

# Separation Logic, Heap-Manipulating Programs, and Concurrency

## Lecture 4: Racy concurrency

Étienne Lozes

ENS Cachan & RWTH Aachen

june 17th, 2010

## Reminder

- ▶ a lock on every node
- ▶ the lock protects the next cell (see proof in previous lecture).
- ▶ fine-grained concurrency, and race-free

## Reminder

- ▶ a lock on every node
- ▶ the lock protects the next cell (see proof in previous lecture).
- ▶ fine-grained concurrency, and race-free

## Why we may want to go further

- ▶ taking locks is time-consuming
- ▶ we observed that, in case of GC, racy implementation may still be fine if the updates occurred on disjoint parts of the heap.

## Lazy lock-coupling

- ▶ No locking while searching the desired position
- ▶ Locking is required only for updates

```
cell *search(int i)
cell *p=head;
while(p->next->val<v)
    p=p->next;
end;
```

```
bool is_in(int i)
    cell *p=search(i);
    cell *n=p->next;
return n->val==v
```

## Lazy lock-coupling

- ▶ No locking while searching the desired position
- ▶ Locking is required only for updates
- ▶ Marking elements for deletion will be essential for correctness

```
cell *search(int i)
cell *p=head;
while(p->next->val<v)
    p=p->next;
end;
```

```
bool is_in(int i)
    cell *p=search(i);
    cell *n=p->next;
return n->val==v & ! n->marked;
```

## Lazy lock-coupling: removal

---

```
#def validate(p,n) !p->marked & !n->marked &
n==p->next

void remove(int i){
    while true{ // transaction loop
        cell *p=search(i);
        cell *n=p->next;
        lock(p->lock)
        lock(n->lock)
        if validate(p,n){ // transaction is successful
            n->marked=true; // logical removal
            p->next=n->next; // physical removal
        }
        break;}
    unlock(p->lock);
    unlock(n->lock);
}
```

## Lazy lock-coupling: insertion

```
cell *add(int i)
while !done do
  cell *p=search(i)
  cell *n=p->next;
  lock(p->lock);
  lock(n->lock);
  if validate(p,n){
    cell c:= cons(i,newlock(),false,n);
    p->next:=c;
  }
  unlock(p->lock);
  unlock(n->lock);
}
```

## Lazy lock-coupling : some questions

- ▶ insertions and removals may require several traversals: what guarantees of progress do we have? and what do we want?

## Lazy lock-coupling : some questions

- ▶ insertions and removals may require several traversals: what guarantees of progress do we have? and what do we want?
- ▶ safety (absence of memory violation) is not that obvious...
- ▶ ... and correctness (it actually implements a multiset) neither

1. A taste of some notions of correctness
2. A taste of some notions of progress
3. A taste of how to prove safety

# Linearizability



## Treiber's concurrent stack

```
atomic bool CAS(a,b,c){  
if (*a==b) {*a=c;return true;} return false;}
```

## Treiber's concurrent stack

```
atomic bool CAS(a,b,c){  
if (*a==b) {*a=c;return true;} return false;}
```

```
void push(value v) {  
    cell *t, *x;  
    x = alloc();  
    x->data = v;  
    do {  
        t = top_ptr;  
        x->next = t;  
    } while (!CAS(&top_ptr,t,x));  
}  
  
value pop() {  
    cell *t, *x;  
    do {  
        t = top_ptr;  
        if (t == NULL) return EMPTY;  
        x = t->next;  
    }  
    while (!CAS(&top_ptr,t,x));  
    return t->data;  
}
```

## Trying to get rid of the GC

---

```
void push(value v) {
    cell *t, *x;
    x = alloc();
    x->data = v;
    do {
        t = top_ptr;
        x->next = t;
    } while (!CAS(&top_ptr,t,x));
}

value pop() {
    cell *t, *x;
    do {
        t = top_ptr;
        if (t == NULL) return EMPTY;
        x = t->next;
    }
    while (!CAS(&top_ptr,t,x));
    value data = t->data;
    free(t);
    return data;
}
```

## Trying to get rid of the GC

```
void push(value v) {
    cell *t, *x;
    x = alloc();
    x->data = v;
    do {
        t = top_ptr;
        x->next = t;
    } while (!CAS(&top_ptr,t,x));
}

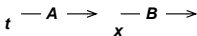
value pop() {
    cell *t, *x;
    do {
        t = top_ptr;
        if (t == NULL) return EMPTY;
        x = t->next;
    }
    while (!CAS(&top_ptr,t,x));
    value data = t->data;
    free(t);
    return data;
}
```

**Exercise:** find an execution with a memory violation

## The ABA problem, why GC matters

1. start with stack A::B::nil.
2. Thread 1 starts popping, then is **preempted**

```
5. value pop() {
    cell *t, *x;
    do {
        t = top_ptr;
        if (t == NULL) return EMPTY;
        x = t->next; (1 STOP)
    }
    while (!CAS(&top_ptr,t,x));
    data_t data = t->data;
    free(t);
    return data;
}
```



## The ABA problem, why GC matters

1. start with stack A::B::nil.
2. Thread 1 starts popping, then is **preempted**
3. Thread 2 does pop:A (t of 1 is now freed)

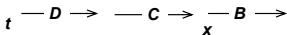
```
5. value pop() {
    cell *t, *x;
    do {
        t = top_ptr;
        if (t == NULL) return EMPTY;
        x = t->next; (1 STOP)
    }
    while (!CAS(&top_ptr,t,x));
    data_t data = t->data;
    free(t);
    return data;
}
```

t             $\xrightarrow{B}$

x

## The ABA problem, why GC matters

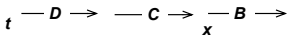
1. start with stack A::B::nil.
2. Thread 1 starts popping, then is **preempted**
3. Thread 2 does pop:A
4. Thread 2 does push(C);push(D) and allocates D at the t of 1



```
5. value pop() {
    cell *t, *x;
    do {
        t = top_ptr;
        if (t == NULL) return EMPTY;
        x = t->next; (1 STOP)
    }
    while (!CAS(&top_ptr,t,x));
    data_t data = t->data;
    free(t);
    return data;
}
```

## The ABA problem, why GC matters

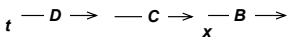
1. start with stack A::B::nil.
2. Thread 1 starts popping, then is **preempted**
3. Thread 2 does pop:A
4. Thread 2 does push(C);push(D) and allocates D at the t of 1
5. Thread 1 recovers, test succeeds



```
value pop() {
    cell *t, *x;
    do {
        t = top_ptr;
        if (t == NULL) return EMPTY;
        x = t->next; (1 STOP)
    }
    while (!CAS(&top_ptr,t,x));
    data_t data = t->data;
    free(t);
    return data;
}
```

## The ABA problem, why GC matters

1. start with stack A::B::nil.
2. Thread 1 starts popping, then is **preempted**
3. Thread 2 does pop:A
4. Thread 2 does push(C);push(D) and allocates D at the t of 1
5. Thread 1 remove C,D at once pop:D, then does pop:B



```
value pop() {
    cell *t, *x;
    do {
        t = top_ptr;
        if (t == NULL) return EMPTY;
        x = t->next; (1 STOP)
    }
    while (!CAS(&top_ptr,t,x));
    data_t data = t->data;
    free(t);
    return data;
}
```

## The ABA problem, why GC matters

1. start with stack A::B::nil.
2. Thread 1 starts popping, then is **preempted**
3. Thread 2 does pop:A
4. Thread 2 does push(C);push(D) and allocates D at the t of 1
5. Thread 1 remove C,D at once pop:D, then does pop:B

```
value pop() {
    cell *t, *x;
    do {
        t = top_ptr;
        if (t == NULL) return EMPTY;
        x = t->next; (1 STOP)
    }
    while (!CAS(&top_ptr,t,x));
    data_t data = t->data;
    free(t);
    return data;
}
```

(pop:A;push(C);push(D)) || (pop:D;pop:B)

This is not a stack's behavior!

## Towards a formalization

### **Intuition to be formalized**

concurrent method calls, even if they overlap in time, are expected to behave as if non-overlapping, rescheduled in some order that does not conflict with observables.

# Towards a formalization

## **Intuition to be formalized**

concurrent method calls, even if they overlap in time, are expected to behave as if non-overlapping, rescheduled in some order that does not conflict with observables.

## **What should be made precise now**

- ▶ what is meant with *observables*

# Towards a formalization

## **Intuition to be formalized**

concurrent method calls, even if they overlap in time, are expected to behave as if non-overlapping, rescheduled in some order that does not conflict with observables.

## **What should be made precise now**

- ▶ what is meant with *observables*
- ▶ what is meant by *re-scheduled*

## Intuition to be formalized

concurrent method calls, even if they overlap in time, are expected to behave as if non-overlapping, rescheduled in some order that does not conflict with observables.

## What should be made precise now

- ▶ what is meant with *observables*
- ▶ what is meant by *re-scheduled*
- ▶ how to model a method call

## Traces with call and return: "histories"

### Local events are

- ▶ either function call: `push(A)`, `insert(5)`, `deq()`...
- ▶ or function return: `enq:void`, `pop:B`, ...

## Traces with call and return: "histories"

### Local events are

- ▶ either function call: `push(A)`, `insert(5)`, `deq()`...
- ▶ or function return: `enq:void`, `pop:B`, ...

Global events are : pair [*ThreadId*, *evt*]

## Traces with call and return: "histories"

### Local events are

- ▶ either function call: push(A), insert(5), deq()...
- ▶ or function return: enq:void, pop:B, ...

Global events are : pair [*ThreadId*, *evt*]

**History:** a sequence of global events

## Traces with call and return: "histories"

### Local events are

- ▶ either function call: `push(A)`, `insert(5)`, `deq()`...
- ▶ or function return: `enq:void`, `pop:B`, ...

**Global events are** : pair [*ThreadId*, *evt*]

**History**: a sequence of global events

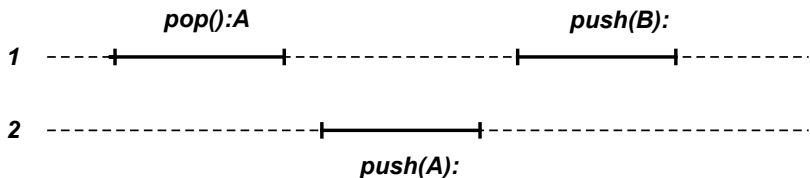
### Sequential history

every function call is immediately followed by its return (except possibly the last call).

## Graphical representation

The sequential history

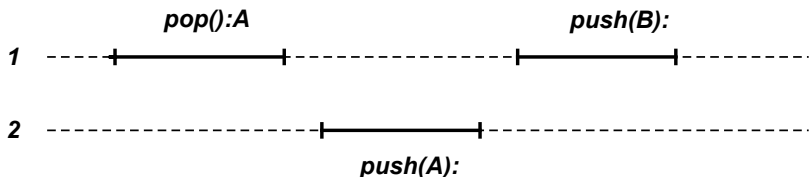
$[1, pop()]$   $[1, pop : A]$   $[2, push(A)]$   $[2, push : ]$   $[1, push(B)]$   $[1, push : ]$



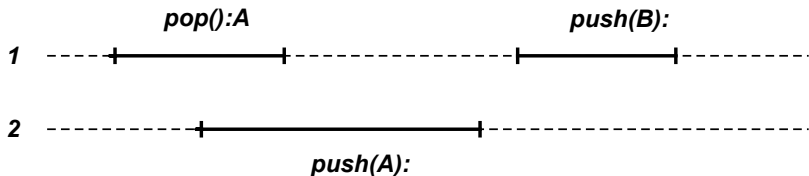
## Graphical representation

The sequential history

$[1, pop()]$   $[1, pop : A]$   $[2, push(A)]$   $[2, push : ]$   $[1, push(B)]$   $[1, push : ]$



A non sequential history



## Correctness parameterized by simulation

### **Correctness of a sequential history**

A sequential history is correct if it corresponds to what would be observed using such a scheduling

## Correctness parameterized by simulation

### **Correctness of a sequential history**

A sequential history is correct if it corresponds to what would be observed using such a scheduling

Tautological: the specification of a correct behavior is defined by the code itself.

## Correctness parameterized by simulation

---

### Correctness of a sequential history

A sequential history is correct if it corresponds to what would be observed using such a scheduling

Tautological: the specification of a correct behavior is defined by the code itself.

### Correctness of a non-sequential history

$H$  is correct if there is a sequential history  $H'$  that *simulates*  $H$  ( $H' \prec H$ ) and that is correct.

# Correctness parameterized by simulation

---

## Correctness of a sequential history

A sequential history is correct if it corresponds to what would be observed using such a scheduling

Tautological: the specification of a correct behavior is defined by the code itself.

## Correctness of a non-sequential history

$H$  is correct if there is a sequential history  $H'$  that *simulates*  $H$  ( $H' \prec H$ ) and that is correct.

## Correctness of a concurrent data structure

For all client code and for all scheduling, the observed history is correct.

## Serializability

We need to define what “*simulates*” means.

## Serializability

We need to define what “*simulates*” means.

Different notions of simulation, different notions of correctness.

We need to define what “*simulates*” means.

Different notions of simulation, different notions of correctness.

**The most relaxed one: serializability**

$H' \prec H$  if  $H'$  is a permutation of events of  $H$

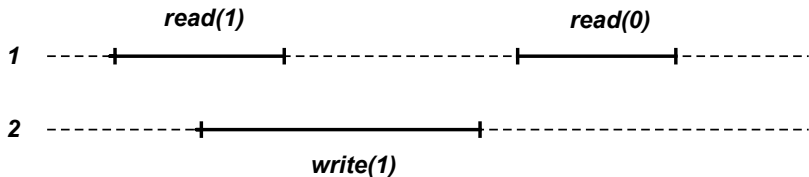
We need to define what “*simulates*” means.

Different notions of simulation, different notions of correctness.

**The most relaxed one: serializability**

$H' \prec H$  if  $H'$  is a permutation of events of  $H$

Good if each thread executes at most one operation, otherwise may authorize very strange behaviors:



## Sequential consistency [Lamport]

**Intuitively:** don't reorder events of the same thread.

## Sequential consistency [Lamport]

**Intuitively:** don't reorder events of the same thread.

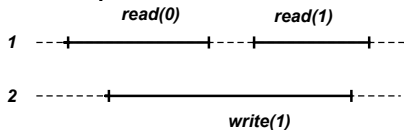
**Formally:**  $H \prec_{seq} H'$  if  $H \downarrow t = H' \downarrow t$  for all thread  $t$ .

## Sequential consistency [Lamport]

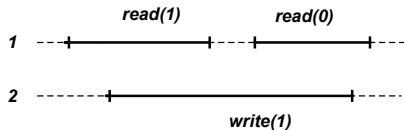
**Intuitively:** don't reorder events of the same thread.

**Formally:**  $H \prec_{seq} H'$  if  $H \downarrow t = H' \downarrow t$  for all thread  $t$ .

### Examples

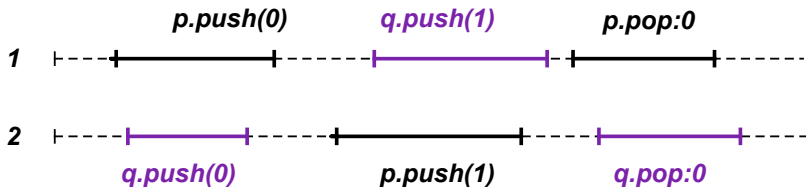


*Sequentially consistent*



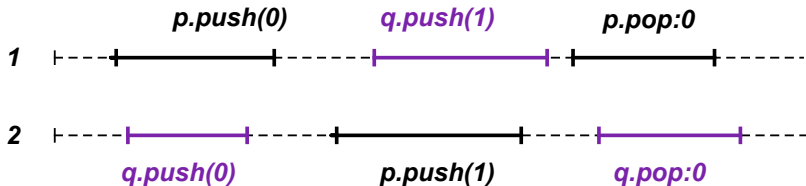
*Not Sequentially consistent*

SC is not compositional:



$H \downarrow p, H \downarrow q$  are SC, but not  $H$

SC is not compositional:



$H \downarrow p, H \downarrow q$  are SC, but not  $H$

Moreover:

- ▶ sometimes too restrictive (for some optimizations)
- ▶ sometimes not restrictive enough (like here).

## Linearizability [Herlihy & Wing]

---

**Intuitively:** don't reorder intervals that don't overlap

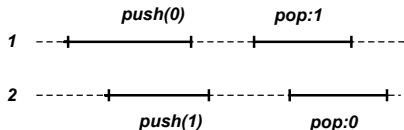
**Formally:**  $H \prec_{lin} H'$  if for all intervals  $I, I'$ , if  $end(I) < init(I')$  in  $H'$ , then so does it in  $H$ .

# Linearizability [Herlihy & Wing]

**Intuitively:** don't reorder intervals that don't overlap

**Formally:**  $H \prec_{lin} H'$  if for all intervals  $I, I'$ , if  $end(I) < init(I')$  in  $H'$ , then so does it in  $H$ .

## Examples



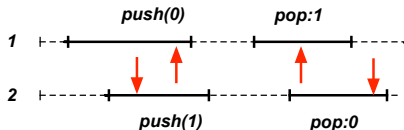
*Linearizable*

# Linearizability [Herlihy & Wing]

**Intuitively:** don't reorder intervals that don't overlap

**Formally:**  $H \prec_{lin} H'$  if for all intervals  $I, I'$ , if  $end(I) < init(I')$  in  $H'$ , then so does it in  $H$ .

## Examples



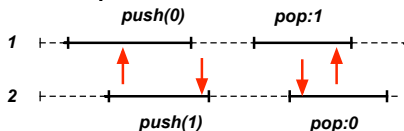
*Linearizable*

# Linearizability [Herlihy & Wing]

**Intuitively:** don't reorder intervals that don't overlap

**Formally:**  $H \prec_{lin} H'$  if for all intervals  $I, I'$ , if  $end(I) < init(I')$  in  $H'$ , then so does it in  $H$ .

## Examples



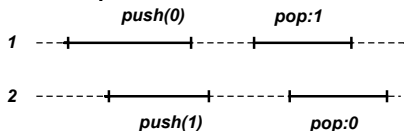
*Linearizable*

# Linearizability [Herlihy & Wing]

**Intuitively:** don't reorder intervals that don't overlap

**Formally:**  $H \prec_{lin} H'$  if for all intervals  $I, I'$ , if  $end(I) < init(I')$  in  $H'$ , then so does it in  $H$ .

## Examples



*Linearizable*



*Not linearizable*

## Quiescent consistency

**Intuitively:** inactive times are synchronizing times, can't reorder across

**Examples:**



Serializability, Sequential Consistency, Linearizability,  
Quiescent Consistency...

Serializability, Sequential Consistency, Linearizability,  
Quiescent Consistency...

1. which notions of simulation are equivalence relations?

Serializability, Sequential Consistency, Linearizability,  
Quiescent Consistency...

1. which notions of simulation are equivalence relations?  
*serializing, sequentializing, quiescent preserving, but not  
linearizing*

Serializability, Sequential Consistency, Linearizability,  
Quiescent Consistency...

1. which notions of simulation are equivalence relations?  
*serializing, sequentializing, quiescent preserving, but not linearizing*
2. which correctness conditions are compositional?

Serializability, Sequential Consistency, Linearizability,  
Quiescent Consistency...

1. which notions of simulation are equivalence relations?  
*serializing, sequentializing, quiescent preserving, but not linearizing*
2. which correctness conditions are compositional?  
*serializability, linearizability, quiescent consistency, but not sequential consistency*

Serializability, Sequential Consistency, Linearizability,  
Quiescent Consistency...

1. which notions of simulation are equivalence relations?  
*serializing, sequentializing, quiescent preserving, but not linearizing*
2. which correctness conditions are compositional?  
*serializability, linearizability, quiescent consistency, but not sequential consistency*
3. how do these correctness notions relate to each other?

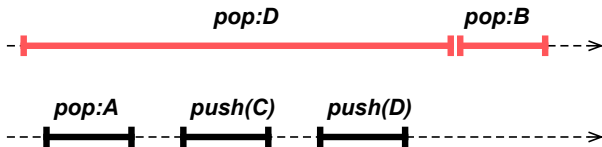
Serializability, Sequential Consistency, Linearizability,  
Quiescent Consistency...

1. which notions of simulation are equivalence relations?  
*serializing, sequentializing, quiescent preserving, but not linearizing*
2. which correctness conditions are compositional?  
*serializability, linearizability, quiescent consistency, but not sequential consistency*
3. how do these correctness notions relate to each other?  
*serializability is the weakest, linearizability is the strongest, and SC and QC are incomparable.*

## Back to ABA problem for the bugged stack

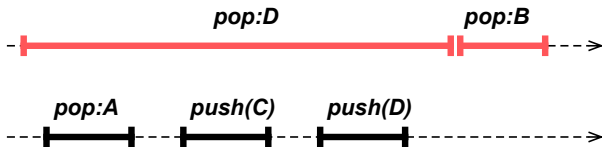
---

(1,pop())(2,pop())(2,pop:A)(2,push(C))(2,push:)...  
...(2,push(D))(2,push:)(1,pop:D)(1,pop())(1,pop:B)



## Back to ABA problem for the bugged stack

(1,pop())(2,pop())(2,pop:A)(2,push(C))(2,push:)...  
...(2,push(D))(2,push:)(1,pop:D)(1,pop())(1,pop:B)



Is SC, but not QC nor linearizable

## The choice of linearizability

Linearizability is the standard requirement for concurrent non-blocking data structures, because

- ▶ it is compositional
- ▶ it implies all other desired correctness properties
- ▶ it is (more or less...) easy to establish exhibiting *linearization points* in the code.

## Linearization points

A linearization point is an instruction that:

- ▶ occurs in the code, between call  $op(dots)$  and return  $op : \dots$

## Linearization points

A linearization point is an instruction that:

- ▶ occurs in the code, between call  $op(dots)$  and return  $op : \dots$
- ▶ can be seen as atomic (if not, it is a linearizable operation)

## Linearization points

A linearization point is an instruction that:

- ▶ occurs in the code, between call  $op(dots)$  and return  $op : \dots$
- ▶ can be seen as atomic (if not, it is a linearizable operation)
- ▶ commits the change of  $op$  on the shared data structure:

## Linearization points

A linearization point is an instruction that:

- ▶ occurs in the code, between call  $op(dots)$  and return  $op : \dots$
- ▶ can be seen as atomic (if not, it is a linearizable operation)
- ▶ commits the change of  $op$  on the shared data structure:
  - all concurrent calls that end before the beginning of this instruction understand  $op$  as not applied

## Linearization points

A linearization point is an instruction that:

- ▶ occurs in the code, between call  $op(dots)$  and return  $op : \dots$
- ▶ can be seen as atomic (if not, it is a linearizable operation)
- ▶ commits the change of  $op$  on the shared data structure:
  - all concurrent calls that end before the beginning of this instruction understand  $op$  as not applied
  - all concurrent calls that start after the end of this instruction understand  $op$  as applied

## The linearization points in Treiber's concurrent stack

---

```
void push(value v) {
    cell *t, *x;
    x = alloc();
    x->data = v;
    do {
        t = top_ptr;
        x->next = t;
    } while (!CAS(&top_ptr,t,x));
}

value pop() {
    cell *t, *x;
    do {
        t = top_ptr;
        if (t == NULL) return EMPTY;
        x = t->next;
    }
    while (!CAS(&top_ptr,t,x));
    return t->data;
}
```

## The linearization points in Treiber's concurrent stack

---

```
void push(value v) {
    cell *t, *x;
    x = alloc();
    x->data = v;
    do {
        t = top_ptr;
        x->next = t;
    } while (!CAS(&top_ptr,t,x));
}

value pop() {
    cell *t, *x;
    do {
        t = top_ptr;
        if (t == NULL) return EMPTY;
        x = t->next;
    }
    while (!CAS(&top_ptr,t,x));
    return t->data;
}
```

## The linearization points in the lazy lists

---

```
bool is_in(int i)
    cell *p=search(i);
    cell *n=p->next;
return n->val==v & !
n->marked;
```

```
void remove(int i){
    while true{    cell
*p=search(i);
    cell *n=p->next;
    lock(p->lock)
    lock(n->lock)
    if validate(p,n){
n->marked=true;
p->next=n->next;    break;}
    unlock(p->lock);
    unlock(n->lock);
}
```

## The linearization points in the lazy lists

---

```
bool is_in(int i)
    cell *p=search(i);
    cell *n=p->next;
return n->val==v & !
n->marked;
```

```
void remove(int i){
    while true{    cell
*p=search(i);
    cell *n=p->next;
    lock(p->lock)
    lock(n->lock)
    if validate(p,n){
n->marked=true;
p->next=n->next;    break;}
    unlock(p->lock);
    unlock(n->lock);
}
```

## Linearization points may be tricky!

- ▶ there might be several linearization points candidate even if only one will act on, depending on the context/the issue of the function call (e.g. M&S queue).
  
- ▶ external linearization  
some linearization points can lie out of the code of the function itself, and are then executed by a concurrent thread.

. Some words on progress properties .

## Progress properties for blocking concurrency

**dead-lock freedom** : for all reachable configuration  $\gamma$  such that  $t$  asks for entering a critical section  $s$ , there is a configuration  $\gamma'$  reachable from  $\gamma$  such that  $t$  enters  $s$ .

**non-starvation**: if each thread, when run in isolation, terminates and asks only finitely many times for entering a critical section, then the program terminates whatever the scheduling.

## Progress properties for blocking concurrency

**dead-lock freedom** : for all reachable configuration  $\gamma$  such that  $t$  asks for entering a critical section  $s$ , there is a configuration  $\gamma'$  reachable from  $\gamma$  such that  $t$  enters  $s$ .

**non-starvation**: if each thread, when run in isolation, terminates and asks only finitely many times for entering a critical section, then the program terminates whatever the scheduling.

These progress notions make thread depend upon each other

- ▶ not fault tolerant
- ▶ nor tolerates long waiting (page fault, processor overload,...)

Independent progress is more efficient...

... if active waiting is considered not *that* bad.

## Independent progress

**Wait-free:** *for all fair scheduling, any method call terminates*

## Independent progress

**Wait-free:** *for all fair scheduling, any method call terminates*

**Lock-free:** *for all fair scheduling, infinitely often, one call terminates.*

## Independent progress

**Wait-free:** *for all fair scheduling, any method call terminates*

**Lock-free:** *for all fair scheduling, infinitely often, one call terminates.*

**Obstruction-free:** *from any configuration, for all thread  $t$ , if  $t$  is scheduled and never preempted before it terminates, then it terminates.*

## Exercise

**How do these progress properties relate to each other?**

**How do these progress properties relate to each other?**

wait-free  $\Rightarrow$  lock-free  $\Rightarrow$  obstruction-free

**How do these progress properties relate to each other?**

wait-free  $\Rightarrow$  lock-free  $\Rightarrow$  obstruction-free

**What kind of progress is guaranteed for**

- ▶ Lazy lock-coupling lists

**How do these progress properties relate to each other?**

wait-free  $\Rightarrow$  lock-free  $\Rightarrow$  obstruction-free

**What kind of progress is guaranteed for**

- ▶ Lazy lock-coupling lists
  - none: it is not obstruction free, since a thread can be blocked by a lock owned by another thread;

**How do these progress properties relate to each other?**

wait-free  $\Rightarrow$  lock-free  $\Rightarrow$  obstruction-free

**What kind of progress is guaranteed for**

- ▶ Lazy lock-coupling lists
  - none: it is not obstruction free, since a thread can be blocked by a lock owned by another thread;
  - some CAS-based versions of lazy lock-coupling lists are lock-free.

**How do these progress properties relate to each other?**

wait-free  $\Rightarrow$  lock-free  $\Rightarrow$  obstruction-free

**What kind of progress is guaranteed for**

- ▶ Lazy lock-coupling lists
  - none: it is not obstruction free, since a thread can be blocked by a lock owned by another thread;
  - some CAS-based versions of lazy lock-coupling lists are lock-free.
- ▶ Treiber's stack:

### How do these progress properties relate to each other?

wait-free  $\Rightarrow$  lock-free  $\Rightarrow$  obstruction-free

### What kind of progress is guaranteed for

- ▶ Lazy lock-coupling lists
  - none: it is not obstruction free, since a thread can be blocked by a lock owned by another thread;
  - some CAS-based versions of lazy lock-coupling lists are lock-free.
- ▶ Treiber's stack:
  - it is not wait-free: a thread can always fail to commit, because its copy of `top_ptr` is repeatedly outdated;

### How do these progress properties relate to each other?

wait-free  $\Rightarrow$  lock-free  $\Rightarrow$  obstruction-free

### What kind of progress is guaranteed for

- ▶ Lazy lock-coupling lists
  - none: it is not obstruction free, since a thread can be blocked by a lock owned by another thread;
  - some CAS-based versions of lazy lock-coupling lists are lock-free.
- ▶ Treiber's stack:
  - it is not wait-free: a thread can always fail to commit, because its copy of `top_ptr` is repeatedly outdated;
  - it is lock-free: everytime a thread fails, at least one thread succeeds.

# Rely-Guarantee and Separation Logic

## Reasoning about races

$y := x$	$\{x > 0\}$	$y' := x;$
$x := y + 1$	$\parallel$	$x := y' + 2$
	$\{x > 0\}$	

## Reasoning about races

$$\begin{array}{ccc} & \{x > 0\} & \\ y:=x & \parallel & y':=x; \\ x:=y+1 & & x:=y'+2 \\ & \{x > 0\} & \end{array}$$

Can be proved modularly:

- ▶ assuming that the environment ( $T - \{t\}$ ) only increments  $x$
- ▶ thread  $t$  can guarantee to only increment  $x$

## Reasoning about races

$$\begin{array}{ccc} & \{x > 0\} & \\ y:=x & \parallel & y':=x; \\ x:=y+1 & & x:=y'+2 \\ & \{x > 0\} & \end{array}$$

Can be proved modularly:

- ▶ assuming that the environment ( $T - \{t\}$ ) only increments  $x$  (assuming  $t$  only increments  $x$ )
- ▶ thread  $t$  can guarantee to only increment  $x$

## Reasoning about races

$$\begin{array}{ccc} & \{x > 0\} & \\ y:=x & \parallel & y':=x; \\ x:=y+1 & & x:=y'+2 \\ & \{x > 0\} & \end{array}$$

Can be proved modularly:

- ▶ assuming that the environment ( $T - \{t\}$ ) only increments  $x$  (assuming  $t$  only increments  $x$ )
- ▶ thread  $t$  can guarantee to only increment  $x$
- ▶ a circularity: unsound for termination, but sound for safety.

## Reasoning about races

$$\begin{array}{ccc} & \{x > 0\} & \\ y:=x & \parallel & y':=x; \\ x:=y+1 & & x:=y'+2 \\ & \{x > 0\} & \end{array}$$

Can be proved modularly:

- ▶ assuming that the environment ( $T - \{t\}$ ) only **increments**  $x$  (assuming  $t$  only increments  $x$ )
- ▶ thread  $t$  can guarantee to only **increment**  $x$
- ▶ a circularity: unsound for termination, but sound for safety.
- ▶ not only state assertions: we need to talk about actions = binary relations among states

## Actions, Stability

**Actions:**  $R, G \subseteq \Sigma \times \Sigma$

Example:  $\&x \mapsto X \rightsquigarrow \&x \mapsto X + 1$

## Actions, Stability

**Actions:**  $R, G \subseteq \Sigma \times \Sigma$

Example:  $\&x \mapsto X \rightsquigarrow \&x \mapsto X + 1$

**Stability:**  $\phi \subseteq \Sigma$  is stable by  $R$  if  $\phi; R \subseteq \phi$ .

$R, G \vdash \{\phi\} p \{\psi\}$  is said valid if

- ▶  $p$  transforms (safely) any state of  $\phi$  into a state of  $\psi$  *when run in a context interfering only through  $R$* :  $\llbracket p \rrbracket_R(\llbracket \phi \rrbracket) \sqsubseteq \llbracket \psi \rrbracket$
- ▶ along any trace of  $p$  in such a context, all actions of  $p$  are guaranteed actions:  $actions(p, R) \subseteq G$ .

Remarks: if  $R, G \vdash \{\phi\} p \{\psi\}$ , then:

- ▶  $\psi$  is stable by  $R$
- ▶  $\{\phi; R^*\} p \{\psi\}$

## The role of atomicity

whether an instruction is atomic or not matters:

example:  $\{x > 0\} x := x + 2 \parallel x := x - 1 \{x > 0\}$

## The role of atomicity

whether an instruction is atomic or not matters:

example:  $\{x > 0\} x := x + 2 || x := x - 1 \{x > 0\}$

Programming language:

$$p ::= p; p, p || p, x := E, \langle p \rangle, \dots$$

Example:

$$\{x > 0\} \langle x := x + 2 \rangle || \langle x := x - 1 \rangle \{x > 0\}$$

becomes sound.

## Some rules for RG

$$\frac{R, G \vdash \{\phi\} p_1 \{\kappa\} \quad \{\kappa\} p_2 \{\psi\}}{\{\phi\} p_1; p_2 \{\psi\}}$$

## Some rules for RG

$$\frac{R, G \vdash \{\phi\} p_1 \{\kappa\} \quad \{\kappa\} p_2 \{\psi\}}{\{\phi\} p_1; p_2 \{\psi\}}$$

$$\frac{R \cup G_2, G_1 \vdash \{\phi\} p_1 \{\psi\} \quad R \cup G_1, G_2 \vdash \{\phi\} p_2 \{\psi\}}{R, G_1 \cup G_2 \vdash \{\phi\} p_1 || p_2 \{\psi\}}$$

## Some rules for RG

$$\frac{R, G \vdash \{\phi\} p_1 \{\kappa\} \quad \{\kappa\} p_2 \{\psi\}}{\{\phi\} p_1; p_2 \{\psi\}}$$

$$\frac{R \cup G_2, G_1 \vdash \{\phi\} p_1 \{\psi\} \quad R \cup G_1, G_2 \vdash \{\phi\} p_2 \{\psi\}}{R, G_1 \cup G_2 \vdash \{\phi\} p_1 || p_2 \{\psi\}}$$

$$\frac{\emptyset, G \vdash \{\phi\} \langle p \rangle \{\psi\} \quad \phi, \psi \in \text{stable}(R)}{R, G \vdash \{\phi\} \langle p \rangle \{\psi\}}$$

## Some rules for RG

$$\frac{R, G \vdash \{\phi\} p_1 \{\kappa\} \quad \{\kappa\} p_2 \{\psi\}}{\{\phi\} p_1; p_2 \{\psi\}}$$

$$\frac{R \cup G_2, G_1 \vdash \{\phi\} p_1 \{\psi\} \quad R \cup G_1, G_2 \vdash \{\phi\} p_2 \{\psi\}}{R, G_1 \cup G_2 \vdash \{\phi\} p_1 || p_2 \{\psi\}}$$

$$\frac{\emptyset, G \vdash \{\phi\} \langle p \rangle \{\psi\} \quad \phi, \psi \in \text{stable}(R)}{R, G \vdash \{\phi\} \langle p \rangle \{\psi\}}$$

$$\frac{\emptyset, \emptyset \vdash \{\phi\} p \{\psi\} \quad \llbracket p \rrbracket \subseteq G}{\emptyset, G \vdash \{\phi\} \langle p \rangle \{\psi\}}$$

# Ownership and interference

## Owned state and shared state

- ▶ the state  $\sigma$  is composed of a shared state  $s$  and a local state  $l$
- ▶ the shared state is submitted to interference, the local state is not

# Ownership and interference

## Owned state and shared state

- ▶ the state  $\sigma$  is composed of a shared state  $s$  and a local state  $l$
- ▶ the shared state is submitted to interference, the local state is not

**Boxed formulas** (capital  $\Phi$  denotes a standard SL formula)

$$\phi ::= \Phi, \boxed{\Phi}, \phi * \phi, \dots$$

# Ownership and interference

## Owned state and shared state

- ▶ the state  $\sigma$  is composed of a shared state  $s$  and a local state  $l$
- ▶ the shared state is submitted to interference, the local state is not

**Boxed formulas** (capital  $\Phi$  denotes a standard SL formula)

$$\phi ::= \Phi, \boxed{\Phi}, \phi * \phi, \dots$$

## Semantics

- ▶  $s, l \models \Phi$  if  $l \models \Phi$

# Ownership and interference

## Owned state and shared state

- ▶ the state  $\sigma$  is composed of a shared state  $s$  and a local state  $l$
- ▶ the shared state is submitted to interference, the local state is not

**Boxed formulas** (capital  $\Phi$  denotes a standard SL formula)

$$\phi ::= \Phi, \boxed{\Phi}, \phi * \phi, \dots$$

## Semantics

- ▶  $s, l \models \Phi$  if  $l \models \Phi$
- ▶  $s, l \models \boxed{\Phi}$  if  $s \models \Phi$  and  $l = \emptyset$

# Ownership and interference

## Owned state and shared state

- ▶ the state  $\sigma$  is composed of a shared state  $s$  and a local state  $l$
- ▶ the shared state is submitted to interference, the local state is not

**Boxed formulas** (capital  $\Phi$  denotes a standard SL formula)

$$\phi ::= \Phi, \boxed{\Phi}, \phi * \phi, \dots$$

## Semantics

- ▶  $s, l \models \Phi$  if  $l \models \Phi$
- ▶  $s, l \models \boxed{\Phi}$  if  $s \models \Phi$  and  $l = \emptyset$
- ▶  $s, l \models \phi * \psi$  if  $s, l_1 \models \phi$  and  $s, l_2 \models \psi$  for some splitting  $l = l_1 \bullet l_2$ .

# Ownership and interference

## Owned state and shared state

- ▶ the state  $\sigma$  is composed of a shared state  $s$  and a local state  $l$
- ▶ the shared state is submitted to interference, the local state is not

**Boxed formulas** (capital  $\Phi$  denotes a standard SL formula)

$$\phi ::= \Phi, \boxed{\Phi}, \phi * \phi, \dots$$

## Semantics

- ▶  $s, l \models \Phi$  if  $l \models \Phi$
- ▶  $s, l \models \boxed{\Phi}$  if  $s \models \Phi$  and  $l = \emptyset$
- ▶  $s, l \models \phi * \psi$  if  $s, l_1 \models \phi$  and  $s, l_2 \models \psi$  for some splitting  $l = l_1 \bullet l_2$ .

Remark:  $\boxed{\Phi \wedge \Psi} = \boxed{\Phi} * \boxed{\Psi}$ .

## Atomicity and shared state

Atomicity makes shared state local:

$$\frac{\emptyset, \emptyset \vdash \{\Phi * \Phi'\} \quad p \quad \{\Psi * \Psi'\} \quad \llbracket p \rrbracket \subseteq G}{\emptyset, G \vdash \{\Phi * \boxed{\Phi'}\} \quad \langle p \rangle \quad \{\Psi * \boxed{\Psi'}\}}$$

## Atomicity and shared state

Atomicity makes shared state local:

$$\frac{\emptyset, \emptyset \vdash \{\Phi * \Phi'\} \quad p \{\Psi * \Psi'\} \quad \llbracket p \rrbracket \subseteq G}{\emptyset, G \vdash \{\Phi * \boxed{\Phi'}\} \quad \langle p \rangle \{\Psi * \boxed{\Psi'}\}}$$

Local state can be modified as usual

$$\frac{\{\Phi\} \quad p \{\Psi\} \text{ in standard SL}}{\emptyset, \emptyset \vdash \{\Phi\} \quad p \{\Psi\}}$$

## Example: Treiber's stack

```
void push(value v) {
    cell *t, *x;
    x = alloc();
    x->data = v;
    do {
        t = top_ptr;
        x->next = t;
    } while (!CAS(&top_ptr,t,x));
}
```

## Example: Treiber's stack

---

push(v) = local t,x,b in

```
x:=new(); x→data=v; failed:=true;  
while failed do
```

```
  ⟨t:=top⟩;
```

```
  x→next := t;
```

```
  ⟨if top=t then top:=x;failed:=false⟩
```

```
endwhile
```

## Example: Treiber's stack

---

action PUSH [ $!s(\text{top}) * \text{top} = X$ ] [ $\text{top} \rightarrow X * !s(X)$ ]  
action POP [ $\text{top} \rightarrow X * !s(X)$ ] [ $!s(\text{top}) * \text{top} = X$ ]

push( $v$ ) = local  $t, x, b$  in

```
 $x := \text{new}()$ ;  $x \rightarrow \text{data} = v$ ; failed := true;  
while failed do
```

```
   $\langle t := \text{top} \rangle$ ;
```

```
   $x \rightarrow \text{next} := t$ ;
```

```
   $\langle \text{if } \text{top} = t \text{ then } \text{top} := x; \text{failed} := \text{false} \rangle$ 
```

```
endwhile
```

## Example: Treiber's stack

---

action PUSH [ $!s(\text{top}) * \text{top} = X$ ] [ $\text{top} \rightarrow X * !s(X)$ ]  
action POP [ $\text{top} \rightarrow X * !s(X)$ ] [ $!s(\text{top}) * \text{top} = X$ ]

$\{ !s(\text{top}) \}$

push( $v$ ) = local  $t, x, b$  in

$x := \text{new}()$ ;  $x \rightarrow \text{data} = v$ ;  $\text{failed} := \text{true}$ ;  
while  $\text{failed}$  do

$\langle t := \text{top} \rangle$ ;

$x \rightarrow \text{next} := t$ ;

$\langle \text{if } \text{top} = t \text{ then } \text{top} := x; \text{failed} := \text{false} \rangle$

endwhile

## Example: Treiber's stack

action PUSH [ $ls(top)*top=X$ ] [ $top \mapsto X*ls(X)$ ]  
action POP [ $top \mapsto X*ls(X)$ ] [ $ls(top)*top=X$ ]

$\{ ls(top) \}$

push(v) = local t,x,b in

$\{ ls(top) * \&x \mapsto * \&t \mapsto * \&b \mapsto \}$

$x := \text{new}(); x \mapsto \text{data} = v; \text{failed} := \text{true};$

**while** failed **do**

$\{ ls(top) * x \mapsto * \&t \mapsto * \text{failed} = \text{true} \}$

$\langle t := \text{top} \rangle;$

$x \mapsto \text{next} := t;$

$\langle \text{if } \text{top} = t \text{ then } \text{top} := x; \text{failed} := \text{false} \rangle$

**endwhile**

## Example: Treiber's stack

action PUSH [ $ls(top) * top = X$ ] [ $top \mapsto X * ls(X)$ ]  
action POP [ $top \mapsto X * ls(X)$ ] [ $ls(top) * top = X$ ]

```
{  $ls(top)$  }  
push(v) = local t,x,b in  
{  $ls(top) * \&x \mapsto * \&t \mapsto * \&b \mapsto$  }  
  x := new(); x  $\rightarrow$  data = v; failed := true;  
  while failed do  
  {  $ls(top) * x \mapsto * \&t \mapsto * failed = true$  }  
     $\langle t := top \rangle$ ;  
  {  $ls(top) * x \mapsto * \&t \mapsto * failed = true$  }  
    x  $\rightarrow$  next := t;  
  
   $\langle$  if top = t then top := x; failed := false  $\rangle$   
  
endwhile
```

## Example: Treiber's stack

action PUSH [ $ls(top) * top = X$ ] [ $top \mapsto X * ls(X)$ ]  
action POP [ $top \mapsto X * ls(X)$ ] [ $ls(top) * top = X$ ]

```
{  $ls(top)$  }  
push(v) = local t,x,b in  
{  $ls(top) * \&x \mapsto * \&t \mapsto * \&b \mapsto$  }  
  x := new(); x → data = v; failed := true;  
  while failed do  
  {  $ls(top) * x \mapsto * \&t \mapsto * failed = true$  }  
    ⟨t := top⟩;  
  {  $ls(top) * x \mapsto * \&t \mapsto * failed = true$  }  
    x → next := t;  
  {  $ls(top) * x \mapsto t * failed = true$  }  
    ⟨if top = t then top := x; failed := false⟩  
  
endwhile
```

## Example: Treiber's stack

action PUSH [ $ls(top) * top = X$ ] [ $top \mapsto X * ls(X)$ ]  
action POP [ $top \mapsto X * ls(X)$ ] [ $ls(top) * top = X$ ]

```
{  $ls(top)$  }  
push(v) = local t,x,b in  
{  $ls(top) * \&x \mapsto * \&t \mapsto * \&b \mapsto$  }  
  x := new(); x → data = v; failed := true;  
  while failed do  
  {  $ls(top) * x \mapsto * \&t \mapsto * failed = true$  }  
    ⟨t := top⟩;  
  {  $ls(top) * x \mapsto * \&t \mapsto * failed = true$  }  
    x → next := t;  
  {  $ls(top) * x \mapsto t * failed = true$  }  
    ⟨if top = t then top := x; failed := false⟩ as PUSH  
  
endwhile
```

## Example: Treiber's stack

action PUSH [ $ls(top) * top = X$ ] [ $top \mapsto X * ls(X)$ ]  
action POP [ $top \mapsto X * ls(X)$ ] [ $ls(top) * top = X$ ]

```
{  $ls(top)$  }  
push(v) = local t,x,b in  
{  $ls(top) * \&x \mapsto * \&t \mapsto * \&b \mapsto$  }  
  x := new(); x  $\rightarrow$  data = v; failed := true;  
  while failed do  
  {  $ls(top) * x \mapsto * \&t \mapsto * failed = true$  }  
     $\langle t := top \rangle$ ;  
  {  $ls(top) * x \mapsto * \&t \mapsto * failed = true$  }  
    x  $\rightarrow next := t$ ;  
  {  $ls(top) * x \mapsto t * failed = true$  }  
     $\langle \text{if } top = t \text{ then } top := x; failed := false \rangle$  as PUSH  
  {  $ls(top) * \text{if } failed \text{ then } x \mapsto - \text{ else emp}$  }  
  endwhile
```

## Why RG Sep is not a breakthrough

### **RG Sep combines two distinct approaches**

- ▶ Concurrent SL requires a resource invariant  $I_r$
- ▶ RG requires actions  $a_1, \dots, a_n$  and assumptions  $\boxed{\Phi_1}, \dots, \boxed{\Phi_n}$  about the shared state.

## Why RG Sep is not a breakthrough

### **RG Sep combines two distinct approaches**

- ▶ Concurrent SL requires a resource invariant  $I_r$
- ▶ RG requires actions  $a_1, \dots, a_n$  and assumptions  $\boxed{\Phi_1}, \dots, \boxed{\Phi_n}$  about the shared state.

### **RG Sep is less expressive than each of these approaches**

- ▶ RG Sep can be encoded in RG (RG is indeed already complete).
- ▶ RG Sep can be encoded in Concurrent SL taking a unique resource, with  $I_r = \Phi_1 \wedge \dots \wedge \Phi_n$ .

## Why RG Sep is a breakthrough

- ▶ **RG Sep can be encoded in RG, but...**  
actions specification should precise that the local states are not changed  $\Rightarrow$  that's what SL is good for

## Why RG Sep is a breakthrough

- ▶ **RG Sep can be encoded in RG, but...**  
actions specification should precise that the local states are not changed  $\Rightarrow$  that's what SL is good for
- ▶ **RG Sep can be encoded in Concurrent SL, but...**  
there are some cases for which describing/checking all  $\Phi_i$  can be much more compact and efficient for automation than describing/checking  $I_r$  (factor 5 for the 4Slot algorithm).

## Why RG Sep is a breakthrough

- ▶ **RG Sep can be encoded in RG, but...**  
actions specification should precise that the local states are not changed  $\Rightarrow$  that's what SL is good for
- ▶ **RG Sep can be encoded in Concurrent SL, but...**  
there are some cases for which describing/checking all  $\Phi_i$  can be much more compact and efficient for automation than describing/checking  $I_r$  (factor 5 for the 4Slot algorithm).
- ▶ Turns to be a good basis for reasoning about non-blocking concurrency, and tackle linearizability and lock-freedom.

### On non-blocking concurrency

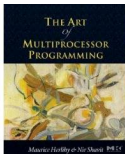
- ▶ non-blocking concurrency is fun and *very, very subtle*
- ▶ new algorithms and manual proofs worth still good publications
- ▶ automatic proofs of linearization are becoming available  
[Vafeiadis, CAV'10]

### On Rely Guarantee and SL

- ▶ extensions and variations have been published
- ▶ in particular, Deny Guarantee:  
putting separation and permissions on the specifications of the interferences.

Before getting separated...

Thanks a lot!



### On non-blocking concurrency

*The art of multiprocessor programming.*

M. Herlihy, N. Shavit.

### On Rely Guarantee and Separation Logic

- ▶ *Modular fine-grained concurrency verification.*

PhD. thesis of Viktor Vafeiadis. University of Cambridge, July 2007.

- ▶ *Deny-Guarantee Reasoning.*

Mike Dodds, Xinyu Feng, Matthew J. Parkinson, Viktor Vafeiadis.  
ESOP'09.

### Linearizability and progress with a touch of SL

- ▶ *Automatically proving linearizability.*

Viktor Vafeiadis. CAV'10.

- ▶ *Proving that nonblocking algorithms don't block.*

Alexey Gotsman, Byron Cook, Matthew J. Parkinson, Viktor Vafeiadis.