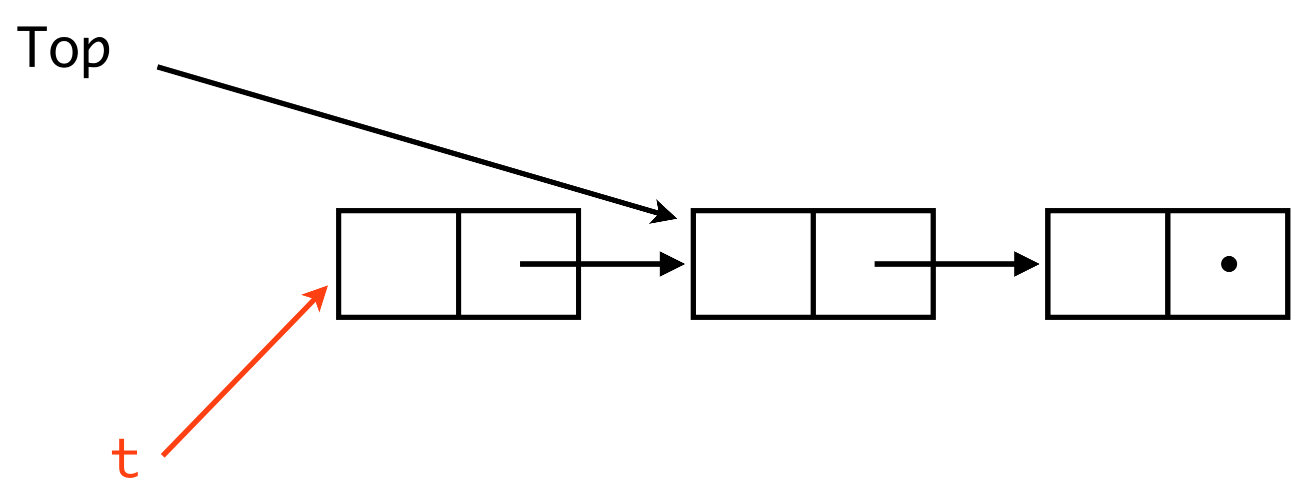
```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```



A sequential stack: pop ( )

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

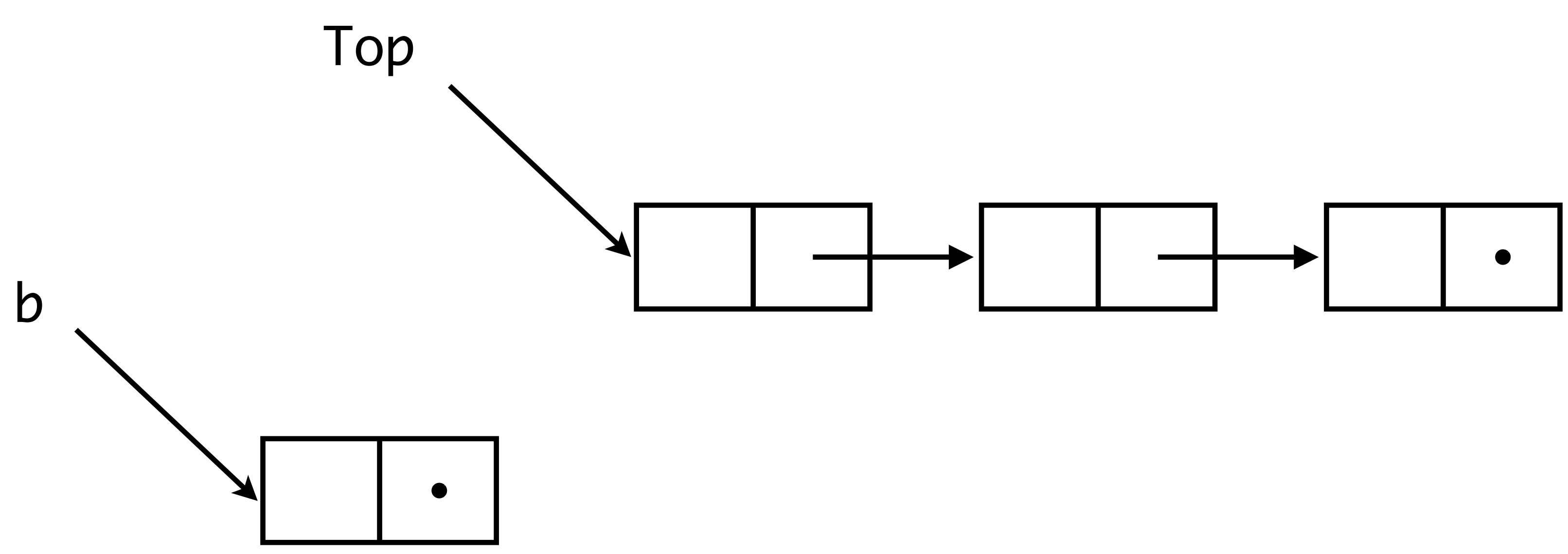
```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```



A sequential stack: push (b)

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

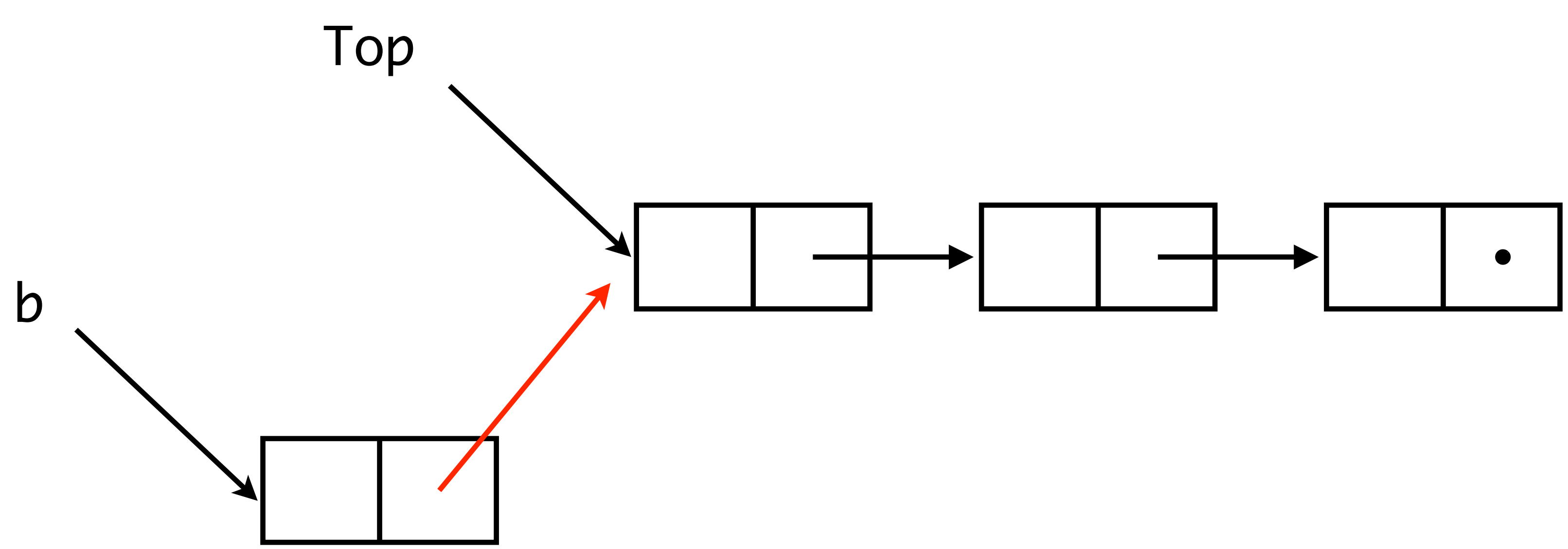
```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```



A sequential stack: push (b)

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

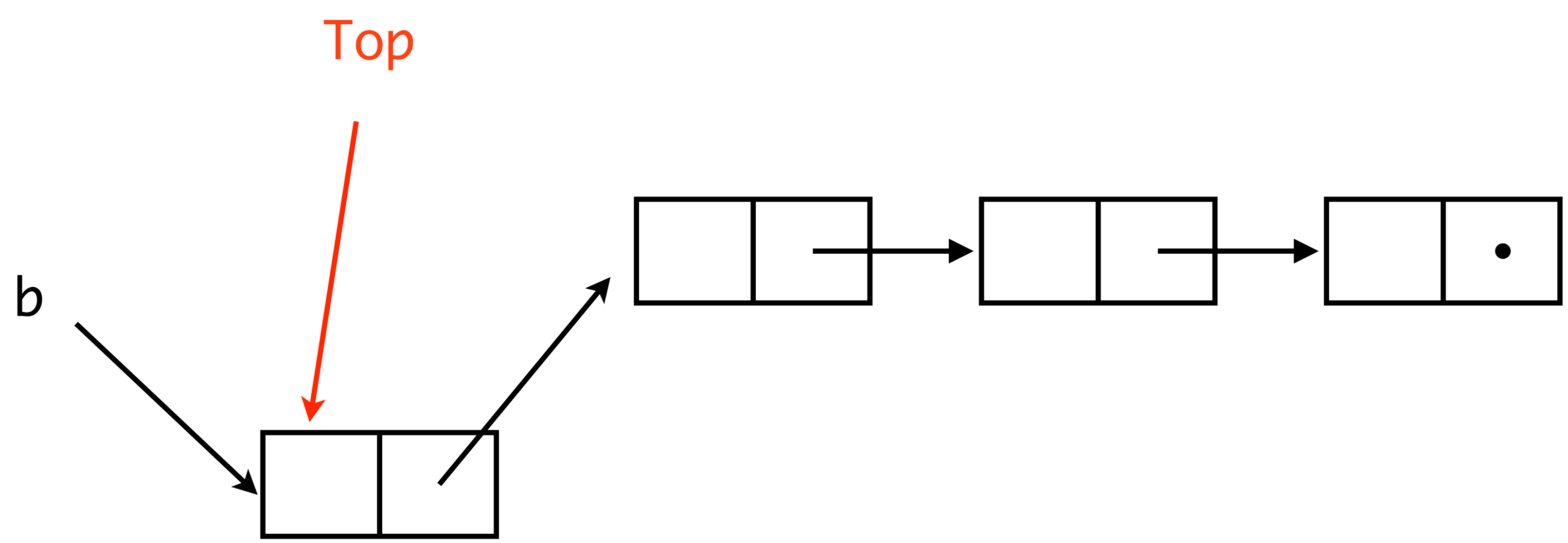
```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```



A sequential stack: push (b)

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

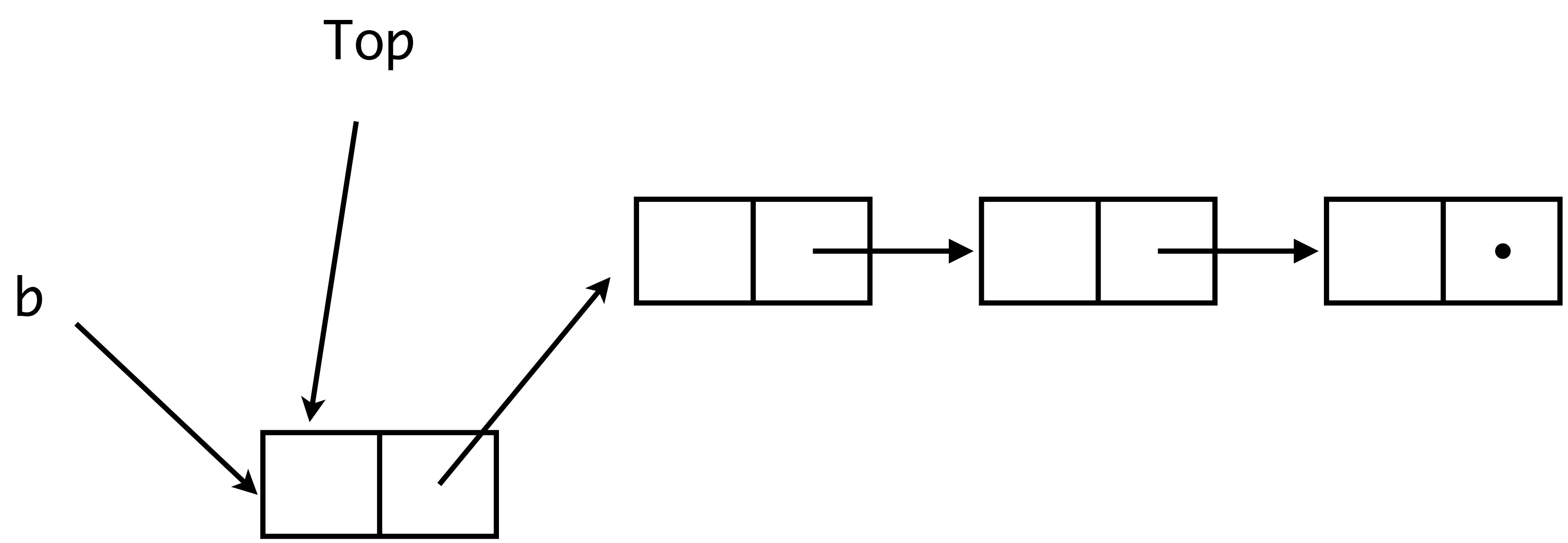
```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```



A sequential stack: push (b)

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```

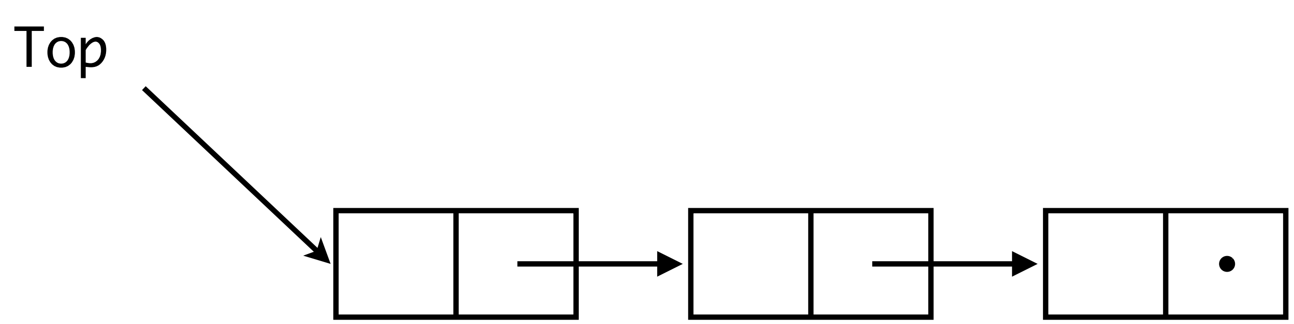


A sequential stack in a concurrent world

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```

Imagine that two threads invoke `pop()` concurrently...

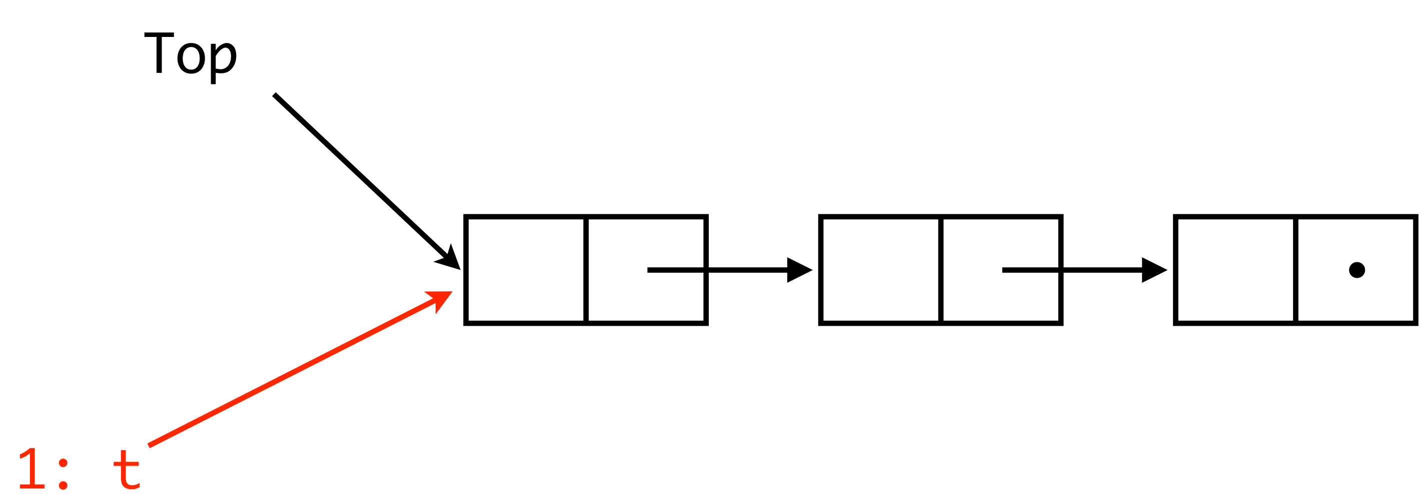


A sequential stack in a concurrent world

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```

Imagine that two threads invoke `pop()` concurrently...

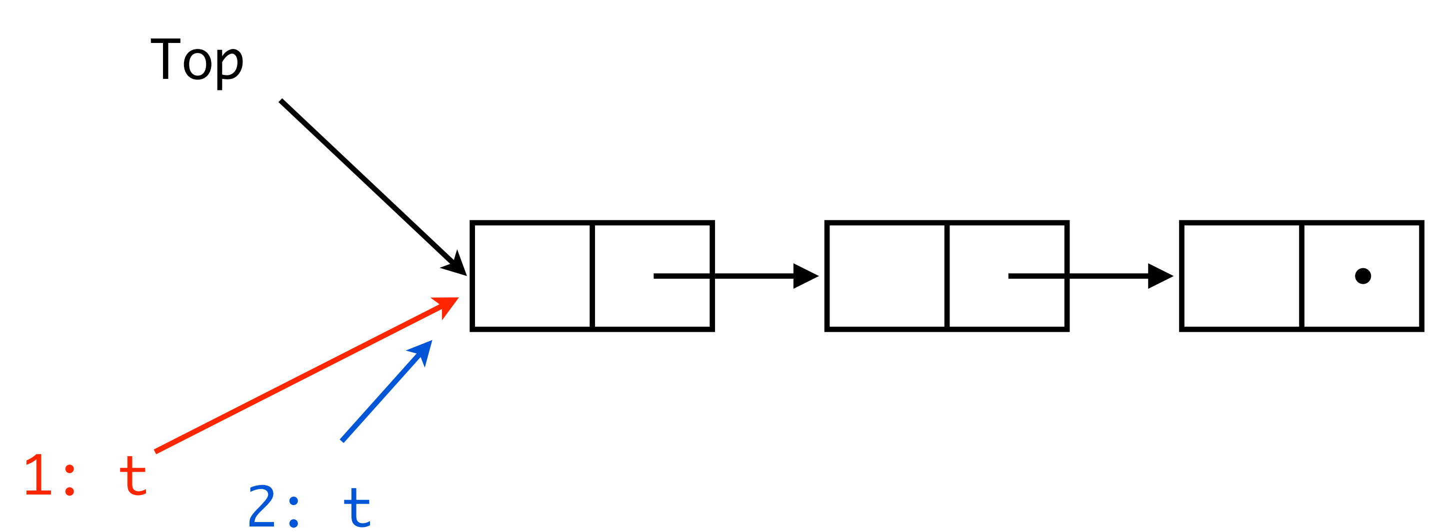


A sequential stack in a concurrent world

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```

Imagine that two threads invoke `pop()` concurrently...

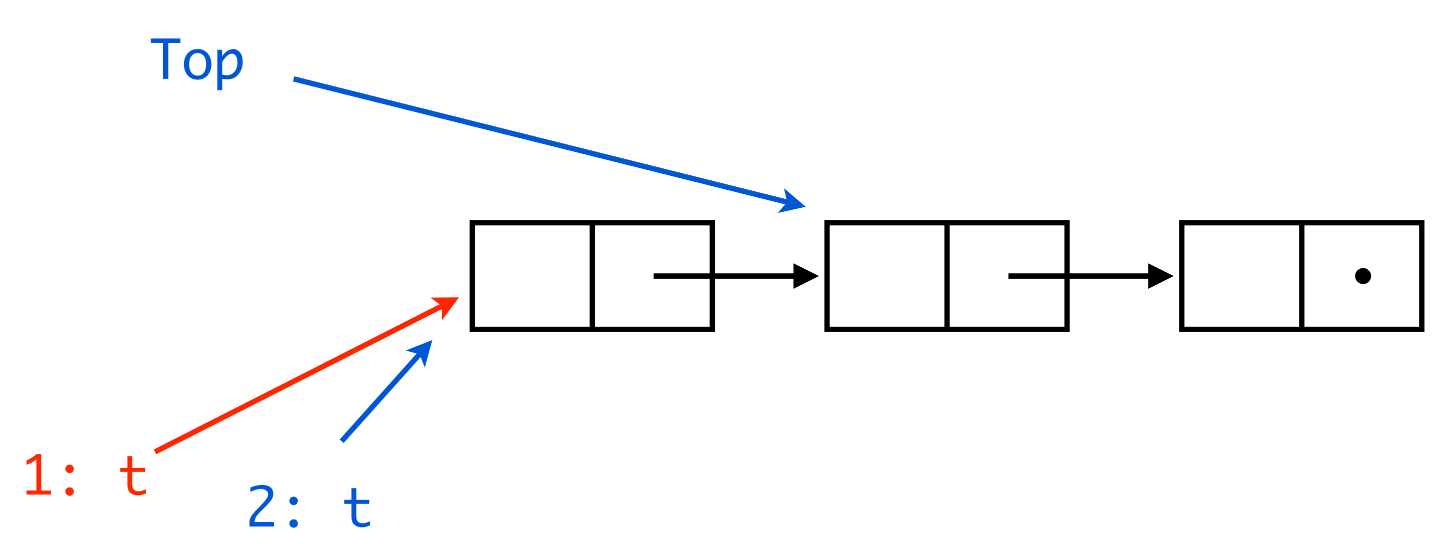


A sequential stack in a concurrent world

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```

Imagine that two threads invoke `pop()` concurrently...



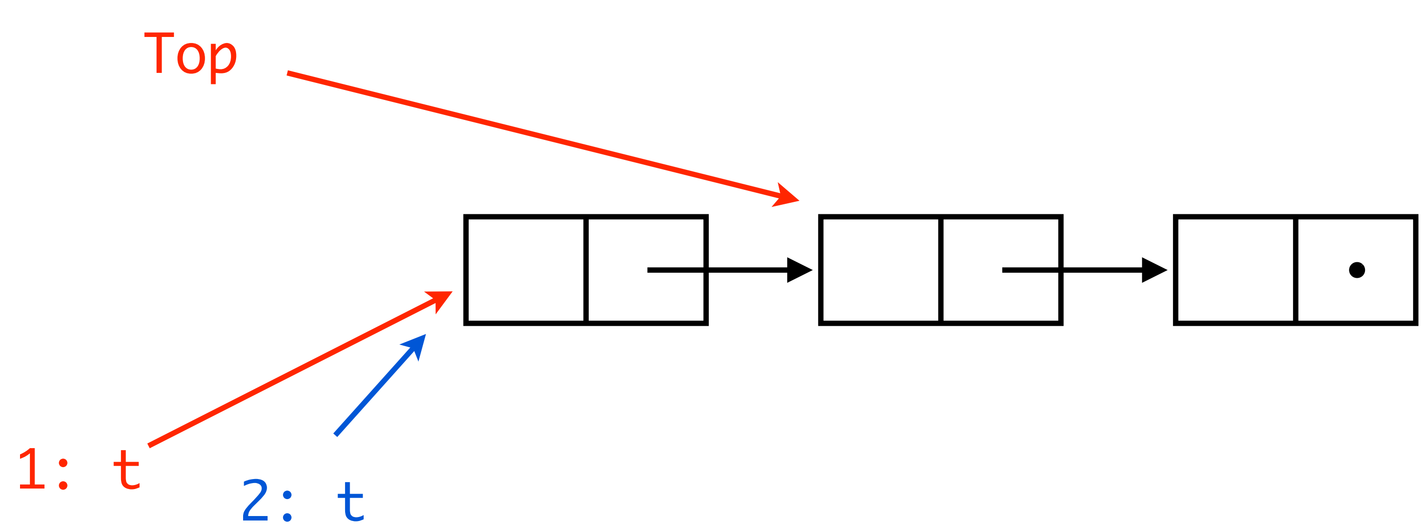
A sequential stack in a concurrent world

```
pop ( ) {
  t = Top;
  if (t != nil)
    Top = t->tl;
  return t;
}
```

```
push (b) {
  b->tl = Top;
  Top = b;
  return true;
}
```

Imagine that two threads invoke `pop()` concurrently...

...the two threads might pop the same entry!



Idea 1: validate the Top pointer using CAS

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  return t;
}
```

```
push (b) {
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

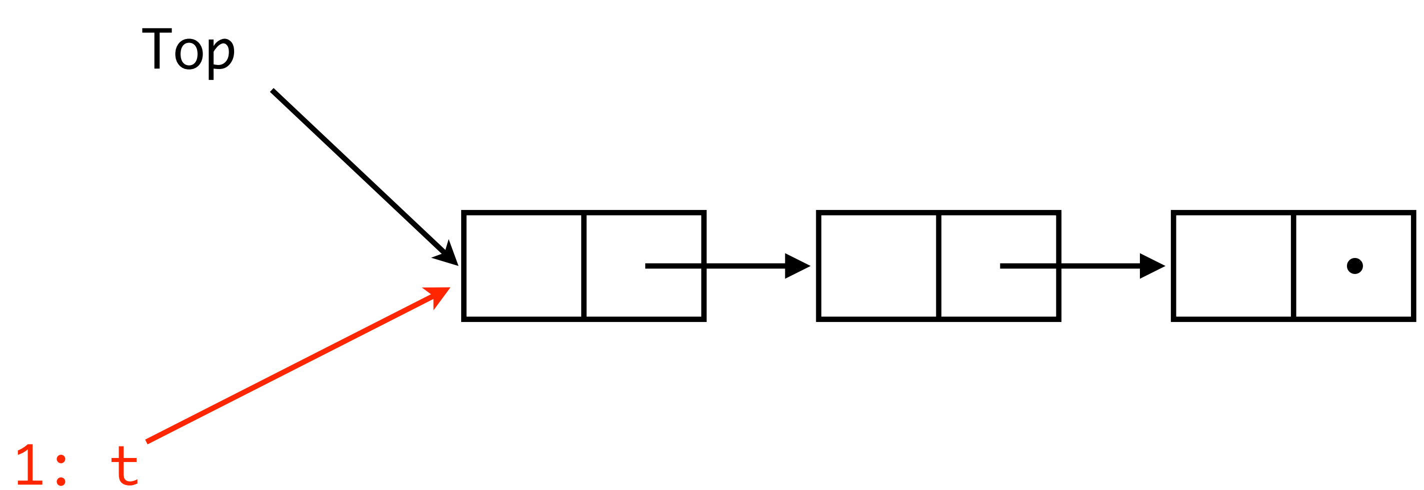
•

Idea 1: validate the Top pointer using CAS

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  return t;
}
```

```
push (b) {
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Two concurrent `pop()` now work fine...

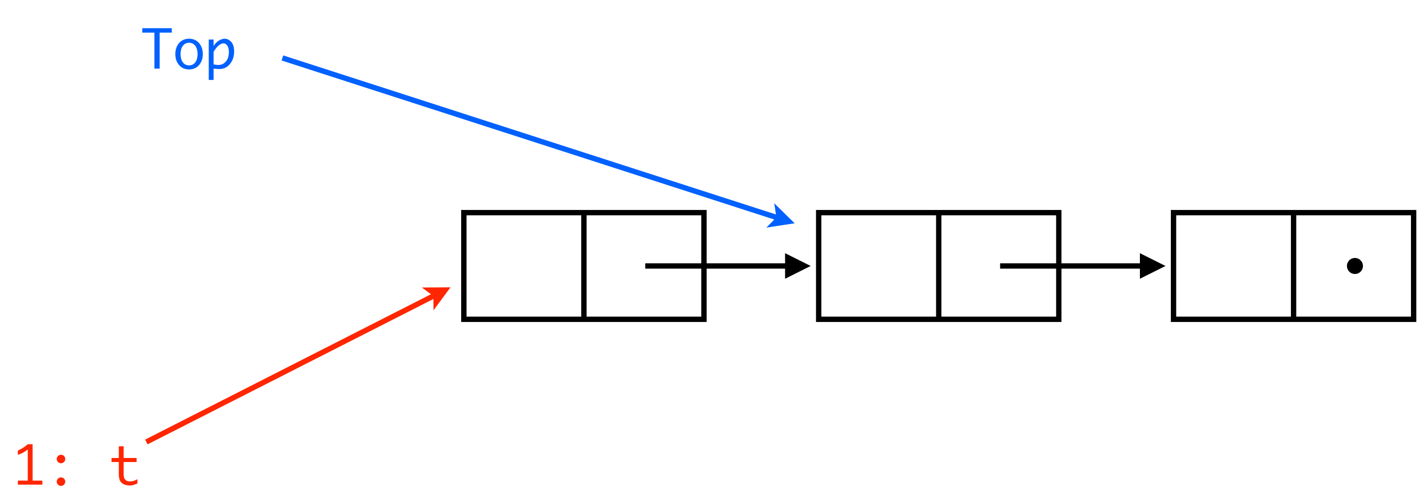


Idea 1: validate the Top pointer using CAS

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  return t;
}
```

```
push (b) {
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Two concurrent `pop()` now work fine...



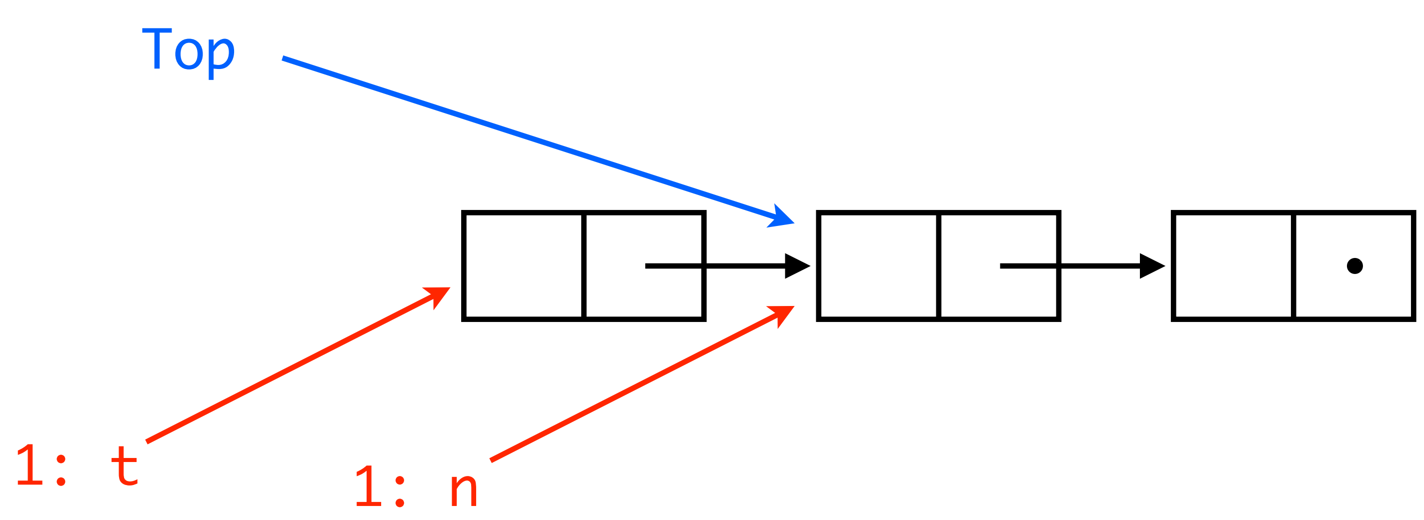
Idea 1: validate the Top pointer using CAS

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  return t;
}
```

```
push (b) {
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Two concurrent `pop()` now work fine...

The `CAS` of Th. 1 fails.

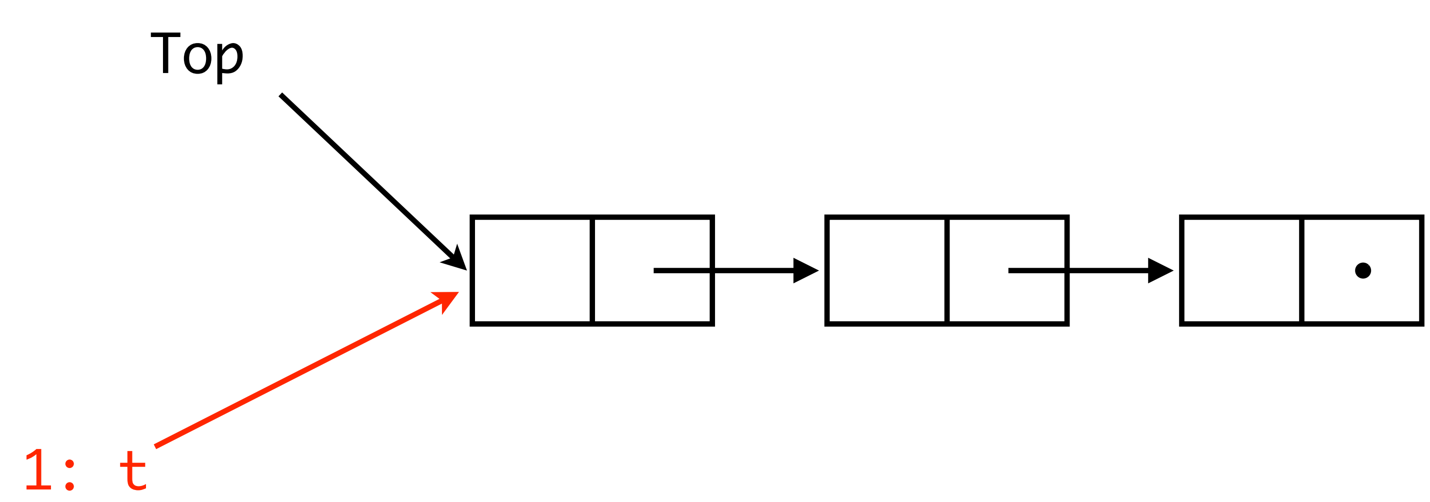


The ABA problem

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  return t;
}
```

```
push (b) {
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Th 1 starts popping...

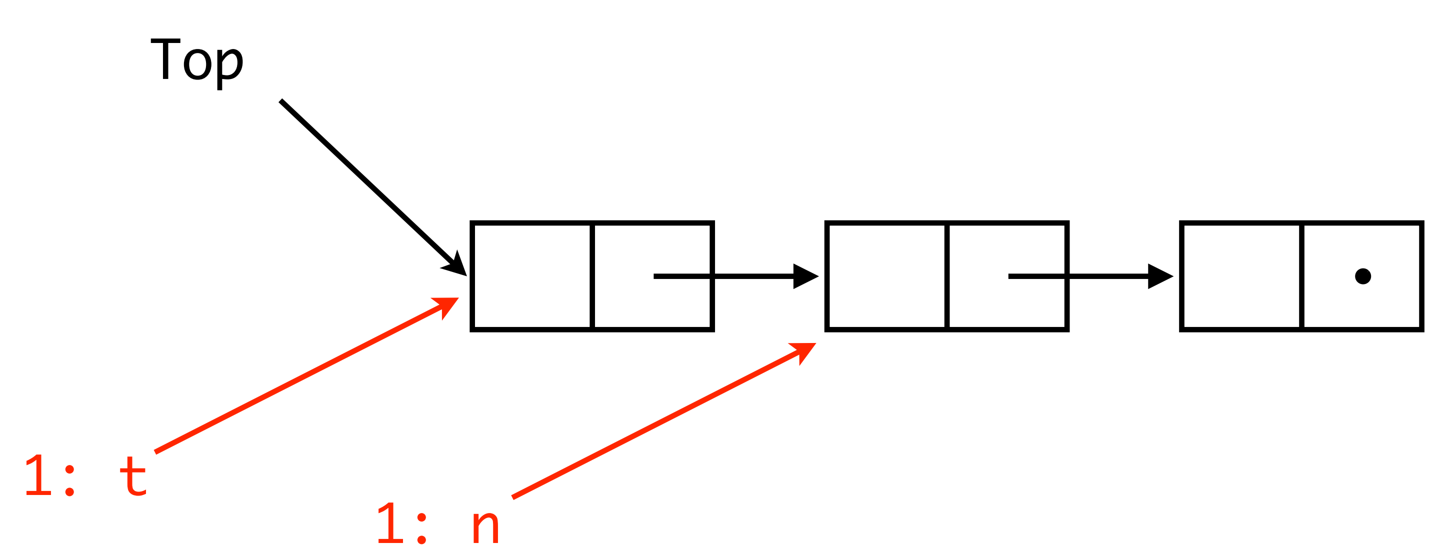


The ABA problem

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  return t;
}
```

```
push (b) {
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Th 1 starts popping...

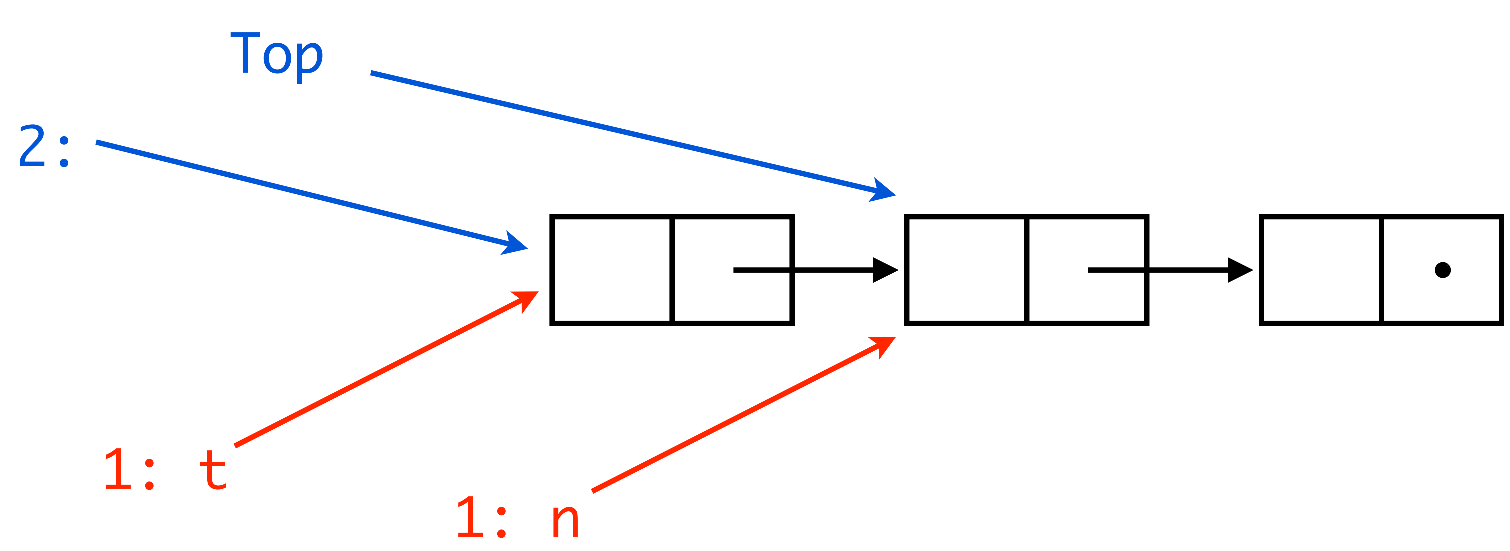


The ABA problem

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  return t;
}
```

```
push (b) {
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Th 2 pops...

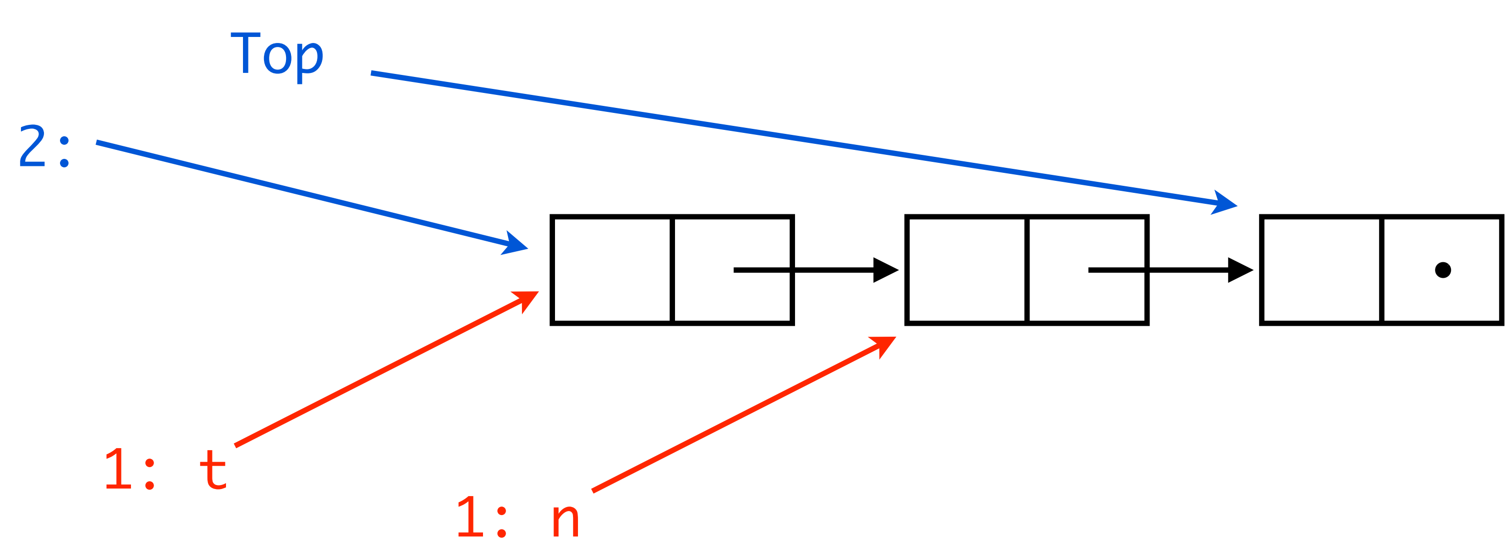


The ABA problem

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  return t;
}
```

```
push (b) {
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Th 2 pops again...

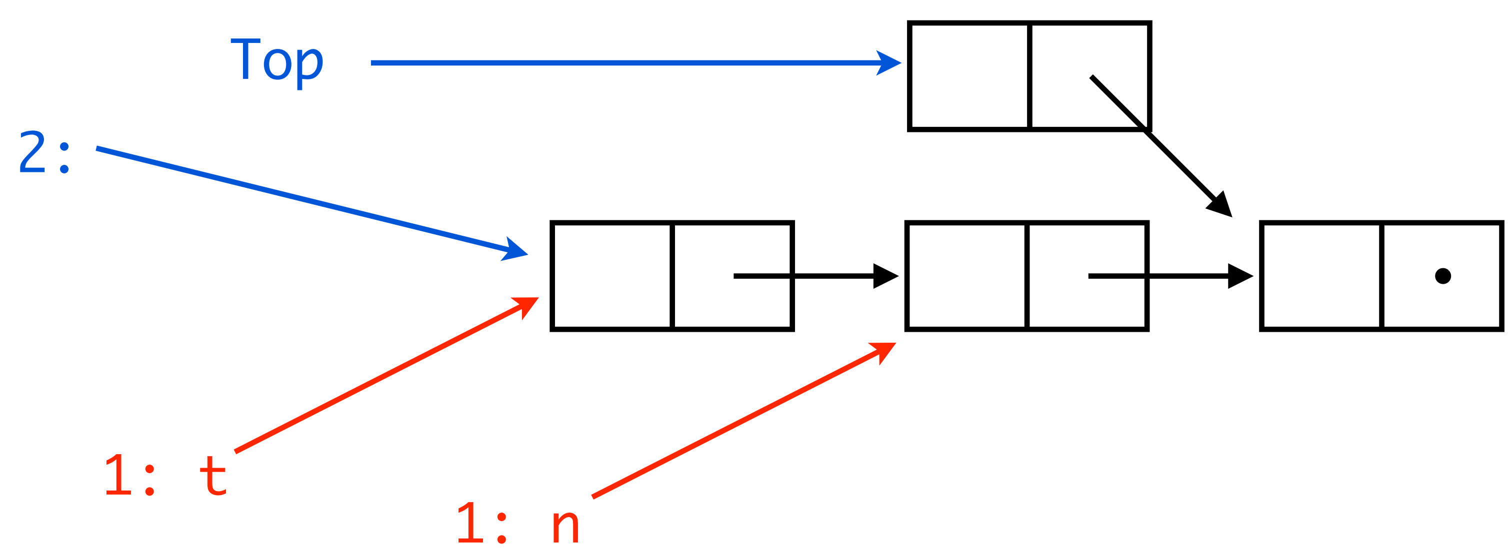


The ABA problem

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  return t;
}
```

```
push (b) {
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Th 2 pushes a new node...

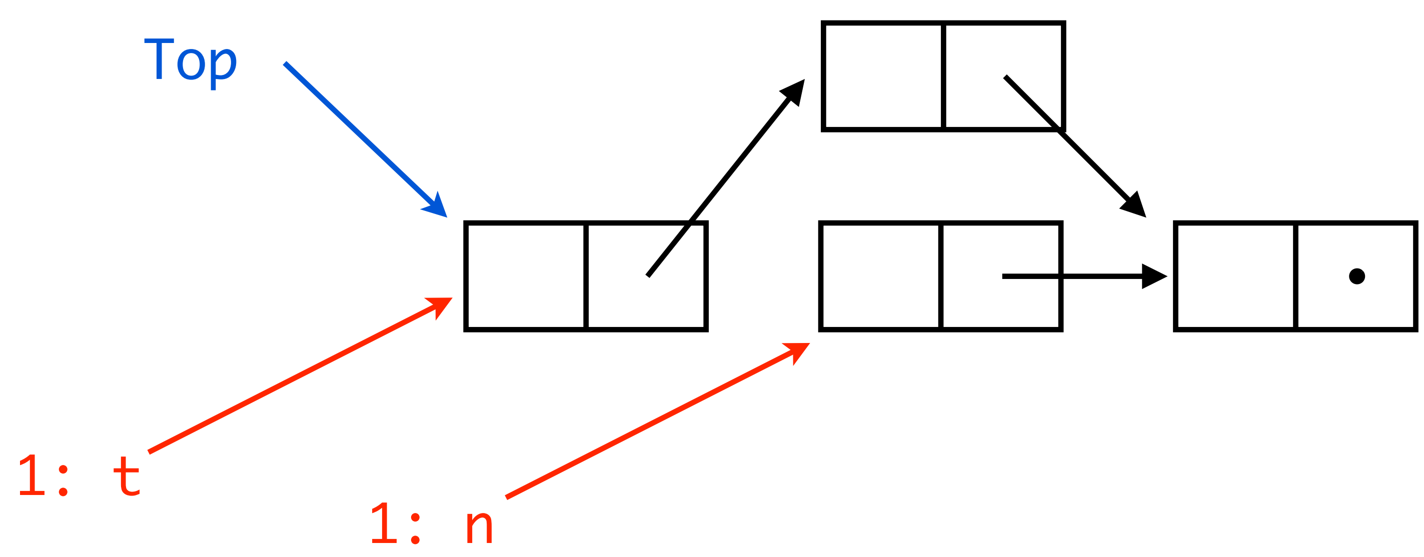


The ABA problem

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  return t;
}
```

```
push (b) {
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Th 2 pushes the old head of the stack...

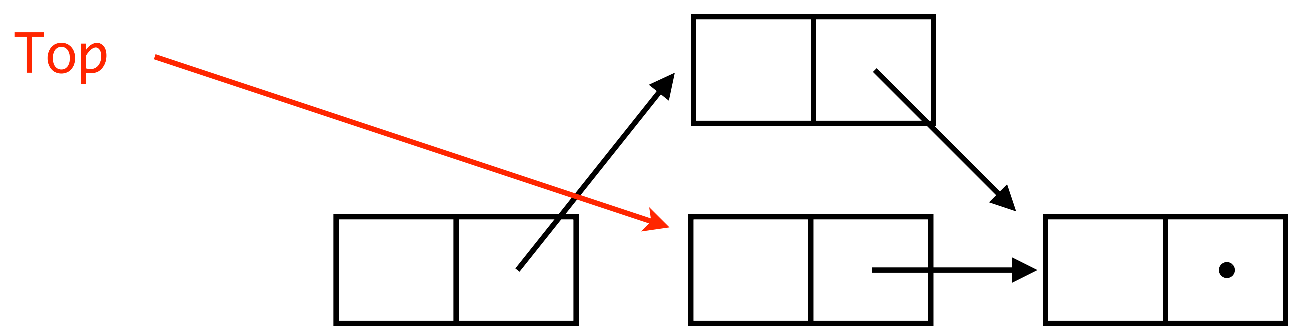


The ABA problem

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  return t;
}
```

```
push (b) {
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Th 1 corrupts the stack...



The hazard pointers methodology

Michael adds to the previous algorithm *a global array* `H` *of hazard pointers*:

- thread `i` alone is allowed to write to element `H[i]` of the array;
- any thread can read any entry of `H`.

The algorithm is then modified:

- before popping a cell, a thread puts its address into its own element of `H`. This entry is cleared only if `CAS` succeeds or the stack is empty;
- before pushing a cell, a thread checks to see whether it is pointed to from any element of `H`. If it is, push is delayed.

Michael's algorithm, simplified

```
pop ( ) {
  while (true) {
    atomic { t = Top;
             H[tid] = t; };
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  H[tid] = nil;
  return t;
}
```

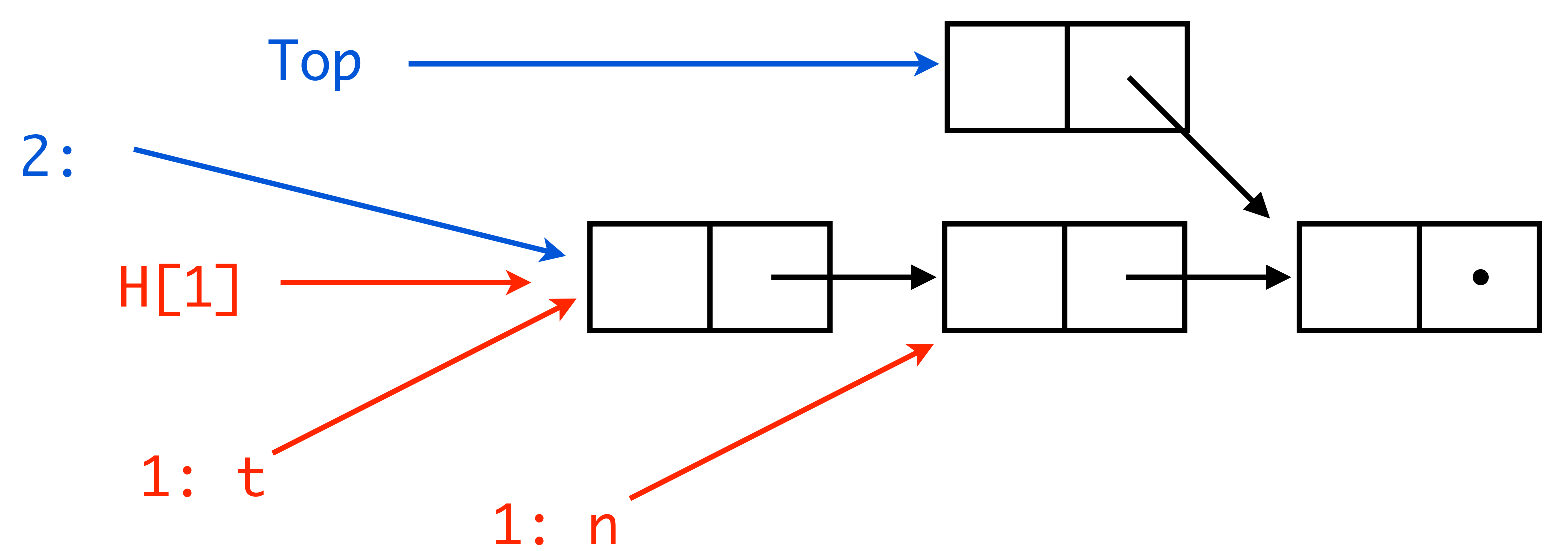
```
push (b) {
  for (n = 0; n < no_threads, n++)
    if (H[n] == b) return false;
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Michael's algorithm, simplified

```
pop ( ) {
  while (true) {
    atomic { t = Top;
             H[tid] = t; };
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  H[tid] = nil;
  return t;
}
```

```
push (b) {
  for (n = 0; n < no_threads, n++)
    if (H[n] == b) return false;
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Th 2 cannot push the old head, because Th 1 has an hazard pointer on it...



Key properties of Michael's simplified algorithm

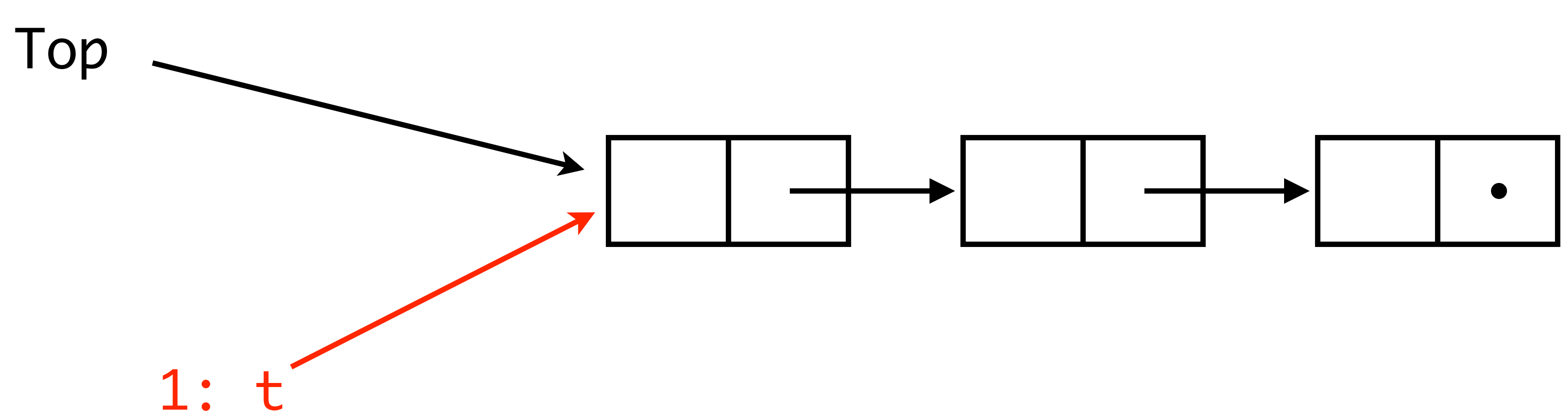
- A node can be added to the hazard array only if it is reachable through the stack;
- a node that has been popped is not reachable through the stack;
- a node that is unreachable in the stack and that is in the hazard array cannot be added to the stack;
- while a node is reachable and in the hazard array, it has a constant tail.

These are a good example of the properties we might want to state and prove about a concurrent algorithm.

The role of *atomic*

```
pop ( ) {
  while (true) {
    t = Top;
    H[tid] = t;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  H[tid] = nil;
  return t;
}
```

```
push (b) {
  for (n = 0; n < no_threads, n++)
    if (H[n] == b) return false;
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

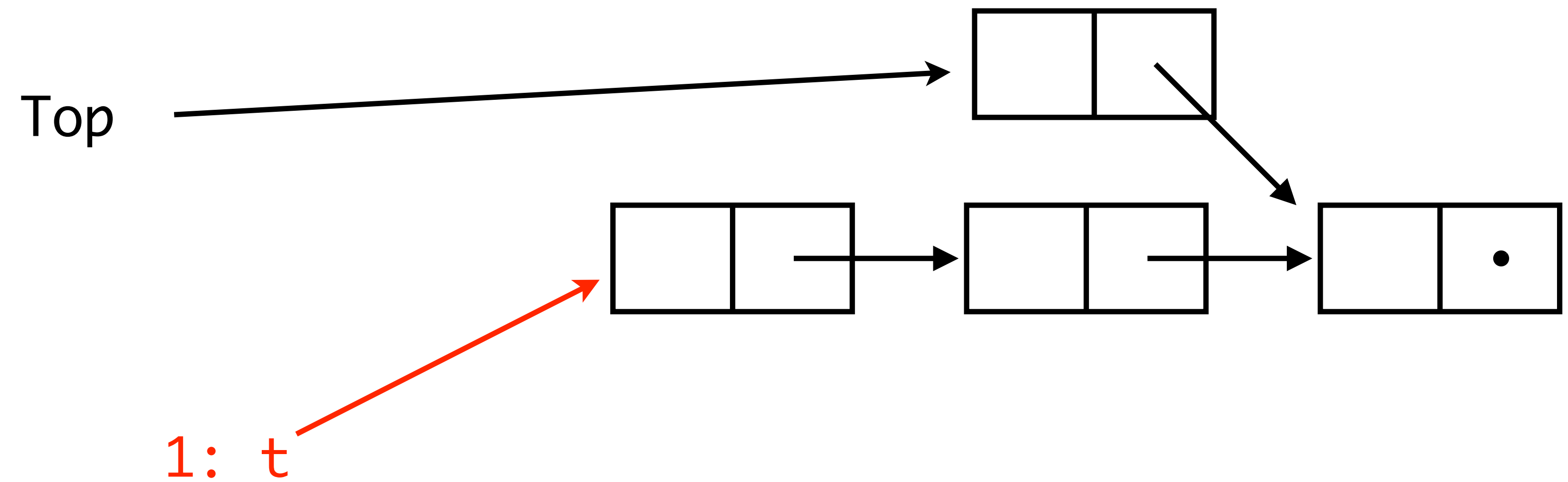


Th 1 copies Top...

The role of *atomic*

```
pop ( ) {
  while (true) {
    t = Top;
    H[tid] = t;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  H[tid] = nil;
  return t;
}
```

```
push (b) {
  for (n = 0; n < no_threads, n++)
    if (H[n] == b) return false;
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

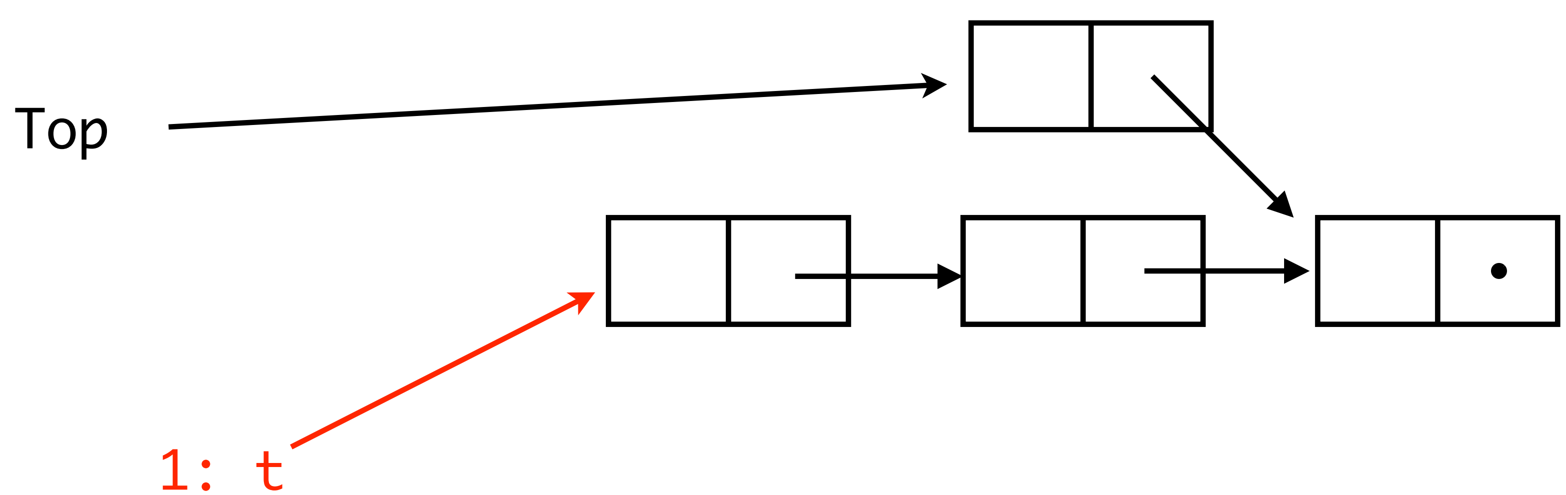


Th 2 pops twice, and pushes a new node...

The role of *atomic*

```
pop ( ) {
  while (true) {
    t = Top;
    H[tid] = t;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  H[tid] = nil;
  return t;
}
```

```
push (b) {
  for (n = 0; n < no_threads, n++)
    if (H[n] == b) return false;
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```



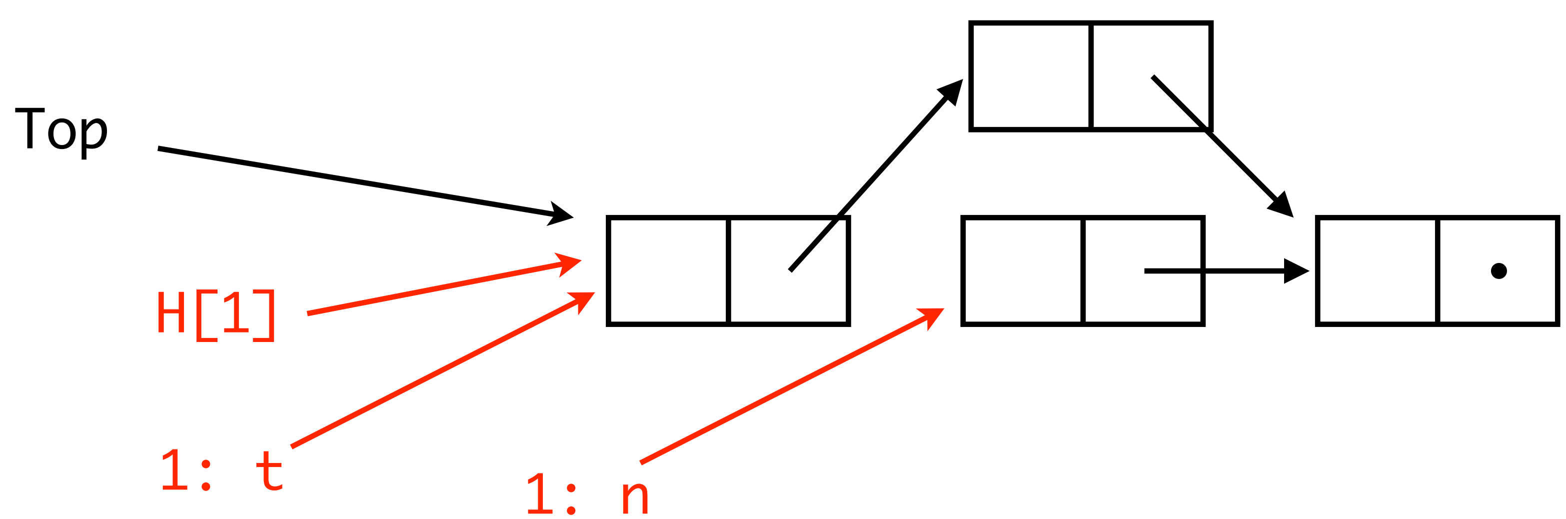
Th 2 starts pushing the old head, and is halfway in the for loop...

The role of *atomic*

```
pop ( ) {
  while (true) {
    t = Top;
    H[tid] = t;
    if (t == nil) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  H[tid] = nil;
  return t;
}
```

```
push (b) {
  for (n = 0; n < no_threads, n++)
    if (H[n] == b) return false;
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Th 1 sets its hazard pointer… but Th 2 might not see the hazard pointer of Th 1!



Michael shared stack

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    H[tid] = t;
    if (t != Top) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  H[tid] = nil;
  return t;
}
```

```
push (b) {
  for (n = 0; n < no_threads, n++)
    if (H[n] == b) return false;
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

Trust me: if we validate `t` against the `Top` pointer before reading `t->tl`, we get a correct algorithm.

Michael shared stack

```
pop ( ) {
  while (true) {
    t = Top;
    if (t == nil) break;
    H[tid] = t;
    if (t != Top) break;
    n = t->tl;
    if CAS(&Top,t,n) break;
  }
  H[tid] = nil;
  return t;
}
```

```
push (b) {
  for (n = 0; n < no_threads, n++)
    if (H[n] == b) return false;
  while (true) {
    t = Top;
    b->tl = t;
    if CAS(&Top,t,b) break;
  }
  return true;
}
```

HOW CAN WE BE SURE?

Michael shared stack

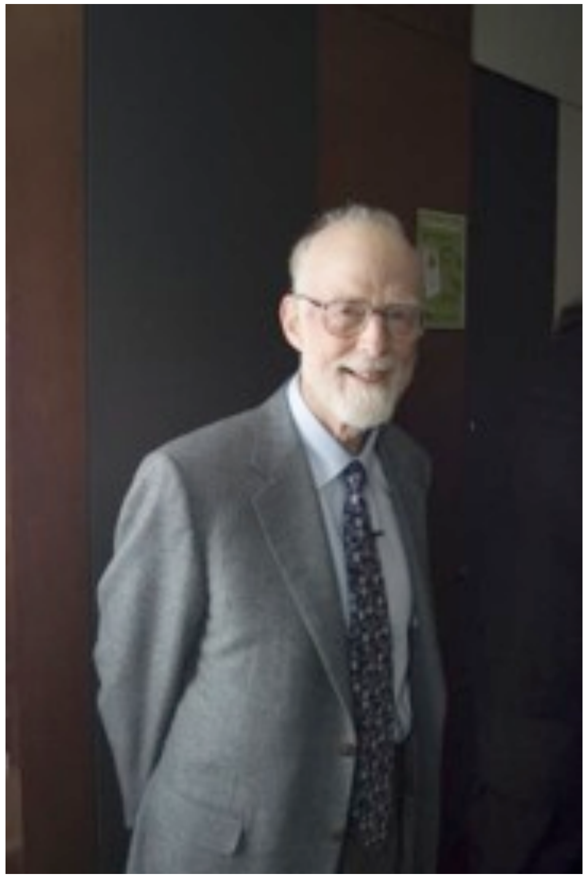
That algorithm is insane… I will never use it in my everyday programming.

Michael shared stack

That algorithm is insane… I will never use it in my everyday programming.

Yes, you will! Michael algorithms are part of `java.util.concurrent`.

Background: Hoare logic



What does it mean fo

In 1969, a seminal paper by Hoare
what a program does:

- C is a program;
- P (the *precondition*) and Q (the
  variables used in C.

We say that

{ P

if whenever C is executed in a stat
terminates, then the state in whicl

An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

KEY WORDS AND PHRASES: axiomatic method, theory of programming' proofs of programs, formal language definition, programming language design, machine-independent programming, program documentation
CR CATEGORY: 4.0, 4.21, 4.22, 5.20, 5.21, 5.23, 5.24

1. Introduction

Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms. It is therefore desirable and interesting to elucidate the axioms and rules of inference which underlie our reasoning about computer programs. The exact choice of axioms will to some extent depend on the choice of programming language. For illustrative purposes, this paper is confined to a very simple language, which is effectively a subset of all current procedure-oriented languages.

2. Computer Arithmetic

The first requirement in valid reasoning about a program is to know the properties of the elementary operations which it invokes, for example, addition and multiplication of integers. Unfortunately, in several respects computer arithmetic is not the same as the arithmetic familiar to mathematicians, and it is necessary to exercise some care in selecting an appropriate set of axioms. For example, the axioms displayed in Table I are rather a small selection of axioms relevant to integers. From this incomplete set of axioms it is possible to deduce such simple theorems as:

$$x = x + y \times 0$$

$$y \leqslant r \supset r + y \times q = (r - y) + y \times (1 + q)$$

The proof of the second of these is:

A5	$(r - y) + y \times (1 + q)$
	$= (r - y) + (y \times 1 + y \times q)$
A9	$= (r - y) + (y + y \times q)$
A3	$= ((r - y) + y) + y \times q$
A6	$= r + y \times q$ provided $y \leqslant r$

The axioms A1 to A9 are, of course, true of the traditional infinite set of integers in mathematics. However, they are also true of the finite sets of "integers" which are manipulated by computers provided that they are confined to *nonnegative* numbers. Their truth is independent of the size of the set; furthermore, it is largely independent of the choice of technique applied in the event of "overflow"; for example:

(1) Strict interpretation: the result of an overflowing operation does not exist; when overflow occurs, the offending program never completes its operation. Note that in this case, the equalities of A1 to A9 are strict, in the sense that both sides exist or fail to exist together.

(2) Firm boundary: the result of an overflowing operation is taken as the maximum value represented.

(3) Modulo arithmetic: the result of an overflowing operation is computed modulo the size of the set of integers represented.

These three techniques are illustrated in Table II by addition and multiplication tables for a trivially small model in which 0, 1, 2, and 3 are the only integers represented.

It is interesting to note that the different systems satisfying axioms A1 to A9 may be rigorously distinguished from each other by choosing a particular one of a set of mutually exclusive supplementary axioms. For example, infinite arithmetic satisfies the axiom:

$A10_I$ $\neg \exists x \forall y \quad (y \leqslant x)$,

where all finite arithmetics satisfy:

$A10_F$ $\forall x \quad (x \leqslant \max)$

where "max" denotes the largest integer represented.

Similarly, the three treatments of overflow may be distinguished by a choice of one of the following axioms relating to the value of max + 1:

$A11_S$	$\neg \exists x \quad (x = \max + 1)$	(strict interpretation)
$A11_B$	$\max + 1 = \max$	(firm boundary)
$A11_M$	$\max + 1 = 0$	(modulo arithmetic)

Having selected one of these axioms, it is possible to use it in deducing the properties of programs; however,

* Department of Computer Science

What does it mean for a program to be correct?

In 1969, a seminal paper by Hoare introduced the following notation to specify what a program does:

$$\{ P \}\ \mathtt{C}\ \{ Q \}$$

- `C` is a program;
- P (the *precondition*) and Q (the *postcondition*) are statements on the program variables used in `C`.

We say that

$$\{ P \}\ \mathtt{C}\ \{ Q \} \text{ is true}$$

if whenever `C` is executed in a state satisfying P and if the execution of `C` terminates, then the state in which `C`'s execution terminates satisfies Q.

Floyd-Hoare logic?

Robert W. Floyd

ASSIGNING MEANINGS TO PROGRAMS[1]

Introduction. This paper attempts to provide an adequate basis for formal definitions of the meanings of programs in appropriately defined programming languages, in such a way that a rigorous standard is established for proofs about computer programs, including proofs of correctness, equivalence, and termination. The basis of our approach is the notion of an interpretation of a program: that is, an association of a proposition with each connection in the flow of control through a program, where the proposition is asserted to hold whenever that connection is taken. To prevent

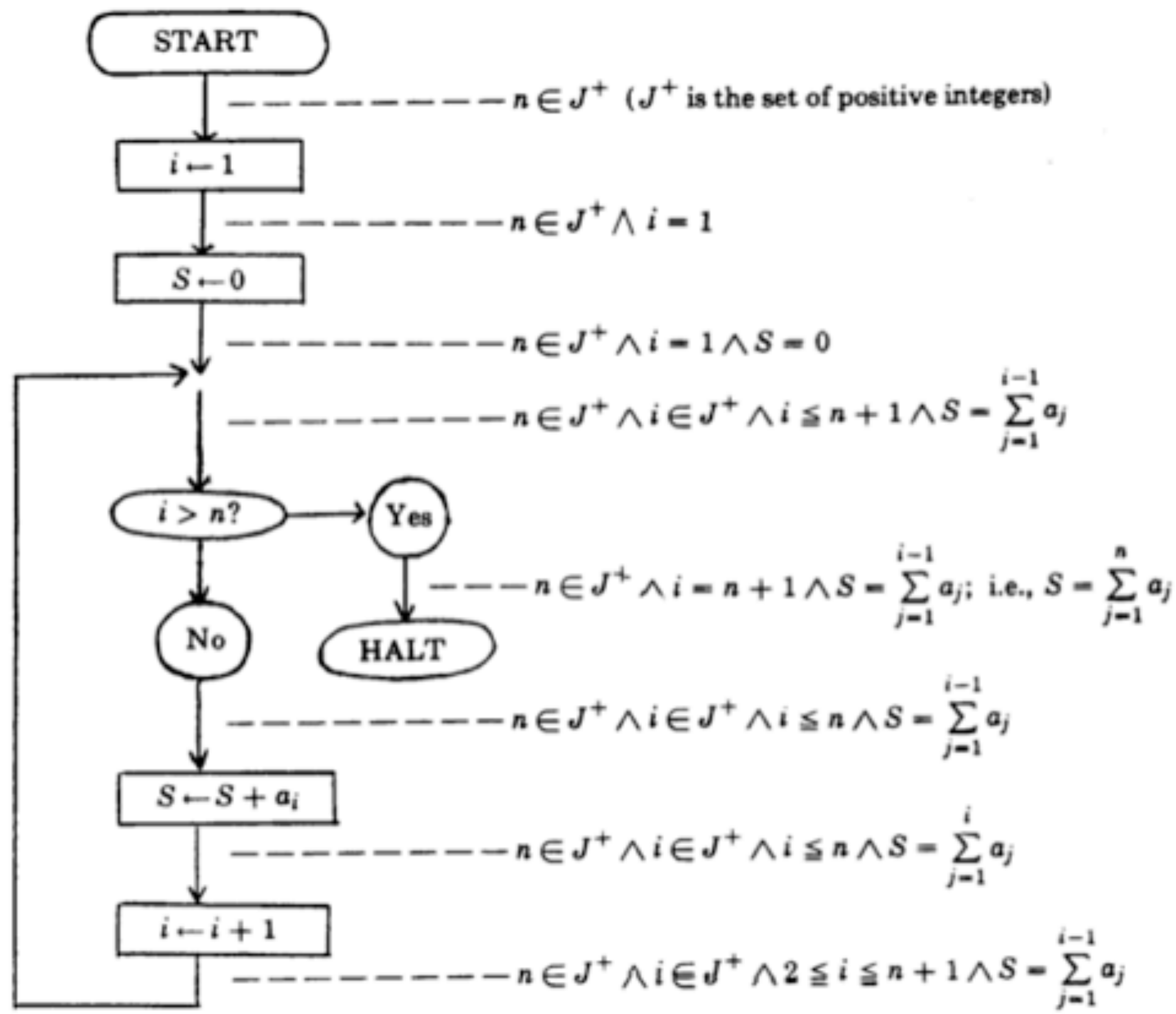


FIGURE 1. Flowchart of program to compute $S = \sum_{j=1}^{n} a_j$ $(n \geq 0)$

Note: the original ideas were seeded by the work of Robert Floyd, who in 1969 had published a similar system for flowcharts.

An imperative programming language

The symbol `S` stands for arbitrary *statements:* these are conditions like `x + 1 < y` which are either *true* or *false.*

The symbol `E` stands for arbitrary *expressions*: these are things like `x + 1` which denote values.

The symbol `C` stands for arbitrary *commands*, where a command is:

- do nothing: `skip`
- an assignment: `x := E`
- the sequential composition of two commands: C_1`;` C_2
- a conditional: `if S then` C_1 `else` C_2
- a loop: `while S do C`

Operational semantics

The computation state is represented with an environment called *stack*:

stack : var $\longrightarrow$ value (denoted s)

Evaluation of expressions and statements:

$$\frac{}{\texttt{3 / s} \longrightarrow \texttt{3}} \qquad \frac{}{\texttt{x / s} \longrightarrow \texttt{s(x)}} \qquad \frac{e_1 \texttt{ / s} \longrightarrow v_1 \quad e_2 \texttt{ / s} \longrightarrow v_2}{e_1 + e_2 \texttt{ / s} \longrightarrow v_1 + v_2}$$

etc...

Evaluation of commands:

$$\frac{\texttt{E / s} \longrightarrow \texttt{v}}{\texttt{x := E / s} \longrightarrow \texttt{skip / s[x:=v]}} \qquad \frac{}{\texttt{skip ; C / s} \longrightarrow \texttt{C / s}} \qquad \frac{C_1 \texttt{ / s} \longrightarrow \texttt{C' / s'}}{C_1 \texttt{ ; } C_2 \texttt{ / s} \longrightarrow \texttt{C' ; } C_2 \texttt{ / s'}}$$

$$\frac{\texttt{S / s} \longrightarrow \texttt{True}}{\texttt{if S then } C_1 \texttt{ else } C_2 \texttt{ / s} \longrightarrow C_1 \texttt{ / s}} \qquad \frac{\texttt{S / s} \longrightarrow \texttt{False}}{\texttt{if S then } C_1 \texttt{ else } C_2 \texttt{ / s} \longrightarrow C_2 \texttt{ / s}}$$

$$\frac{\texttt{S / s} \longrightarrow \texttt{True} \quad \texttt{C ; while S do C / s} \longrightarrow \texttt{C' / s'}}{\texttt{while S do C / s} \longrightarrow \texttt{C' / s'}} \qquad \frac{\texttt{S / s} \longrightarrow \texttt{False}}{\texttt{while S do C / s} \longrightarrow \texttt{skip / s}}$$

Statements

Statements are assertions on the state. For instance, consider:

P, Q	::=	T	true
	|	¬ P	negation
	|	P ∧ Q	conjuction
	|	P ∨ Q	disjunction
	|	P ⇒ Q	implication
	|	S	language statements

A state s *satisfies* an assertion P (or *P holds in* s), denoted $s \vDash P$, if

$s \vDash T$ always

$s \vDash \neg P$ iff $s \vDash P$ is false

$s \vDash P \wedge Q$ iff $s \vDash P$ and $s \vDash Q$

$s \vDash P \vee Q$ iff $s \vDash P$ or $s \vDash Q$

$s \vDash P \Rightarrow Q$ iff $s \vDash P$ implies $s \vDash Q$

$s \vDash S$ iff $S / s \longrightarrow \mathtt{true}$

relates assertions to program state

Examples

- $\{\, x = 1 \,\}$ `x := x + 1` $\{\, x = 2 \,\}$
- $\{\, x = 1 \,\}$ `y := x` $\{\, x = 1 \wedge y = 1 \,\}$
- $\{\, x = 1 \,\}$ `y := x` $\{\, y = 2 \,\}$ (this is clearly false)
- $\{\, x = n \wedge y = m \,\}$ `r := x; x := y; y := r` $\{\, x = m \wedge y = n \,\}$

  The variables `n` and `m` which do not occur in the command and are used to name the initial values of program variables `x` and `y`, are called auxiliary variables (or *ghost variables*).

- $\{\, x = n \wedge y = m \,\}$ `x := y; y := x` $\{\, x = m \wedge y = n \,\}$ (false)
- $\{\, P \,\}$ `C` $\{\, T \,\}$ (always true)
- $\{\, T \,\}$ `C` $\{\, Q \,\}$ (states that whenever `C` terminates, Q holds)

Partial vs. total correctness

An expression { P } `C` { Q } is called a *partial correctness specification*: { P } `C` { Q } can be true even if `C` does not terminate in a state satisfying P.

Total correctness specification: [P] `C` [Q] is true if and only if

(1) whenever `C` is executed in a state satisfying P, then the execution of `C` terminates;

(ii) after termination Q holds.

Informally: Total correctness = Termination + Partial correctness.

In all these lectures we will focus on partial correctness.

Floyd-Hoare logic: the assignment axiom

$$\vdash \{P[E/x]\}\ x := E\ \{P\}$$

Notation:
P where all occurrences of x have been substituted with E.

Examples:

$\vdash \{y = 2\}\ x := 2\ \{y = x\}$

$\vdash \{x + 1 = n + 1\}\ x := x + 1\ \{x = n + 1\}$

$\vdash \{E = E\}\ x := E\ \{x = E\}$ (if x does not occur in E)

Remark: the axiom as a backward flavour. Two common erroneus intuitions are that it should be as follows:

(a) $\vdash \{P\}\ x := E\ \{P[x/E]\}$

(b) $\vdash \{P\}\ x := E\ \{P[E/x]\}$

Exercise: the axioms (a) and (b) are unsound. Why?

(a) $\vdash$ {X=0} X:=1 {X=0}, since the (X=0)[X/1] is equal to (X=0) as 1 doesn't occur in (X=0).

(b) $\vdash$ {X=0} X:=1 {1=0} which follows by taking P to be X=0, V to be X and E to be 1.

Floyd-Hoare logic: weakening and strenghtening

$$\frac{\vdash P \Rightarrow P' \quad \vdash \{\, P' \,\}\; \mathtt{C}\; \{\, Q' \,\} \quad \vdash Q' \Rightarrow Q}{\vdash \{\, P \,\}\; \mathtt{C}\; \{\, Q \,\}}$$

Exercise: deduce the following facts:

$\vdash \{\, \mathtt{x = n} \,\}\ \mathtt{x := x + 1}\ \{\, \mathtt{x = n + 1} \,\}$

$\vdash \{\, \mathrm{T} \,\}\ \mathtt{x := E}\ \{\, \mathtt{x = E} \,\}$

$\vdash \{\, \mathtt{x = r} \,\}\ \mathtt{q := 0}\ \{\, \mathtt{x = r + (y * q)} \,\}$

Remember: $\vdash \{\, P\,[\, \mathtt{E} \,/\, \mathtt{x} \,] \,\}\ \mathtt{x := E}\ \{\, P \,\}$

Floyd-Hoare logic: statement manipulation

$$\frac{\vdash P \Rightarrow P' \quad \vdash \{\, P' \,\}\ \mathsf{c}\ \{\, Q' \,\} \quad \vdash Q' \Rightarrow Q}{\vdash \{\, P \,\}\ \mathsf{c}\ \{\, Q \,\}}$$

$$\frac{\vdash \{\, P \,\}\ \mathsf{c}\ \{\, Q_1 \,\} \quad \vdash \{\, P \,\}\ \mathsf{c}\ \{\, Q_2 \,\}}{\vdash \{\, P \,\}\ \mathsf{c}\ \{\, Q_1 \wedge Q_2 \,\}} \qquad \frac{\vdash \{\, P_1 \,\}\ \mathsf{c}\ \{\, Q \,\} \quad \vdash \{\, P_2 \,\}\ \mathsf{c}\ \{\, Q \,\}}{\vdash \{\, P_1 \vee P_2 \,\}\ \mathsf{c}\ \{\, Q \,\}}$$

Reminscent of sequent calculus...

Floyd-Hoare logic: commands

P is called *loop invariant*

$$\frac{\vdash \{\, P \wedge S \,\}\ \mathtt{C}\ \{\, P \,\}}{\vdash \{\, P \,\}\ \mathtt{while\ S\ do\ C}\ \{\, P \wedge \neg S \,\}}$$

$$\frac{\vdash \{\, P \,\}\ \mathtt{C_1}\ \{\, Q \,\} \quad \vdash \{\, Q \,\}\ \mathtt{C_2}\ \{\, R \,\}}{\vdash \{\, P \,\}\ \mathtt{C_1\ ;\ C_2}\ \{\, R \,\}}$$

$$\frac{\vdash \{\, P \wedge S \,\}\ \mathtt{C_1}\ \{\, Q \,\} \quad \vdash \{\, P \wedge \neg S \,\}\ \mathtt{C_2}\ \{\, Q \,\}}{\vdash \{\, P \,\}\ \mathtt{if\ S\ then\ C_1\ else\ C_2}\ \{\, Q \,\}}$$

Exercise: prove that

$\vdash \{\, T \,\}$

```
r := x; q := 0; while y ≤ r do ( r := r-y; q := q+1 )
```

$\{\, r < y \wedge x = r+(y*q) \,\}$

Remember: $\vdash \{\, P\,[\,E\,/\,x\,]\,\}\ \mathtt{x := E}\ \{\, P \,\}$

Exercise

The Zune's real-time clock stores the time in terms of days and seconds since January 1st, 1980. At the end of the boot sequence, it converts the clock value into date and time. This is the code that, given the number of days since January 1st, 1980, computes the year.

```
while (days > 365) {
    if (IsLeapYear(year)) {
        if (days > 366) {
            days -= 366;
            year += 1;
        }
    }
    else {
        days -= 365;
        year += 1;
    }
}
```

Is this code correct? Does it hold that

$\{$ `days > 0` $\wedge$ `year = 0` $\}$
code
$\{$ `days <= 365` $\wedge$ `year >= 0` $\}$

E

Th es t
Ja of t
int coc
Ja ne y

whi

Plenty of Zunes hang up on December 31st, 2008. They worked perfectly the day after.

How is it possible?

We just proved the code correct!

```
{ days <= 365 ∧ year >= 0 }
}
```

E

Th

Ja

int

Ja

es t

of t

coc

ne y

`whi`

Plenty of Zunes hang up on December 31st, 2008. They worked perfectly the day after.

How is it possible?

We just proved the code correct!

We proved only *partial correctness!*

```
{ days <= 365 ∧ year >= 0 }
```

```
}
```

Relating the initial and final state

Forget leap years for now, and consider a simplified version of the Zune code:

```
while (days > 365) {
     days -= 365;
     year += 1;
 }
```

How can we specify that, after executing the code, the expression

```
days + year * 365
```

is equal to the value of `days` before executing the code?

Relating the initial and final state

Forget leap years, and consider a simplified version of the Zune code:

```
olddays = days;
while (days > 365) {
    days -= 365;
    year += 1;
}
```

We need to introduce an *auxiliary* variable, `olddays`, to record some informations about a particular state of the program,

$$\{\, \texttt{days > 0} \wedge \texttt{year = 0} \,\}\ \textit{code}\ \{\, \texttt{days + year * 365 = olddays} \,\}$$

Remark: the extra assignments must not change the semantics of the program. A set X is auxiliary for `c` if each free occurrence in `c` of an identifier from X is in an assignment whose target is in X: no effect on control flow, no effect on other variables.

Soundness of Floyd-Hoare logic

Imagine you can derive { P } `c` { Q } for some command `c` and statements P and Q. What does this assert on the execution of C in some state s?

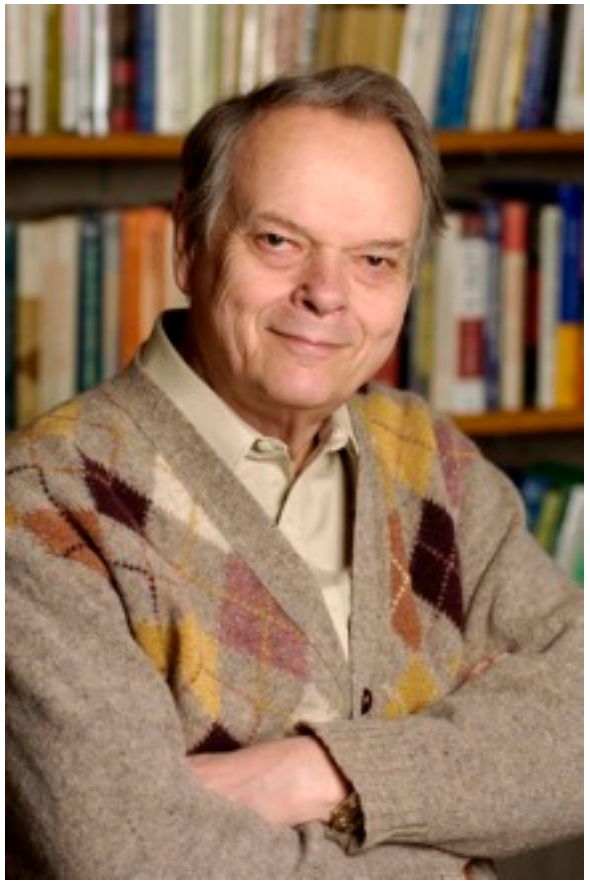
Soundness: Let $\vdash$ { P } `c` { Q } a provable triple.

For all states `s`, `s` $\models$ P and `c` / `s` $\longrightarrow$ `skip` / `s'` imply `s'` $\models$ Q .

Exercise: what about completeness? Is it true that if for all states `s`, `s` $\models$ P and `c` / `s` $\longrightarrow$ `skip` / `s'` imply `s'` $\models$ Q, then $\vdash$ { P } `c` { Q } is provable?

Hint: what does the triple { P } `c` { $\neg$T } state?

Separation logic



Adding the heap

We extend our programming language with

- memory writes, `[E₁] := E₂`
- memory reads, `x := [E]`
- memory allocation, `x := cons(E₁,…,Eₙ)`
- memory deallocation, `dispose E`

The state is now represented by a pair (stack, heap), denoted (`s`,`h`), where

stack : var -> value

heap : loc -> value

where loc $\subseteq$ value.

Operational semantics

$$\frac{E \;/\; s \longrightarrow v}{x := E \;/\; (s,h) \longrightarrow skip \;/\; (s[x{:=}v],h)}$$

$$\frac{E \;/\; s \longrightarrow v}{x := [E] \;/\; (s,h) \longrightarrow skip \;/\; (s[x{:=}h(v)],\ h)}$$

$$\frac{E_1 \;/\; s \longrightarrow v_1 \qquad E_2 \;/\; s \longrightarrow v_2}{[E_1] := E_2 \;/\; (s,h) \longrightarrow skip \;/\; (s,\ h[v_1{:=}v_2])}$$

$$\frac{E_1 \;/\; s \longrightarrow v_1 \quad \ldots \quad E_n \;/\; s \longrightarrow v_n \qquad v \ldots v{+}(n{-}1) \notin dom(h)}{x := cons(E_1,\ldots,E_n) \;/\; (s,h) \longrightarrow skip \;/\; (s[x{:=}v],\ h \oplus [v{:=}v_1 \ldots v{+}(n{-}1){:=}v_n])}$$

$$\frac{E \;/\; s \longrightarrow v}{dispose\ E \;/\; (s,h) \longrightarrow skip \;/\; (s,h\backslash v)}$$

The other rules are straightforward.

Remark: `h[v:=v']` and `h\v` are defined only if $v \in dom(h)$.

Remark: the operational semantics is *stuck* if accesses outside the domain of `s` and `h` are performed.

Example program

```
x = cons(3,3); y = cons(4,4); [x+1] = y; [y+1] = x
```

stack heap

Example program

```
x = cons(3,3); y = cons(4,4); [x+1] = y; [y+1] = x
```

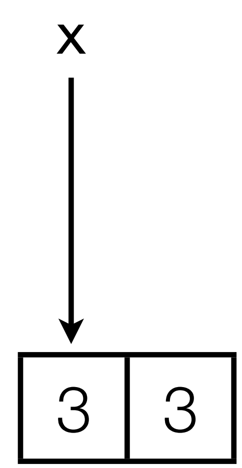
stack

x	43

heap

43	3
44	3

graphically



Example program

```
x = cons(3,3); y = cons(4,4); [x+1] = y; [y+1] = x
```

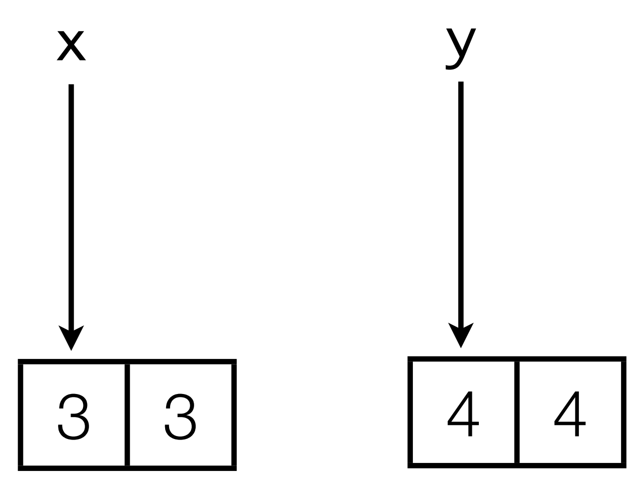
stack

x	43
y	57

heap

43	3
44	3
57	4
58	4

graphically



Example program

```
x = cons(3,3); y = cons(4,4); [x+1] = y; [y+1] = x
```

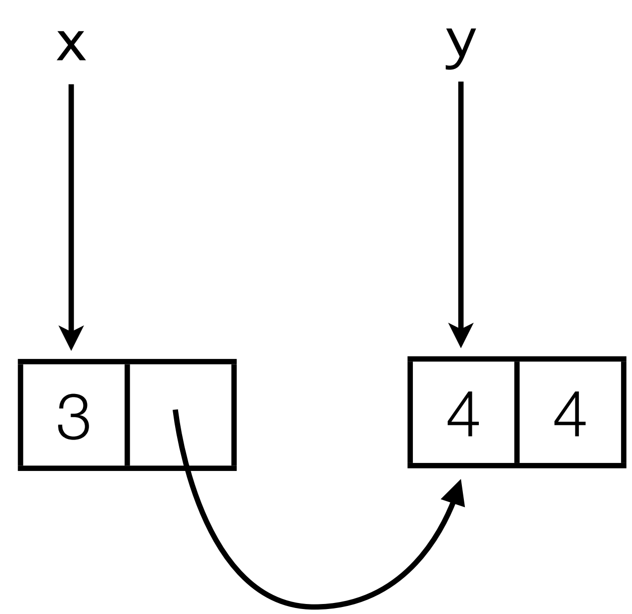
stack

x	43
y	57

heap

43	3
44	57
57	4
58	4

graphically



Example program

```
x = cons(3,3); y = cons(4,4); [x+1] = y; [y+1] = x
```

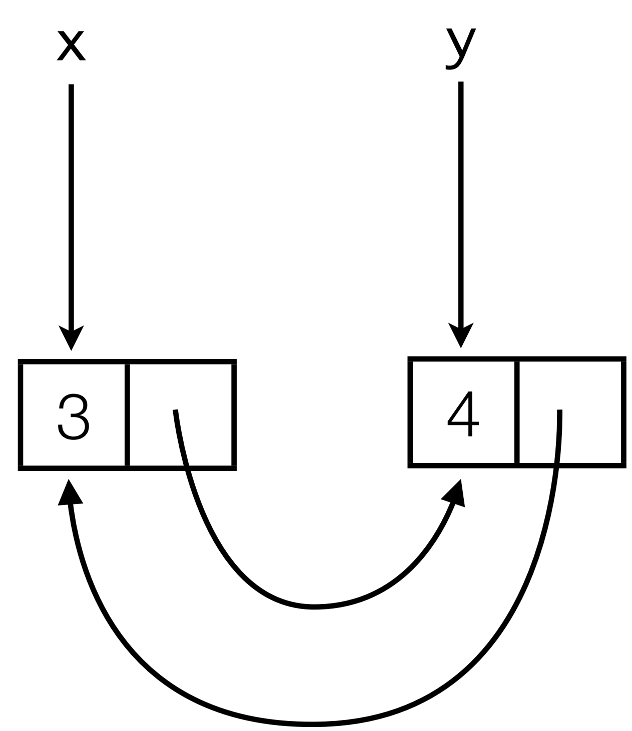
stack

x	43
y	57

heap

43	3
44	57
57	4
58	43

graphically



Why separation logic?

Can you suggest a precondition such that this triple holds?

```
⊢ { ??? }

      [y] := 4;
      [z] := 5;

   { [y] != [z] }
```

Why separation logic?

Can you suggest a precondition such that this triple holds?

```
⊢ { y != z }

      [y] := 4;
      [z] := 5;

  { [y] != [z] }
```

We need to assume that the locations pointed by `y` and `z` are different (*aliasing*).

Why separation logic?

And now?

```
⊢ { ??? }

      [y] := 4;
      [z] := 5;

  { [y] != [z] ∧ [x] = 3 }
```

Why separation logic?

And now?

$$\vdash \{\, y \mathrel{!=} z \wedge x \mathrel{!=} y \wedge x \mathrel{!=} z \wedge [x] = 3 \}$$

```
[y] := 4;
[z] := 5;
```

$$\{\, [y] \mathrel{!=} [z] \wedge [x] = 3 \,\}$$

- we need to assume that the locations pointed by y and z are different (*aliasing*).
- we need to know when things stay the same.

Framing

We want a general concept of things not being affected.

$$\frac{\{\,P\,\}\ \mathtt{C}\ \{\,Q\,\}}{\{\ \mathtt{[x]\ =\ 3} \wedge P\,\}\ \mathtt{C}\ \{\,Q \wedge \mathtt{[x]\ =\ 3}\ \}}$$

What are the conditions on `C` and `[x] = 3`?

These are very hard to define if reasoning about a heap and aliasing.

This is where separation logic comes in:

$$\frac{\{\,P\,\}\ \mathtt{C}\ \{\,Q\,\}}{\{\,R * P\,\}\ \mathtt{C}\ \{\,Q * R\,\}}$$

The new connective * is used to separate the heap.

In the beginning: classical logic

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta} \wedge r \qquad \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \wedge l \qquad \frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\ weakl$$

$$\frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta} \vee l \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta} \vee r \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, A}\ weakr$$

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \Rightarrow B \vdash \Delta} \Rightarrow l \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta} \Rightarrow r \qquad \frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}\ contrl$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta} \neg l \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta} \neg r \qquad \frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta}\ contrr$$

$$\frac{}{A \vdash A}\ BS$$

In the beginning: classical logic

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta} \wedge r \qquad \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \wedge l \qquad \frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\ weakl$$

$$\frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta} \vee l \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta} \vee r \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, A}\ weakr$$

$$\frac{\Gamma \vdash A, \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \Rightarrow B \vdash \Delta} \Rightarrow l \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta} \Rightarrow r \qquad \frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}\ contrl$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta} \neg l \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta} \neg r \qquad \frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta}\ contrr$$

$$\frac{}{A \vdash A}\ BS$$

A substructural logic: bunched implications

Idea: $\wedge$ admits weakening and contraction, but * does not.

We have:

$$\frac{\Delta(\Gamma) \vdash \psi}{\Delta(\Gamma \wedge \Gamma') \vdash \psi} \qquad \frac{\Delta(\Gamma \wedge \Gamma) \vdash \psi}{\Delta(\Gamma) \vdash \psi}$$

But we do not have:

$$\frac{\Delta(\Gamma) \vdash \psi}{\Delta(\Gamma * \Gamma') \vdash \psi} \qquad \frac{\Delta(\Gamma * \Gamma) \vdash \psi}{\Delta(\Gamma) \vdash \psi}$$

The *logic of bunched implications* (BI) mixes substructural logic with classical/intuitionistic logic. BI is the logic behing separation logic.

If this does not make sense, don't panic.

Statements of separation logic

P, Q	::=	T	true
	|	¬ P	negation
	|	$P \wedge Q$	conjuction
	|	$P \vee Q$	disjunction
	|	$P \Rightarrow Q$	implication
	|	S	language statements
	|	P * Q	separating conjunction
	|	$E_1 \longmapsto E_2$	points to
	|	empty	empty heap

$(s,h) \models \text{empty}$ iff $\text{dom}(h) = \varnothing$

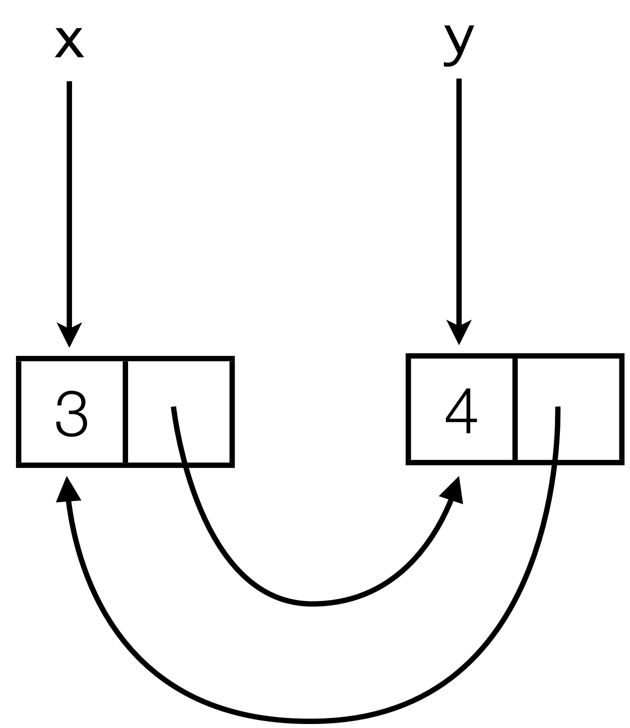
$(s,h) \models E_1 \longmapsto E_2$ iff $E_1 \;/\; s \longrightarrow v_1 \wedge E_2 \;/\; s \longrightarrow v_2 \wedge \text{dom}(h) = v_1 \wedge h(v_1) = v_2$

$(s,h) \models P * Q$ iff

$\exists\, h_1, h_2.\ \text{dom}(h_1) \cap \text{dom}(h_2) = \varnothing \wedge h_1 \oplus h_2 = h \wedge (s,h_1) \models P \wedge (s,h_2) \models Q$

Example

Our previous heap



satisfies the statement: $(x \mapsto 3) * (x{+}1 \mapsto y) * (y \mapsto 4) * (y{+}1 \mapsto x)$,

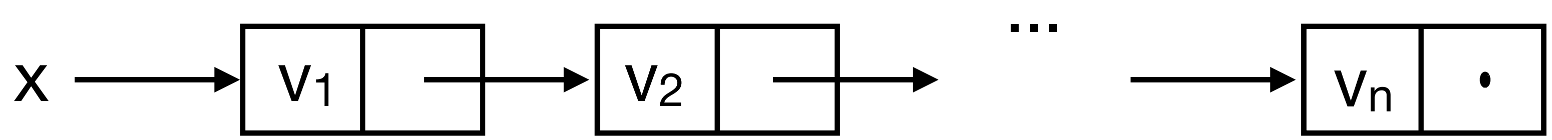
but not the statement: $x \mapsto 3$.

Exercise: does the heap above satisfy

$$(x \mapsto 3 * x{+}1 \mapsto y) \land (y \mapsto 4 * y{+}1 \mapsto x)\ ?$$

Data types: list

A non-cyclic list



can be defined by the following *recursive* statement:

$$\text{list } [\,]\ \mathtt{x} \equiv \text{empty} \wedge \mathtt{x = nil}$$

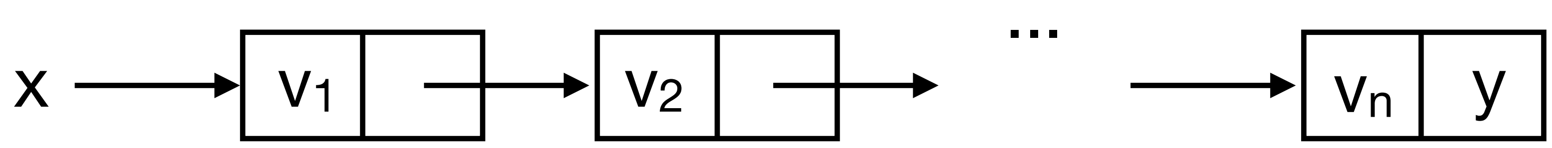
$$\text{list } v_1{::}\alpha\ x \equiv \exists\, \mathtt{j}.\ \mathtt{x} \longmapsto v_1 * (x{+}1 \longmapsto \mathtt{j}\,) * \text{list } \alpha\ \mathtt{j}$$

Example: list $v_{1::\ldots::}v_n$ $\mathtt{x}$ is satisfied by an heap where $\mathtt{x}$ points to a list whose content is $v_{1::\ldots::}v_n$.

Remark: we have (implicitely) added sequences (ranged over by α) to the logic.

Data types: list segment

Often it is useful to be able to denote *list segments*:



$$\mathrm{lseg}\ [\,]\ (x,y) \equiv \mathrm{empty} \wedge x = y$$

$$\mathrm{lseg}\ v{::}\alpha\ (x,y) \equiv \exists\, j.\ x \mapsto v * (x{+}1 \mapsto j) * \mathrm{lseg}\ \alpha\ (j,y)$$

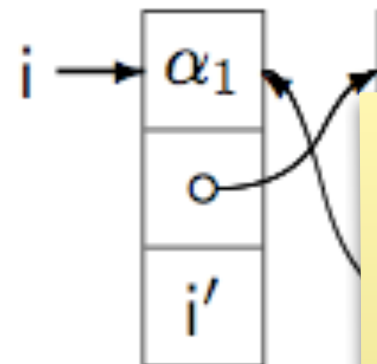
Exercise: prove, by structural induction on α, that:

$$\mathrm{lseg}\ \alpha \cdot \beta\ (x,y) \Leftrightarrow \exists\, j.\ \mathrm{lseg}\ \alpha\ (x,j) * \mathrm{lseg}\ \beta\ (j,y)$$

where · denotes concatenation of sequences.

Exercises

Exercise: can you write a statement that encodes doubly-linked lists?



dlsegε(i,i’,j’,j) = empty $\wedge$ i=j’ $\wedge$ i’=j

dlseg(a·α)(i,i’,j’,j) = $\exists$k.i$\rightarrow$a,k,i’ $*$ dlsegα(k,i,j’,j)

and consider the definition of doubly-linked lists below:
dlsα(f,b) = dlsegα(f,null,null,b)

Exercise: which data structure i

gue

guess (τ , τ′) i ≡ ∃ j, k. i

(Local) axioms

Here are three of the axioms:

- *write:* $\{ \texttt{E} \mapsto \_ \}$ `[E] = E'` $\{ \texttt{E} \mapsto \texttt{E'} \}$

- *dispose:* $\{ \texttt{E} \mapsto \_ \}$ `dispose(E)` { empty }

- *alloc:* { empty } `x = cons(E1,…,En)` $\{ \texttt{x} \mapsto \texttt{E1} * \texttt{x+1} \mapsto \texttt{E2} * \ldots * \texttt{x+(n-1)} \mapsto \texttt{En} \}$

where $\texttt{E} \mapsto \_$ is a shorthand for $\exists\, \texttt{x}.\ \texttt{E} \mapsto \texttt{x}$.

The frame rule

The most important rule, called *the frame rule*:

$$\frac{\{\ P\ \}\ \ \mathtt{C}\ \ \{\ Q\ \}}{\{\ P * R\ \}\ \ \mathtt{C}\ \ \{\ Q * R\ \}}$$

provided that $fv(R) \cap \text{modifies}(\mathtt{C}) = \emptyset$

Note: modifies(C) denotes the set of stack variables assigned by a given command, C, e.g. modifies(x=3) = {x}. However assignment through a stack variable to the heap is not counted: modifies([x]=3) = ∅. See the references for full definition.

Exercice: show that $\{\ P\ \}\ \mathtt{C}\ \{\ Q\ \} \Rightarrow \{\ P \wedge R\ \}\ \mathtt{C}\ \{\ Q \wedge R\ \}$ is not sound.

Exercises

Prove that:

{ lseg α (i,j) } `k := cons(a,i); i := k` { lseg $a \cdot \alpha$ (i,j) }

{ lseg α (i,j) * $j \mapsto a,k$ } `l := cons(b,k); [j+1] := l` { lseg $\alpha \cdot a \cdot b$ (i,k) }

{ lseg $a \cdot \alpha$ (i,k) } `j := [i+1]; dispose i; dispose i+1; i := j` { lseg α (i,k) }

Remember:

lseg [] (x,y) ≡ empty $\wedge$ x = y

lseg v::α (x,y) ≡ $\exists$ j. x $\longmapsto$ v $*$ (x+1 $\longmapsto$ j) $*$ lseg α (j,y)

Notation: $j \mapsto a,k$ stands for $j \mapsto a$ * j+1 $\longmapsto$ k.





Thanks to:

Mike Gordon

John Reynolds

Tony Hoare

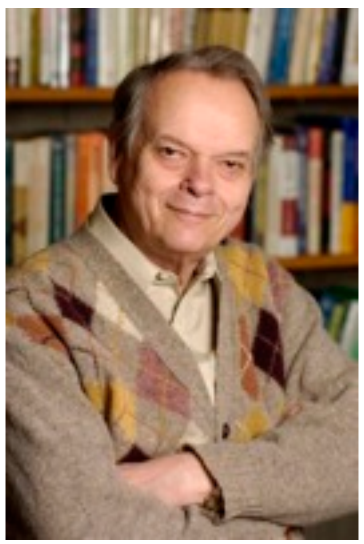
Maged Michael

Peter O'Hearn

Robert Floyd

Doug Lea

Robin Milner



Exercise: associate each picture with its owner….

References:

Mike Gordon, *Specification and Verification I*, chapters 1 and 2.

John Reynolds, *Introduction to Separation Logic*, parts 1-4.

both available from http://moscova.inria.fr/~zappa/teaching/mpri/2010/ .

Next lecture: and concurrency?