Proving Linearizability Using Partial Orders

Artem Khyzha Mike Dodds
Alexey Gotsman Matthew Parkinson

IMDEA Software Seminar Series Nov 3rd 2016

Concurrent libraries

- Encapsulate efficient concurrent data structures:
  - Java: java.util.concurrent
  - C++: Intel Threading Building Blocks
  - C#: System.Collections.Concurrent
- Implement stacks, queues, skip lists, hash tables etc
- Hard to prove correct

Concurrent data structure

- Proven to be simulated by their sequential specification

$$\boxed{\begin{array}{c}\text{data structure}\\\text{implementation}\end{array}} \sqsubseteq \boxed{\begin{array}{c}\text{data structure}\\\text{specification}\end{array}}$$

- History = a well-formed sequence of operation invokations and responses
- Formalised by "linearizability"

Dummy Queue

```
struct Node {
  Node *next; int val;
} *Tail;

void enqueue(Val v) {
  lock();
  try {
    Node e = new Node*;
    e.val = v;
    Tail.next = e;
    Tail = e;
  } finally {
    unlock();
  }
```

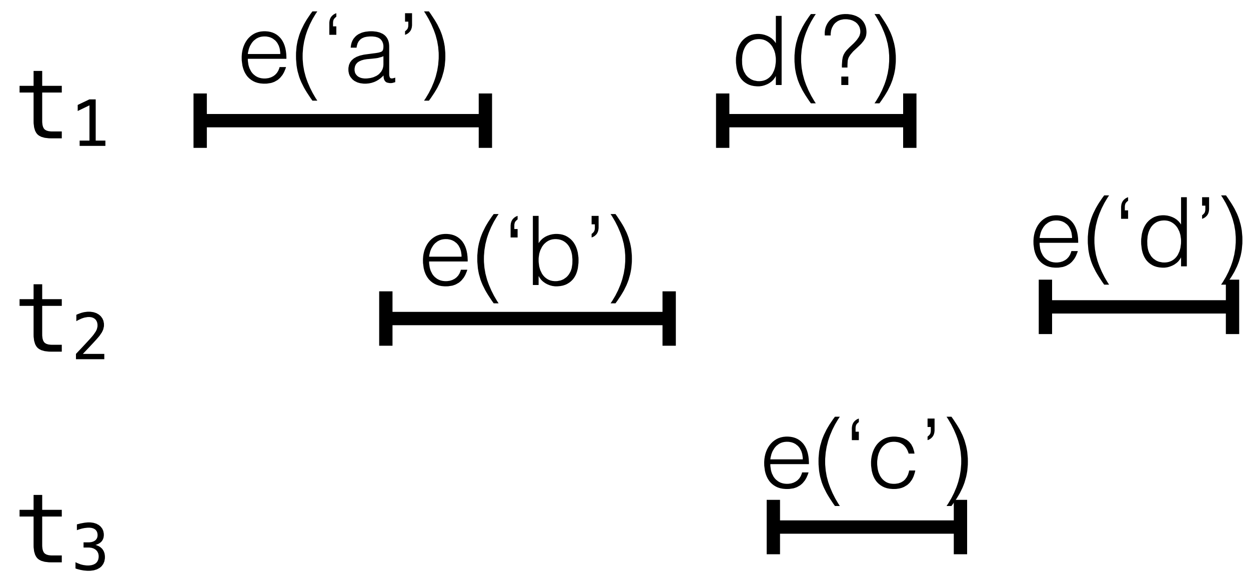
$\sqsubseteq$

Atomic queue ADT

```
Sequence S;

void enqueue(int v) {
  atomic { S = S :: v; }
}
```

Linearizability

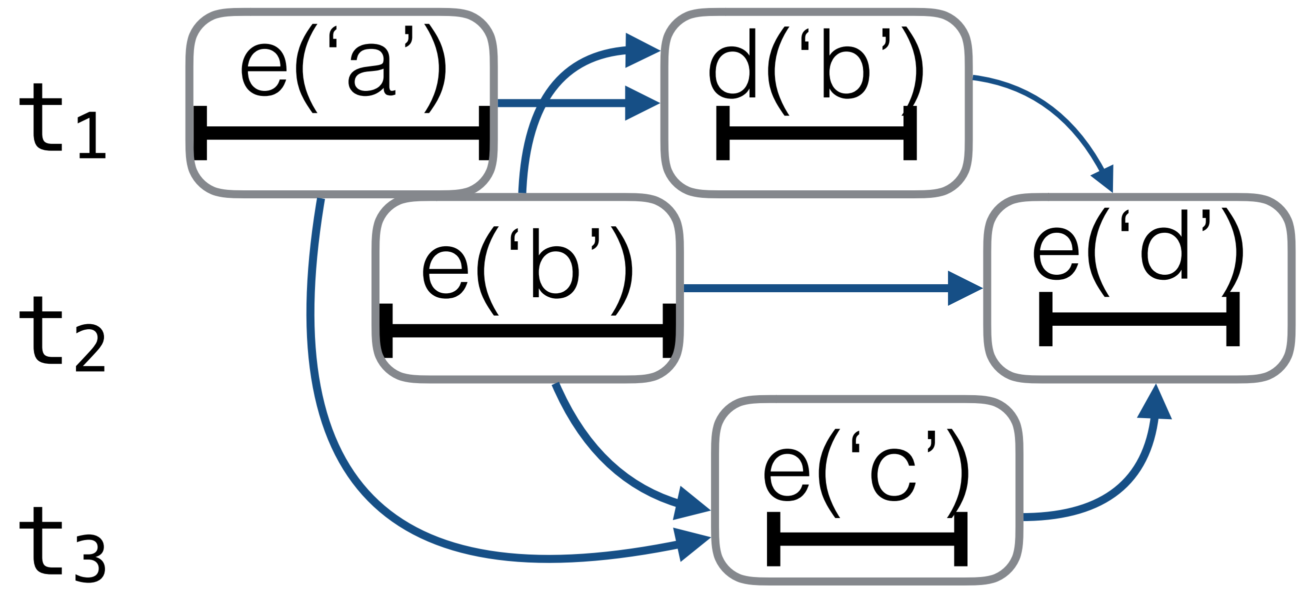
for every concrete history



find matching behaviour of the seq. spec.

Linearizability

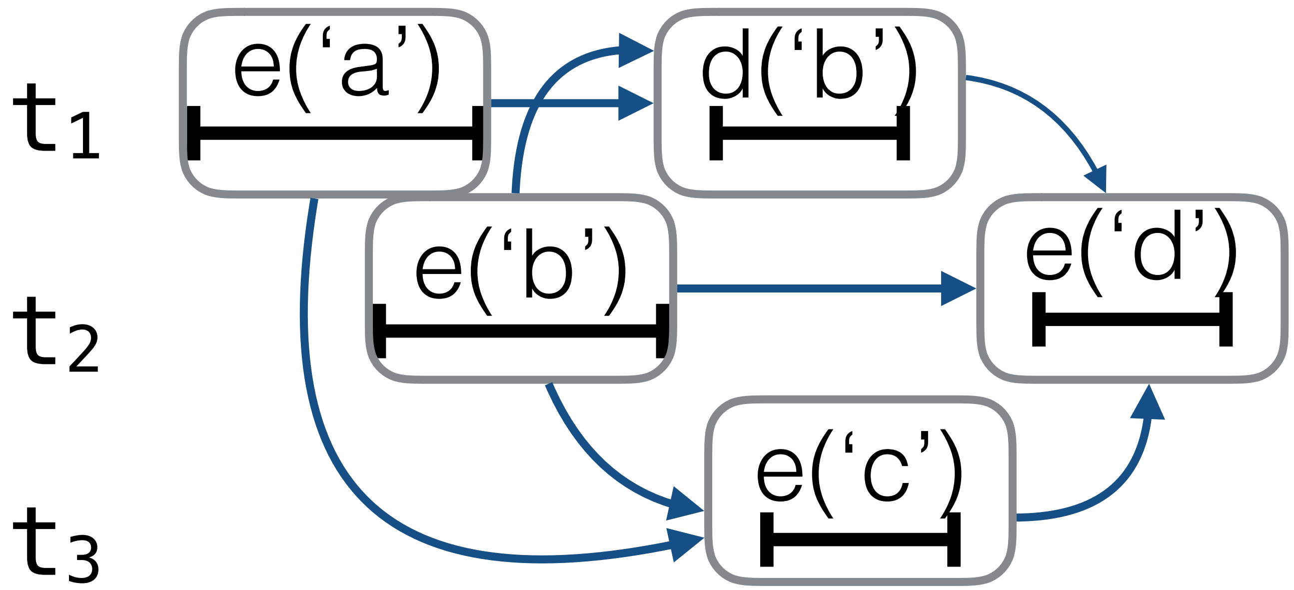
for every concrete history



find matching behaviour of the seq. spec.

Linearizability

for every concrete history

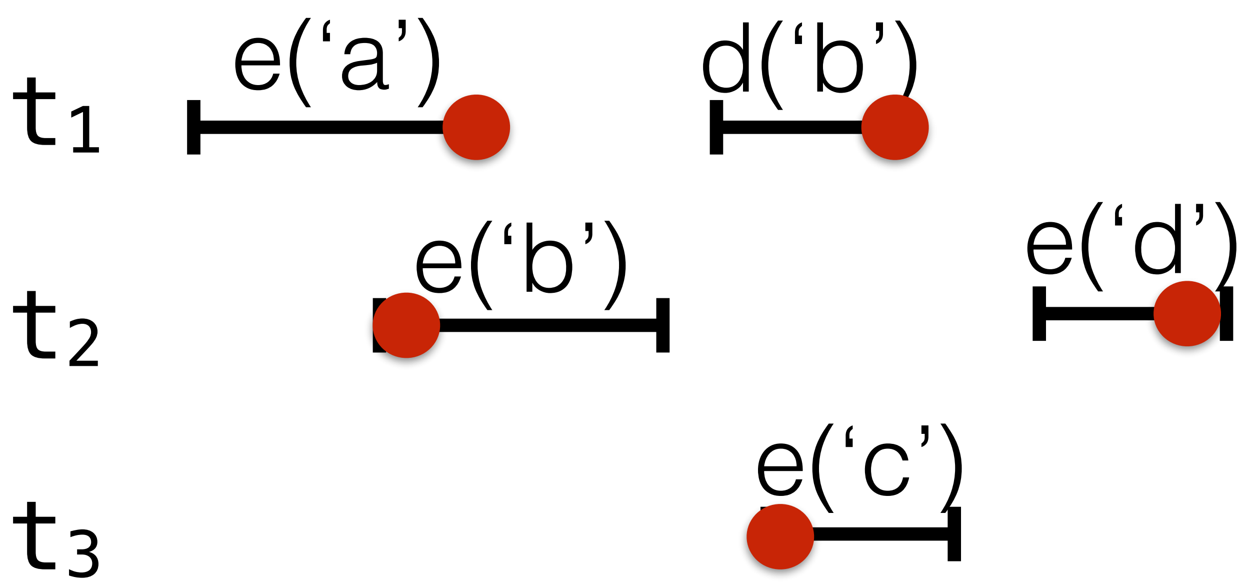


find a linearization

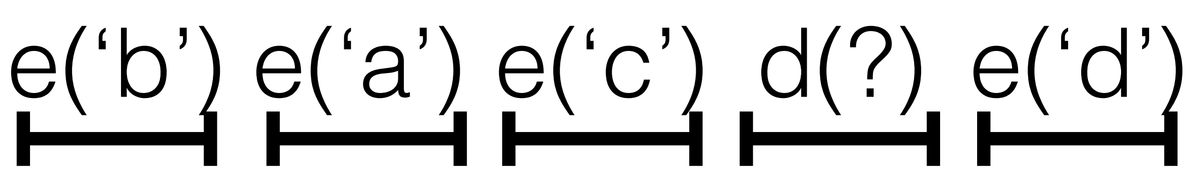
1. e('a') e('b') d('b') e('c') e('d')
2. e('a') e('b') e('c') d('b') e('d')
3. e('b') e('a') d('b') e('c') e('d')
4. e('b') e('a') e('c') d('b') e('d')

Linearization points

for every concrete history



find a linearization



Standard proof technique: linearization points

Dummy Queue

```
struct Node {
  Node *next; int val;
} *Top;

void enqueue(Val v) {
  lock();
  try {
    Node e = new Node*;
    e.val = v;
    tail.next = e;
    tail = e;
  } finally {
    unlock();
  }
```

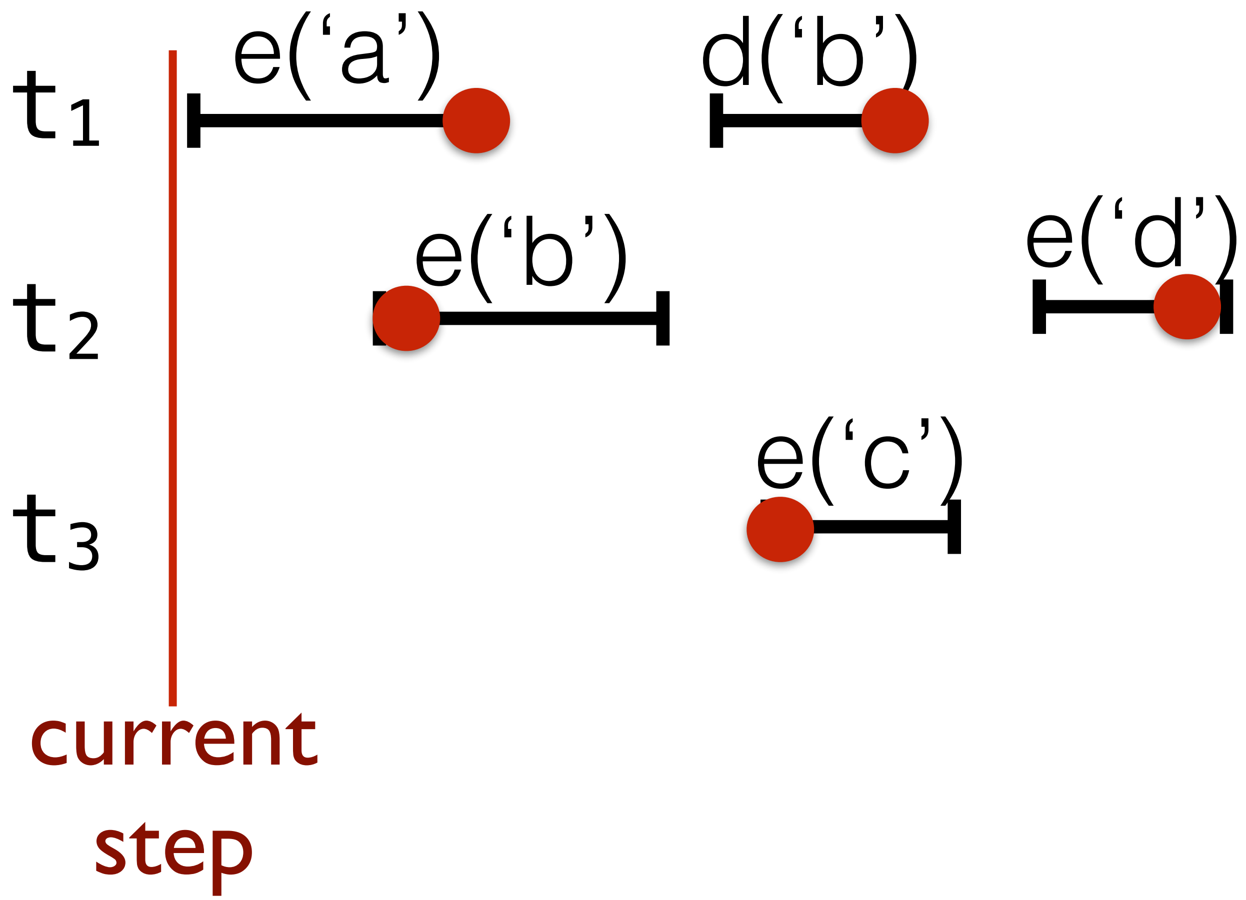
$\sqsubseteq$

Atomic queue ADT

```
Sequence S;

void enqueue(int v) {
  atomic { S = S :: v; }
}
```

Forward simulation

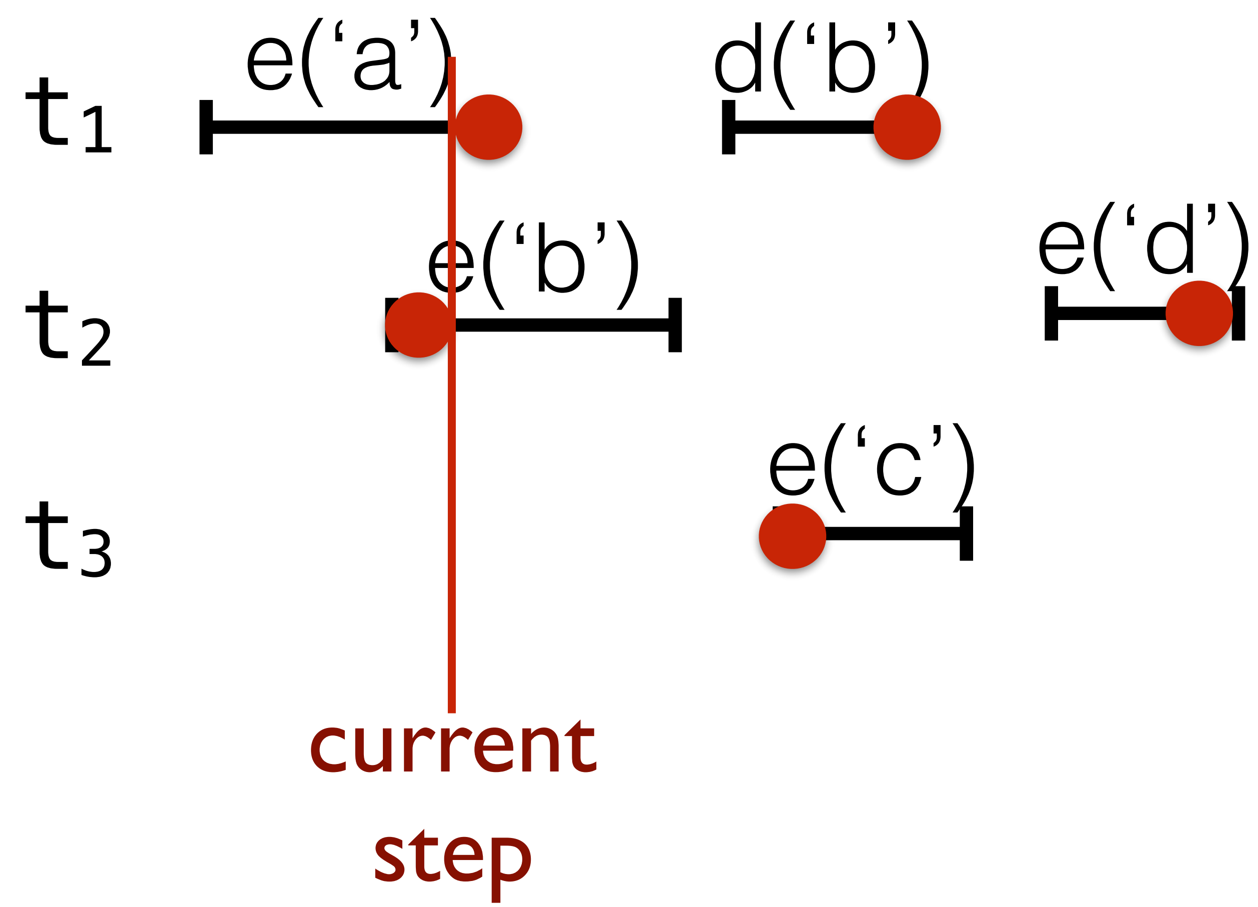
for every concrete history



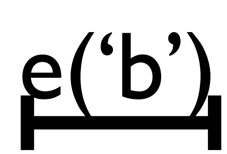
build a linearization: `<empty sequence>`

Forward simulation

for every concrete history

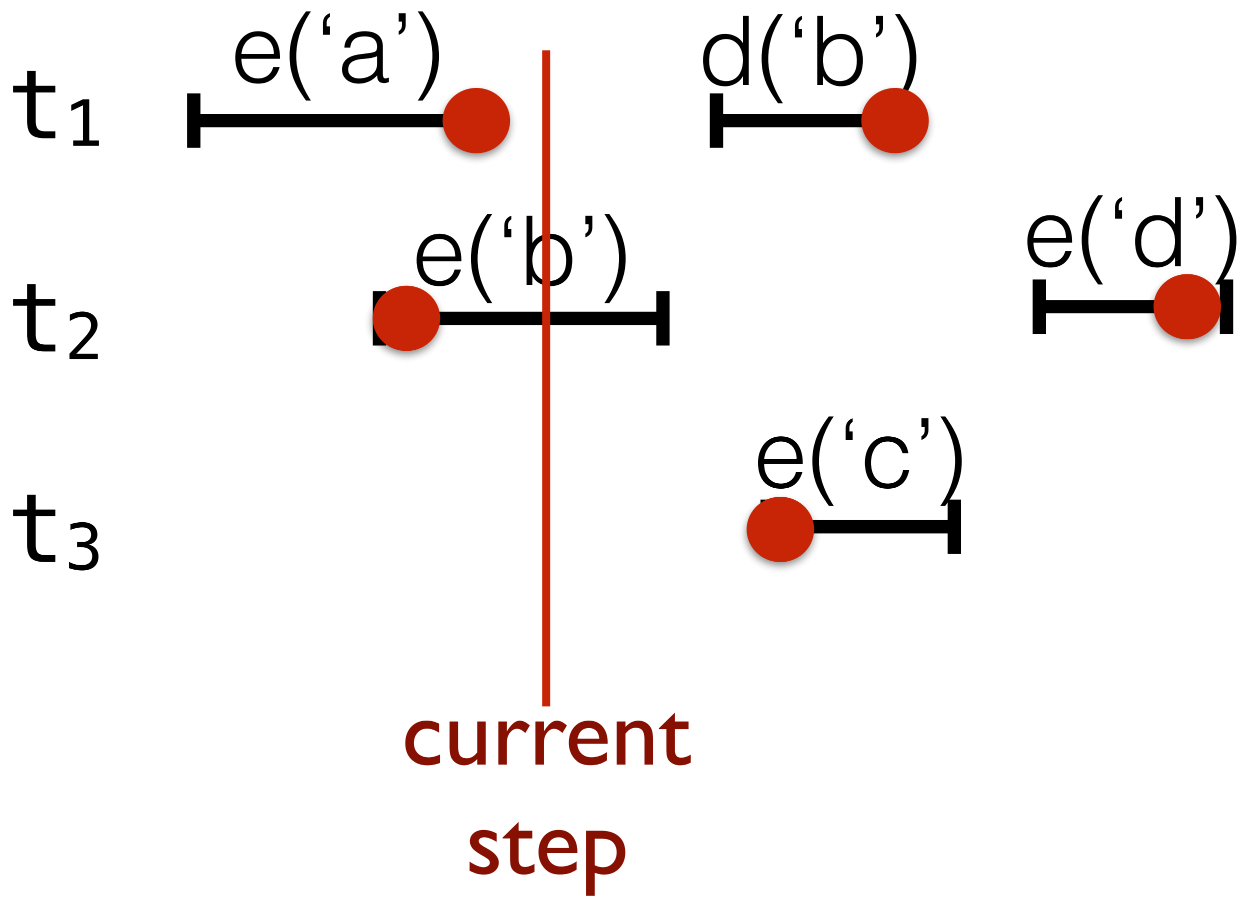


build a linearization:

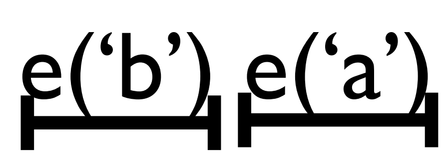


Forward simulation

for every concrete history

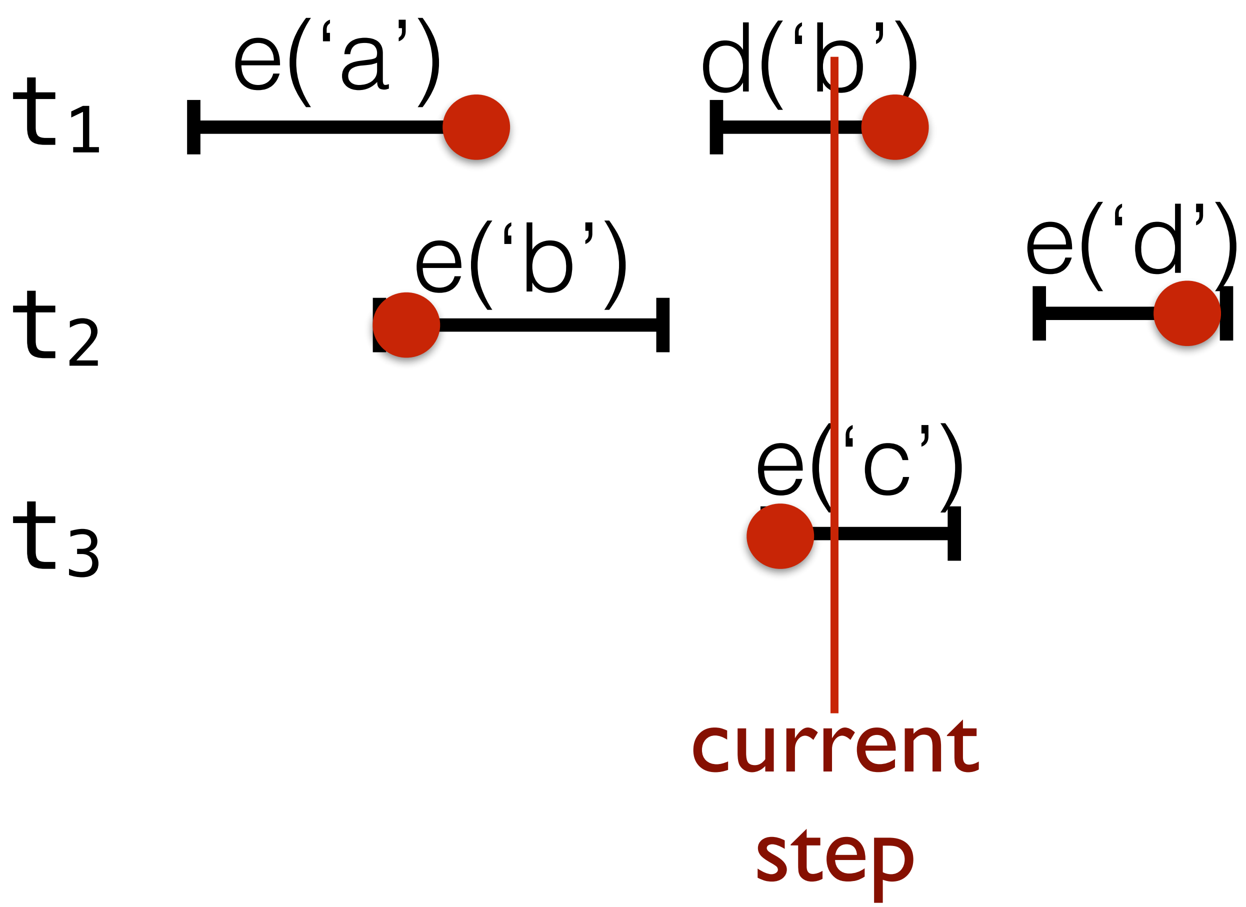


build a linearization:

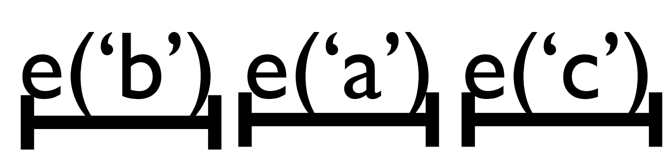


Forward simulation

for every concrete history

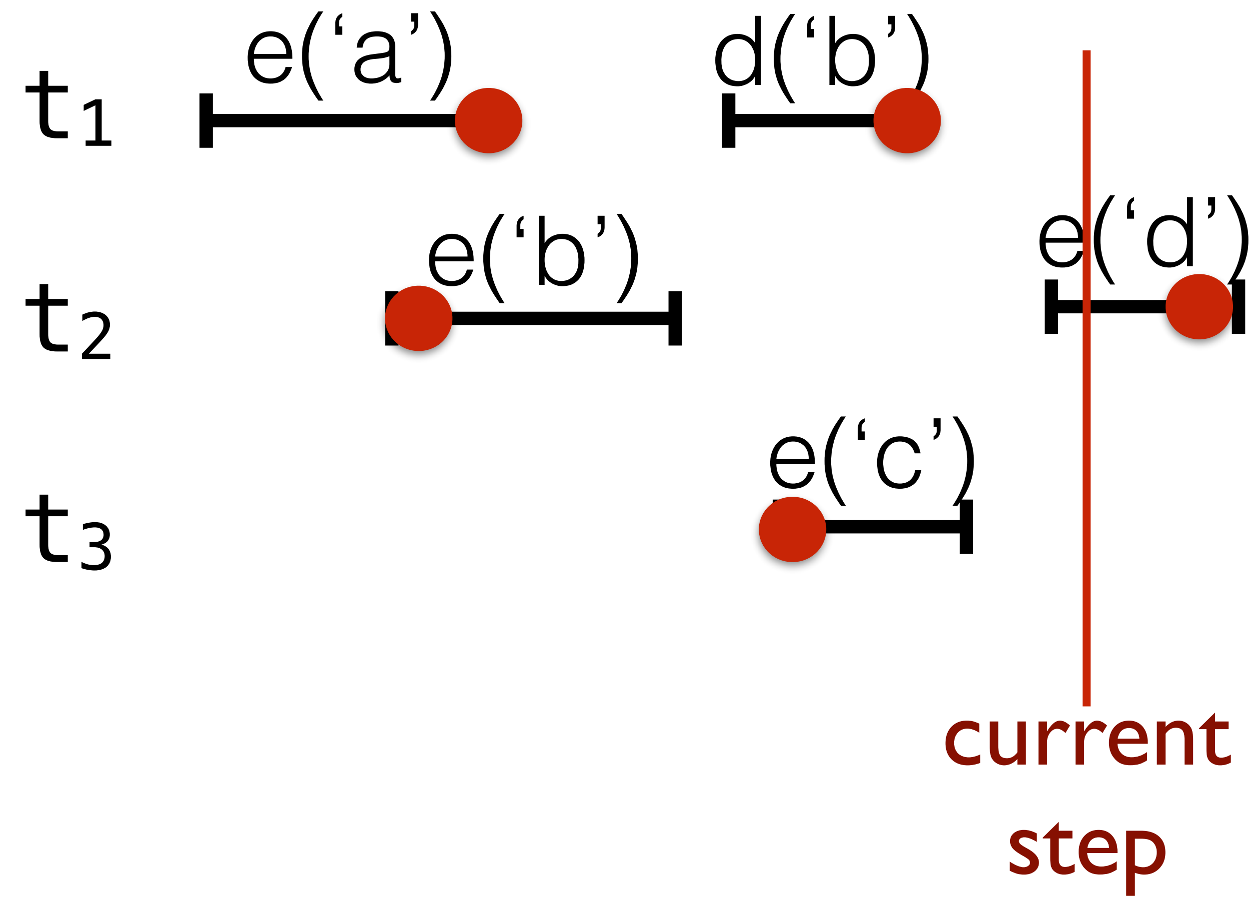


build a linearization:

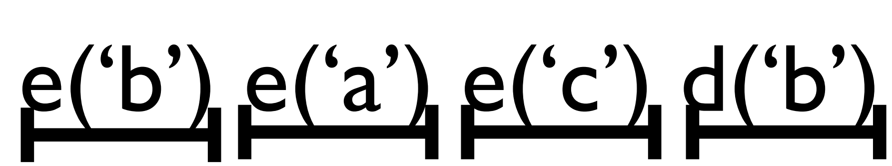


Forward simulation

for every concrete history

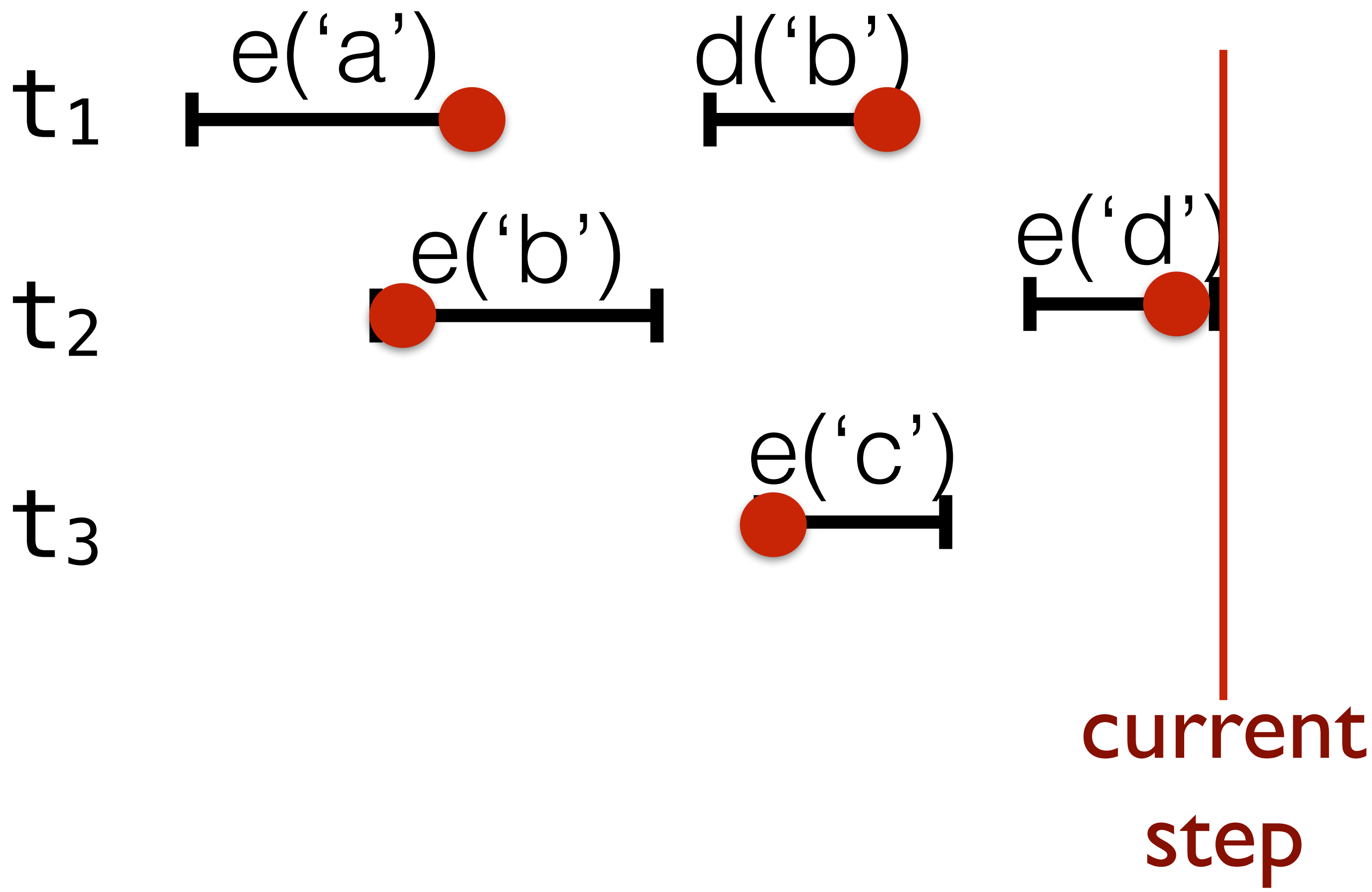


build a linearization:

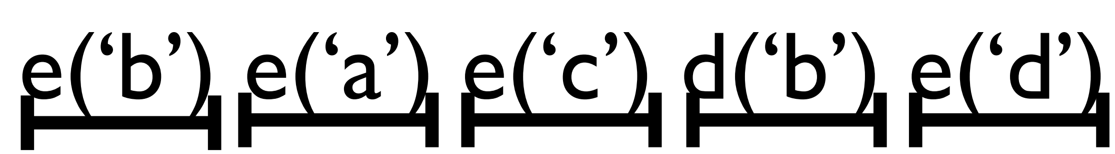


Forward simulation

for every concrete history

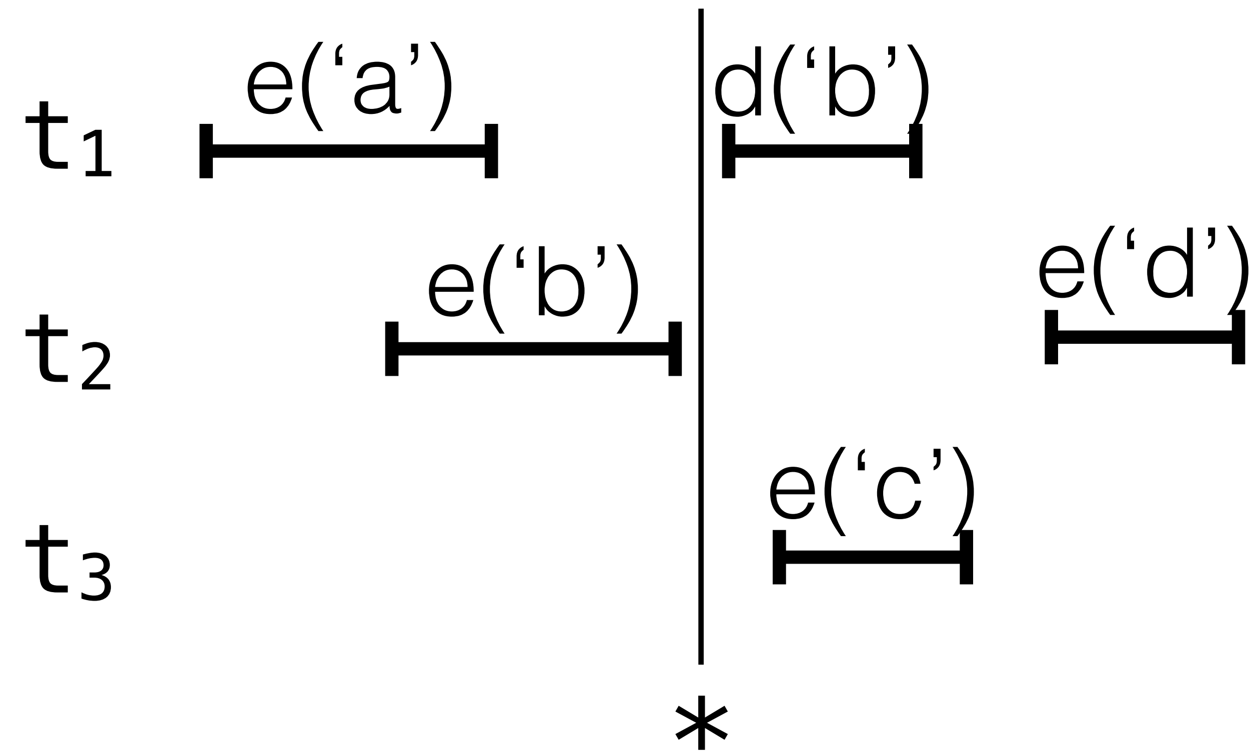


build a linearization:



Problem

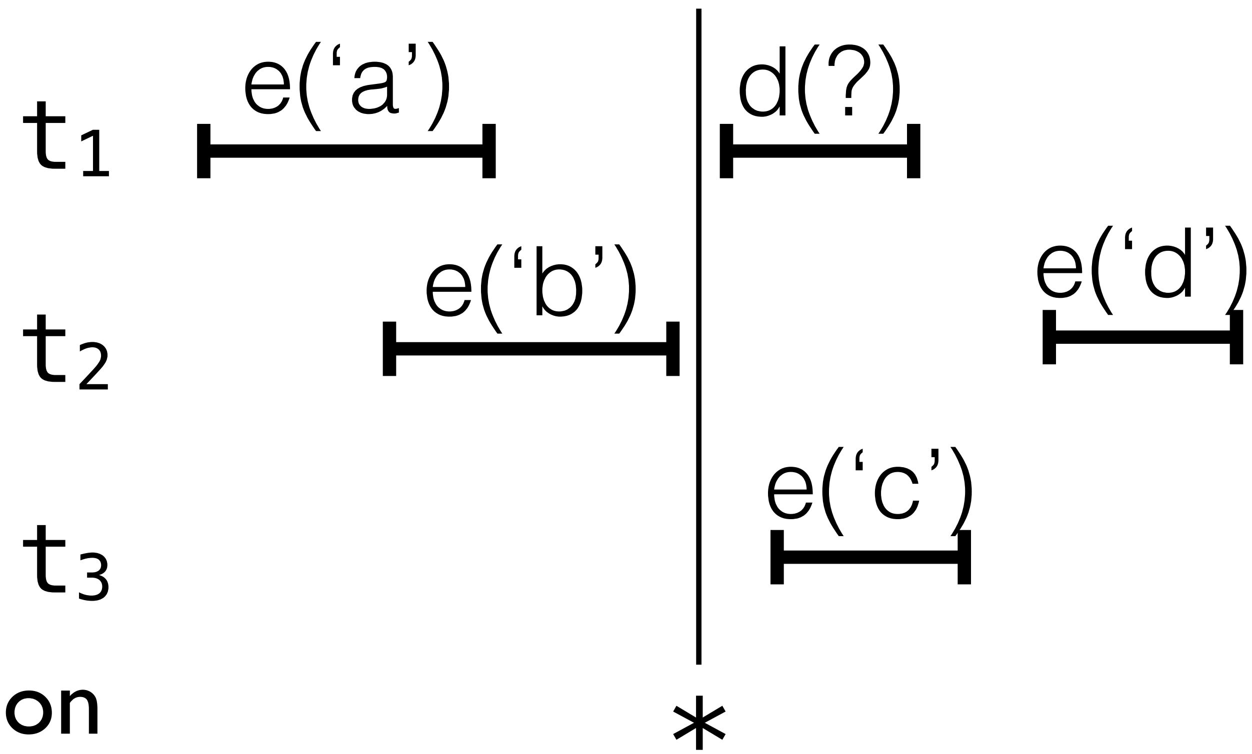
concrete history



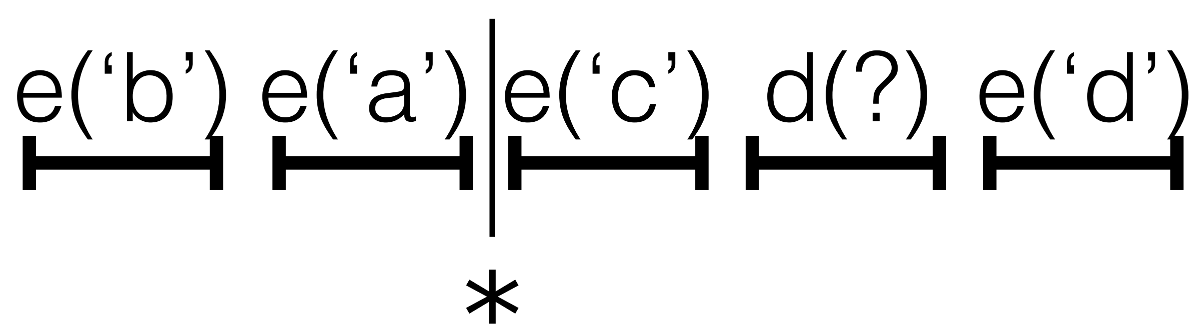
- by the moment (*) we still may not know where the linearization points of e('a') and e('b') are

Problem

for every concrete history

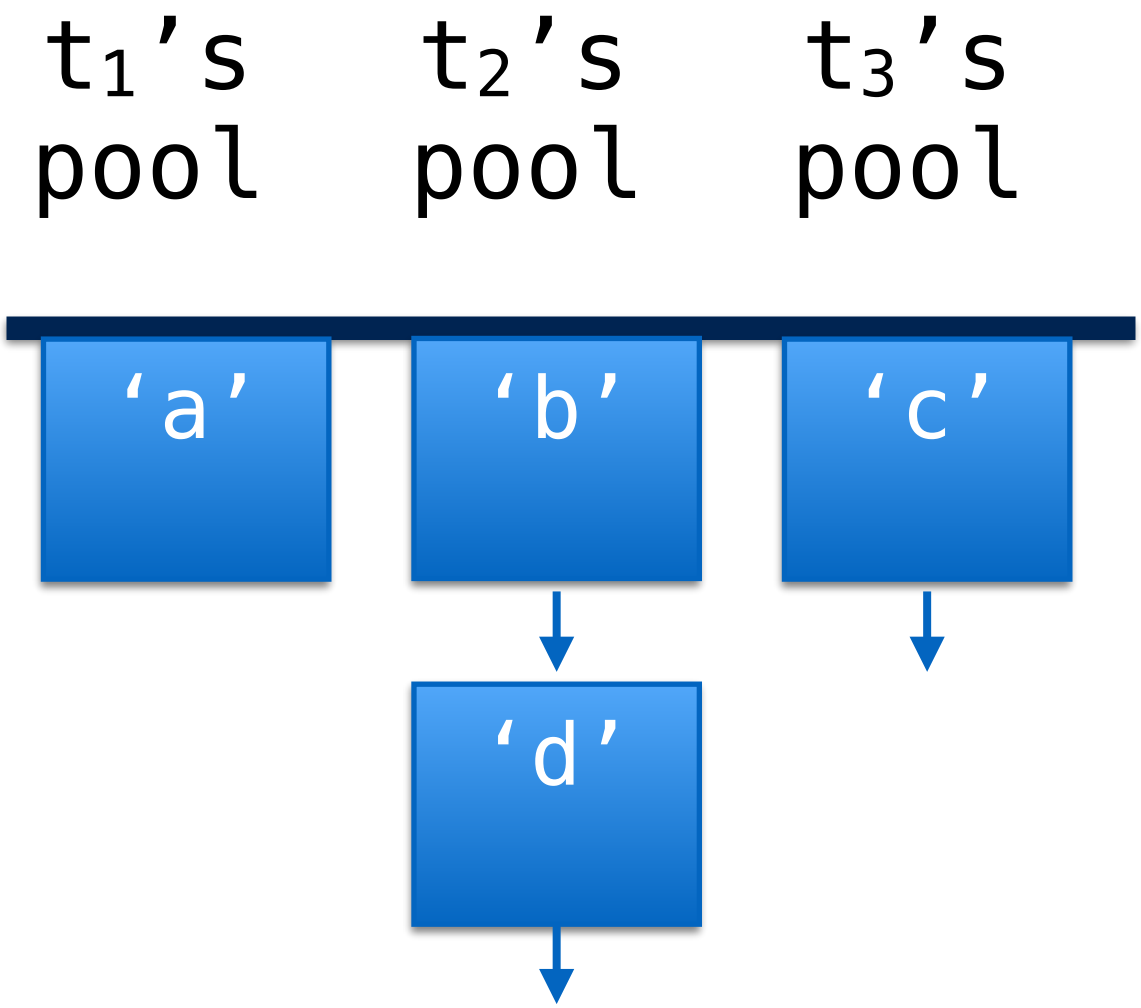


find a linearization



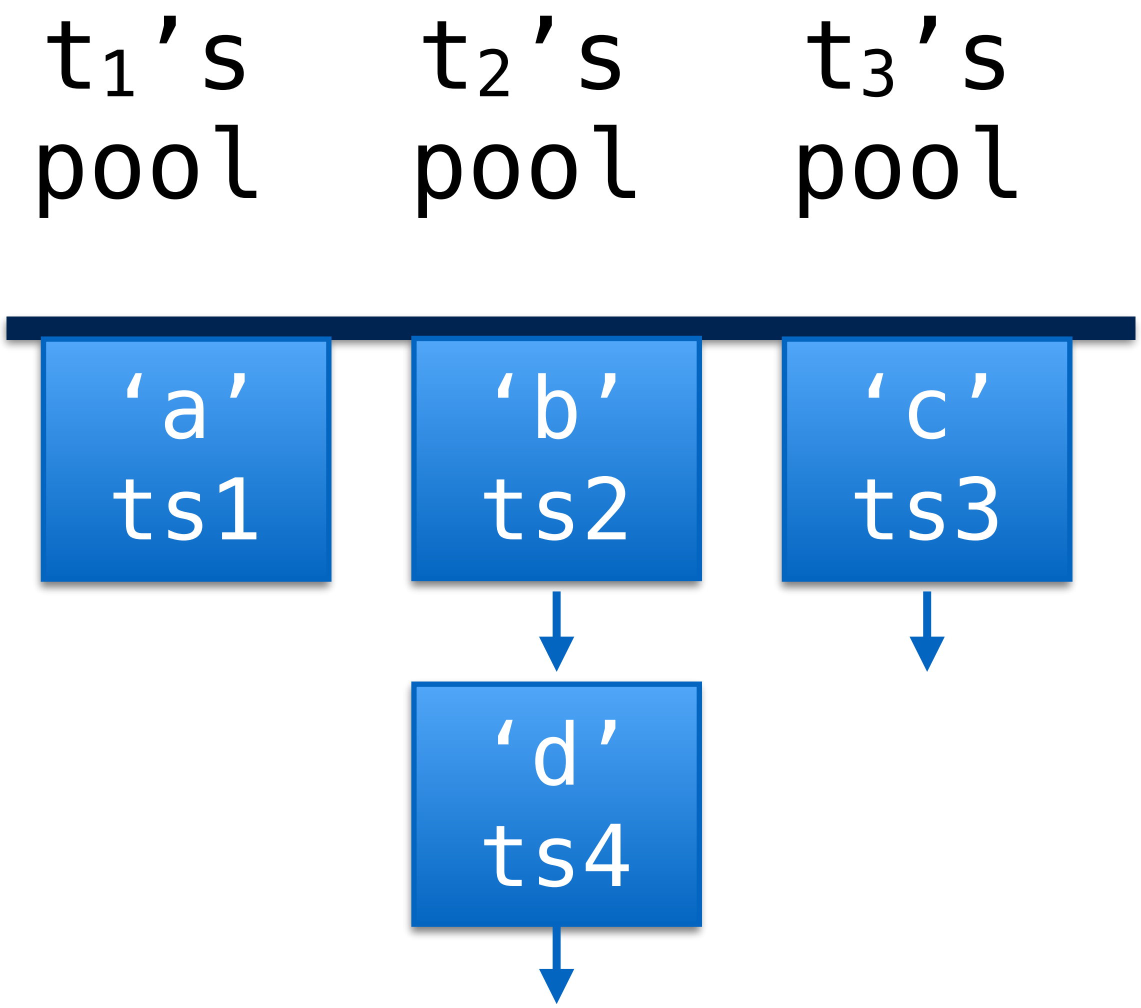
? = 'b'

The TS queue



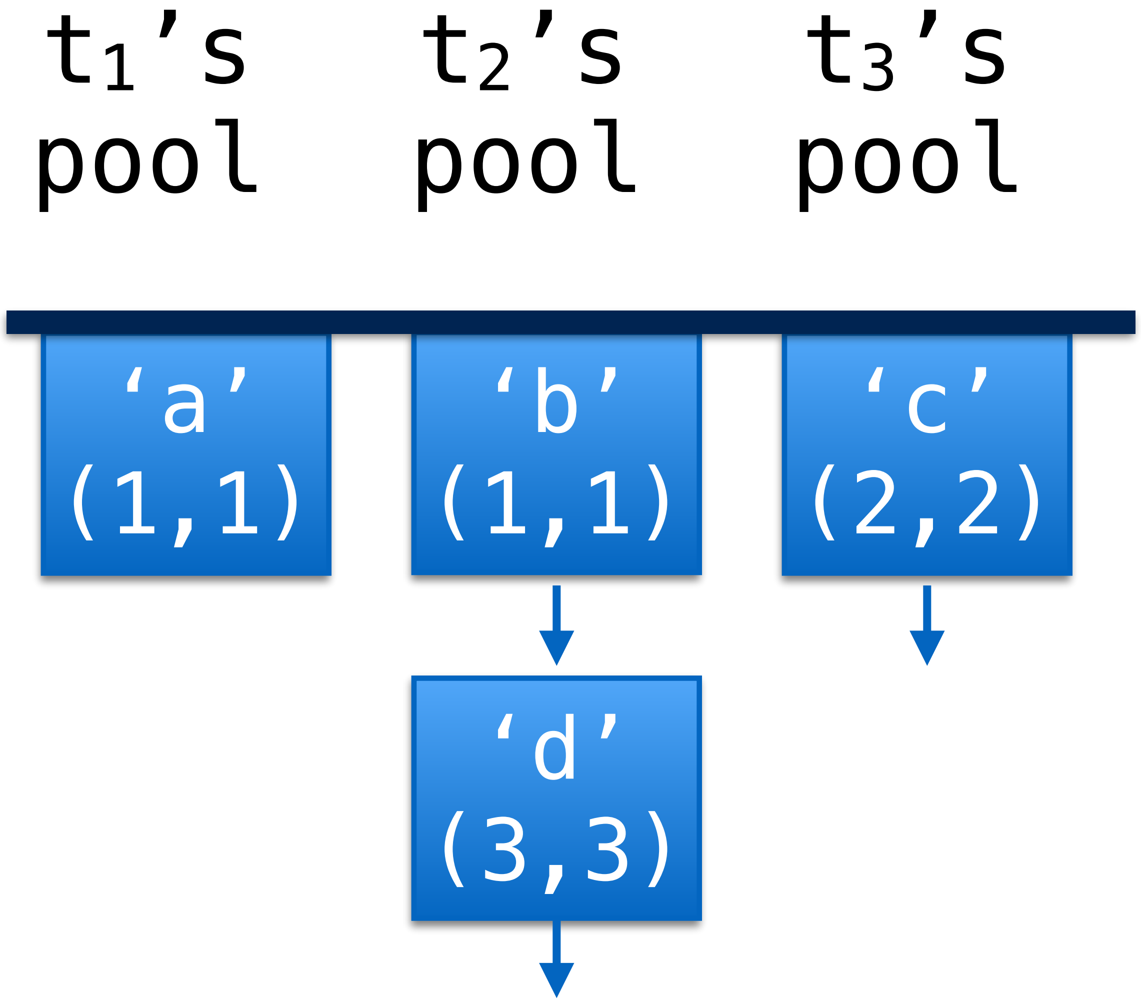
- each thread has its own pool
- pool is an abstract sequence
- (single producer multiple consumers)

The TS queue




```
enqueue(Val v) {
  ts := newTimestamp();
  insert(this_thread, v, ts);
}
```

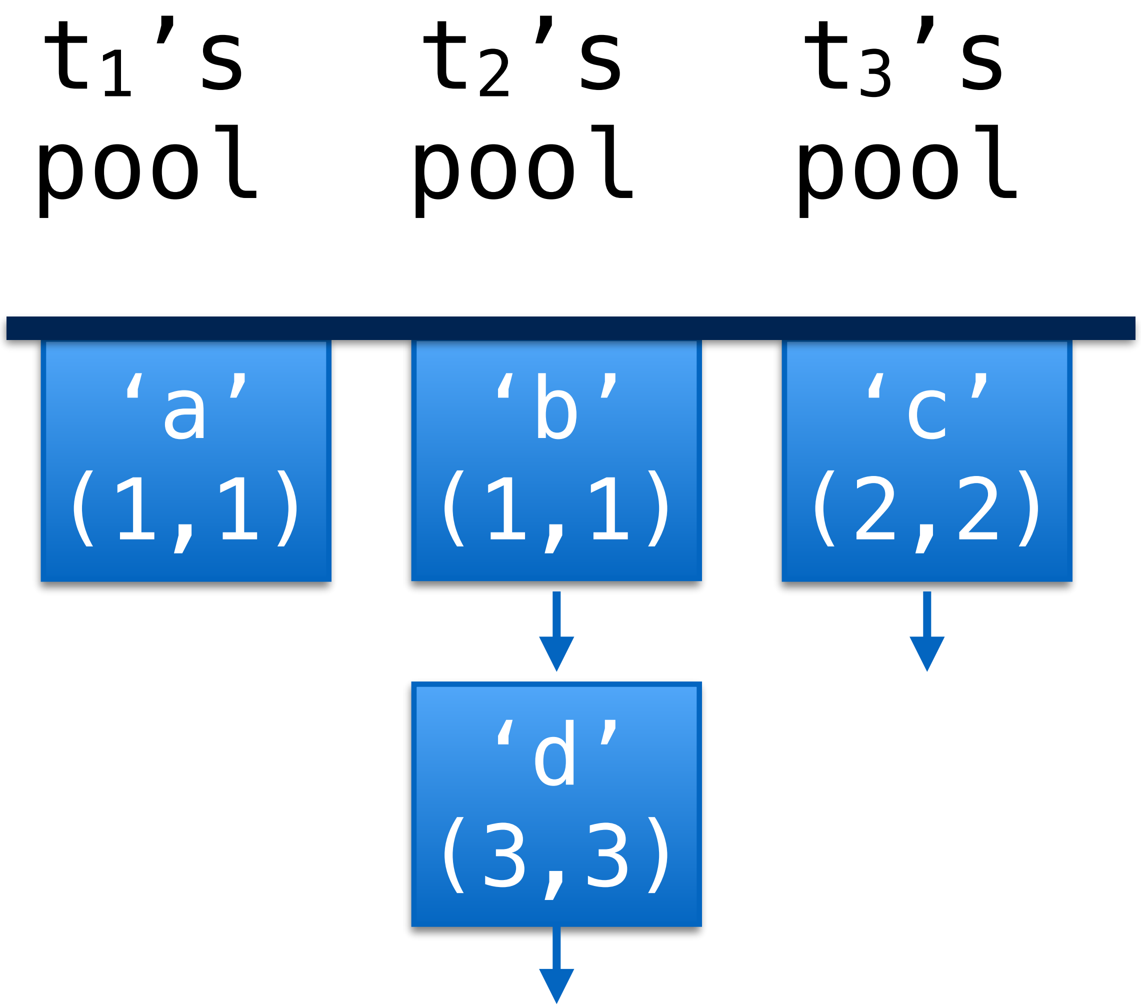
Timestamps




```
enqueue(Val v) {
  ts := newTimestamp();
  insert(this_thread, v, ts);
}
```

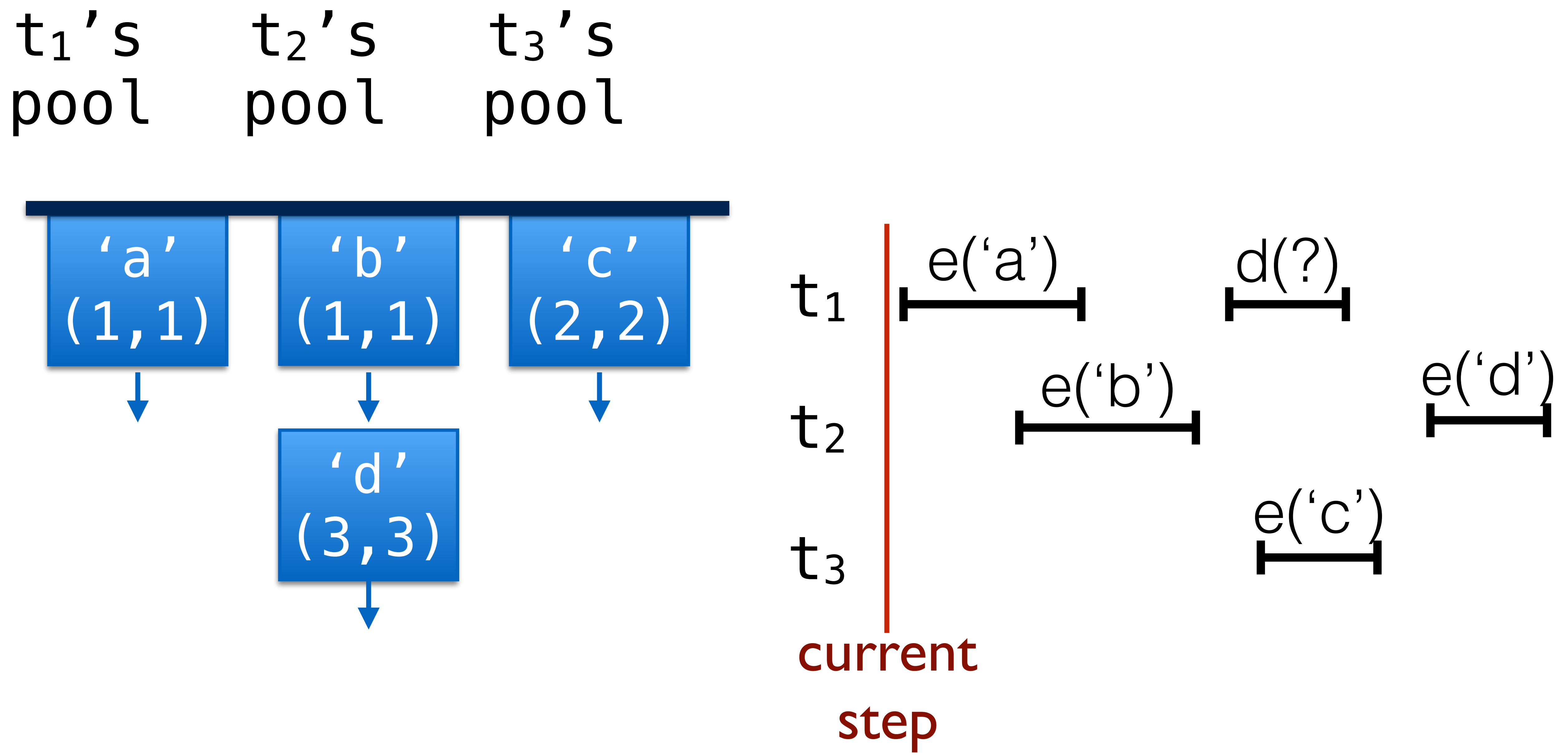
- timestamps = intervals
- (a, b) < (c, d) iff b < c

The TS queue



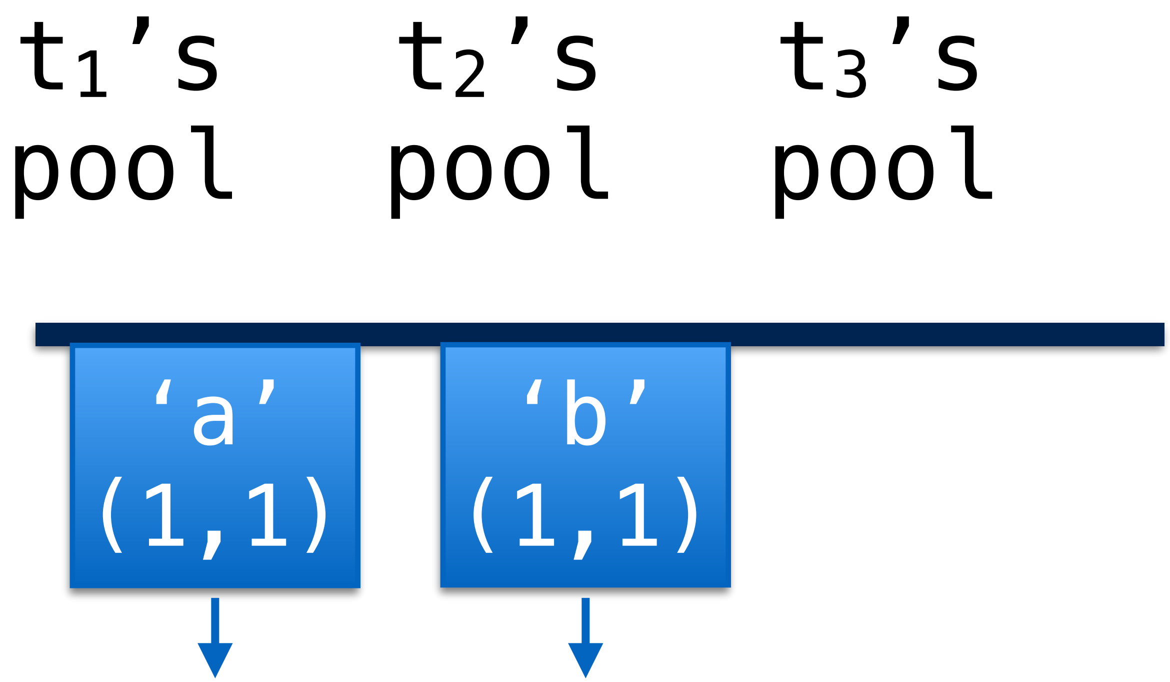

```
int counter = 1;

TS newTimestamp() {
   int l := counter;
   if CAS(counter, l, l+1)
      r := l;
   else
      r := counter-1;
   return (l, r);
}
```

The TS queue

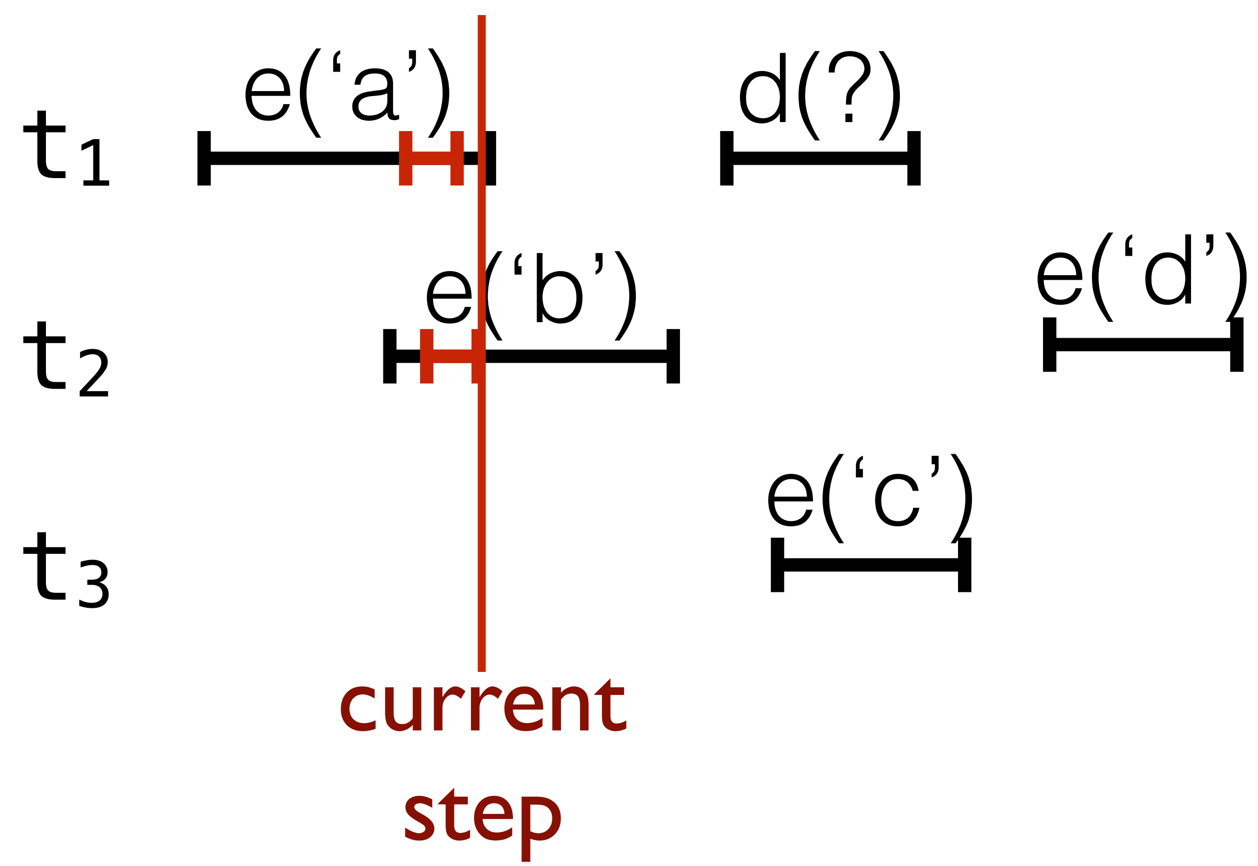


t1's pool
t2's pool
t3's pool
'a' (1,1)
'b' (1,1)
'c' (2,2)
'd' (3,3)
t1
t2
t3
e('a')
d(?)
e('b')
e('d')
e('c')
current step

The TS queue

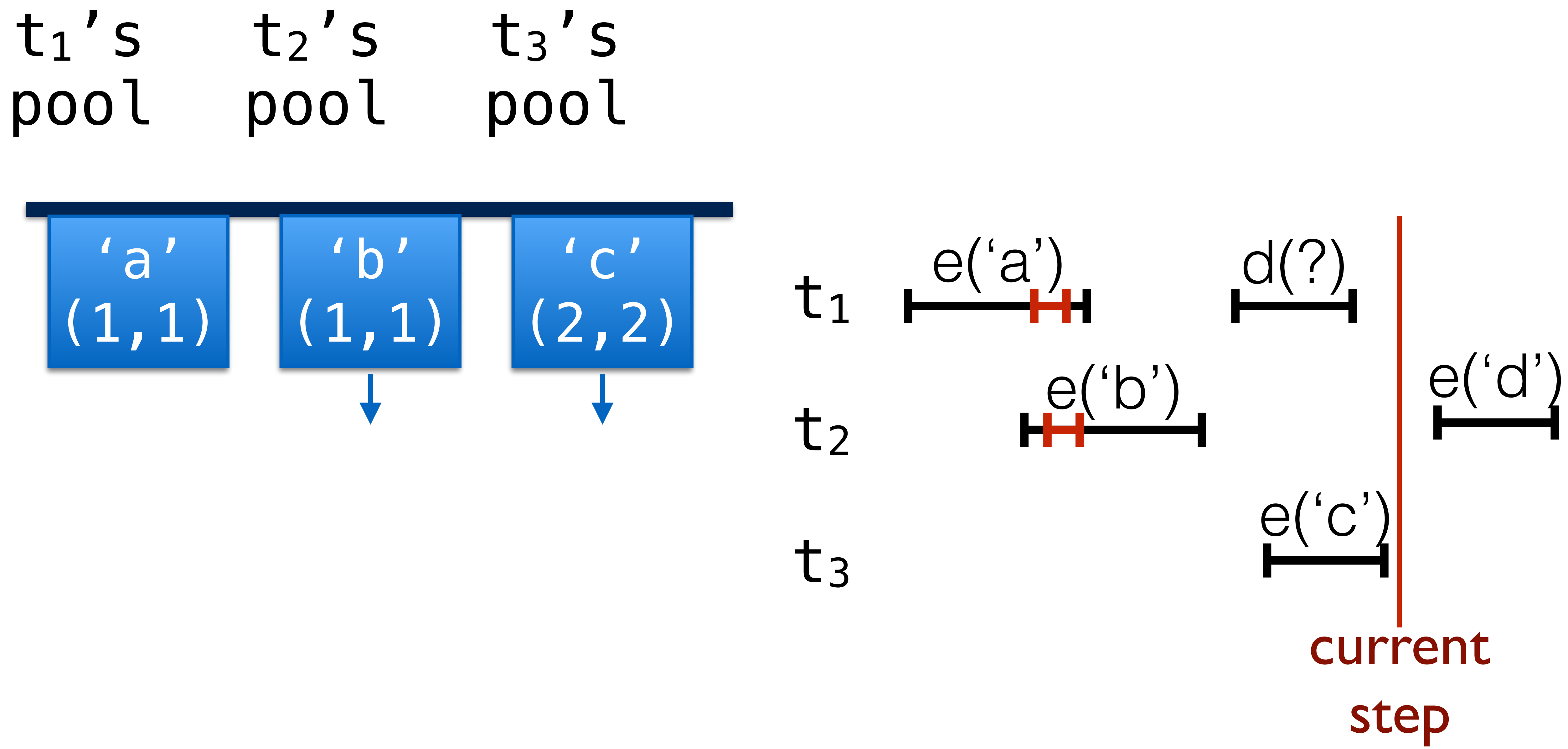


t1's pool
t2's pool
t3's pool
'a' (1,1)
'b' (1,1)



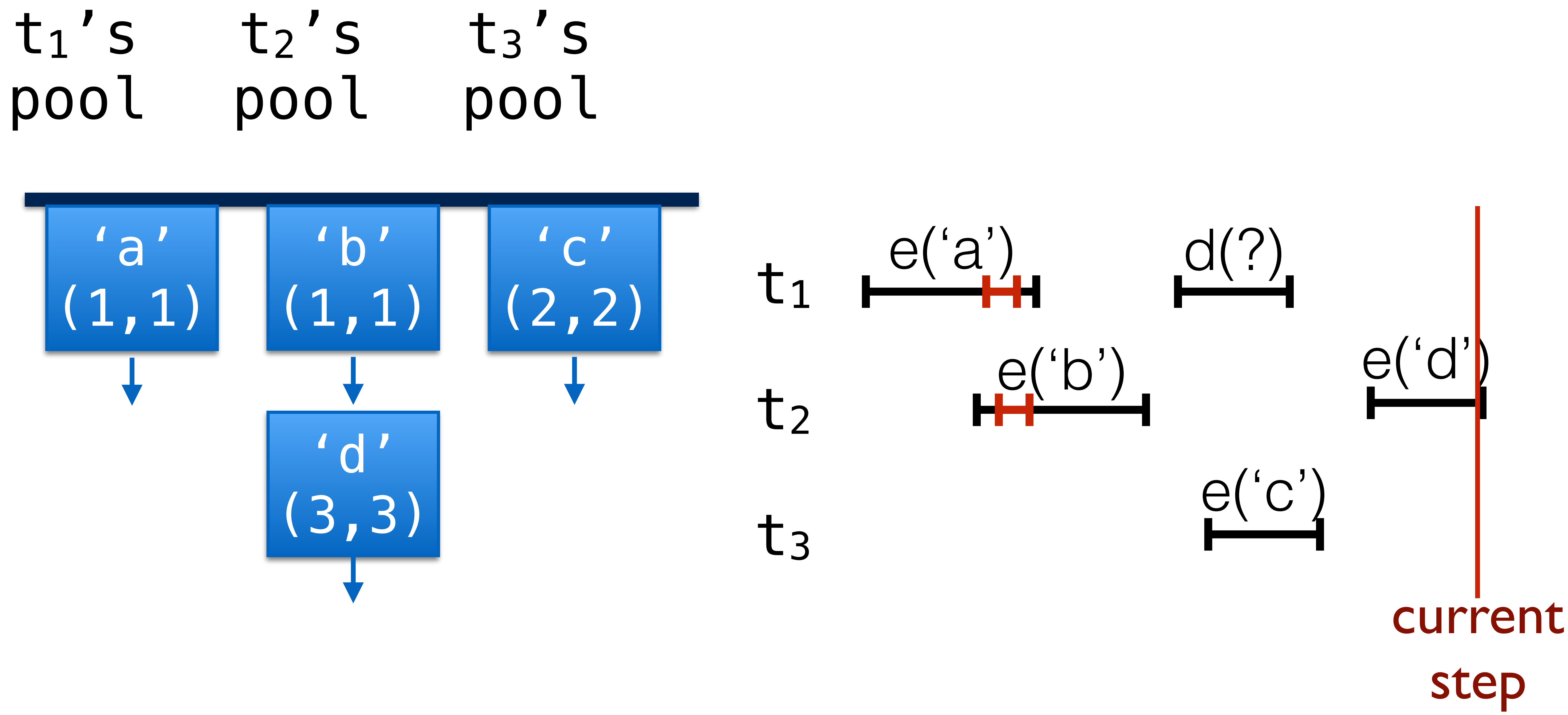
t1
t2
t3
e('a')
d(?)
e('b')
e('d')
e('c')
current step

The TS queue



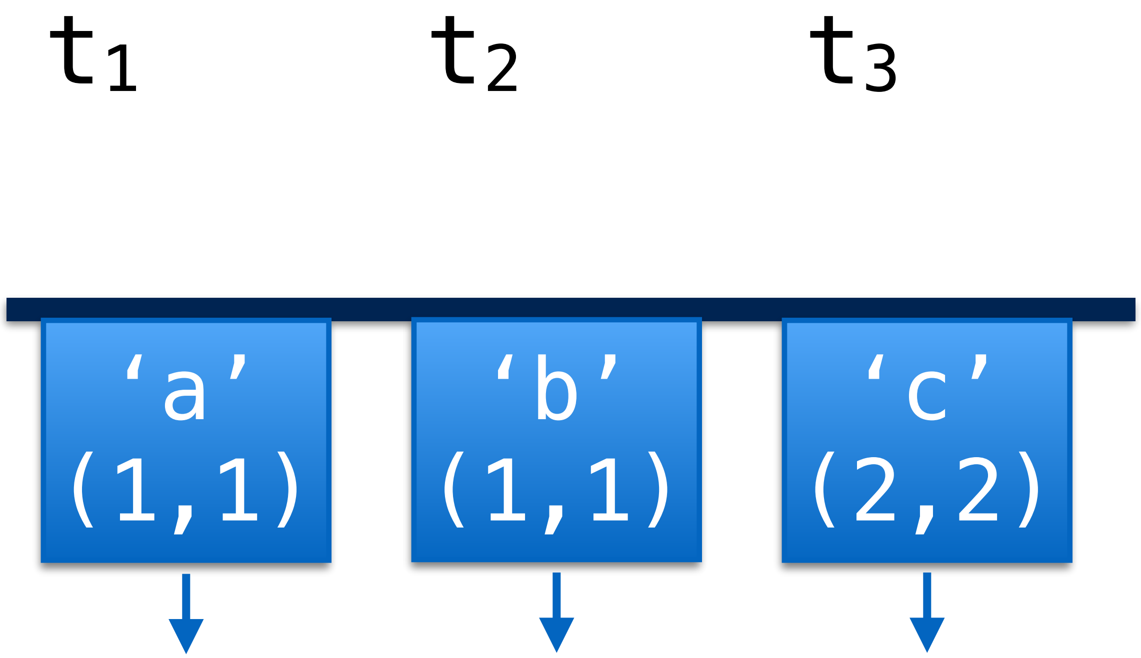
t1's pool
t2's pool
t3's pool
'a' (1,1)
'b' (1,1)
'c' (2,2)
t1
t2
t3
e('a')
d(?)
e('b')
e('d')
e('c')
current step

The TS queue



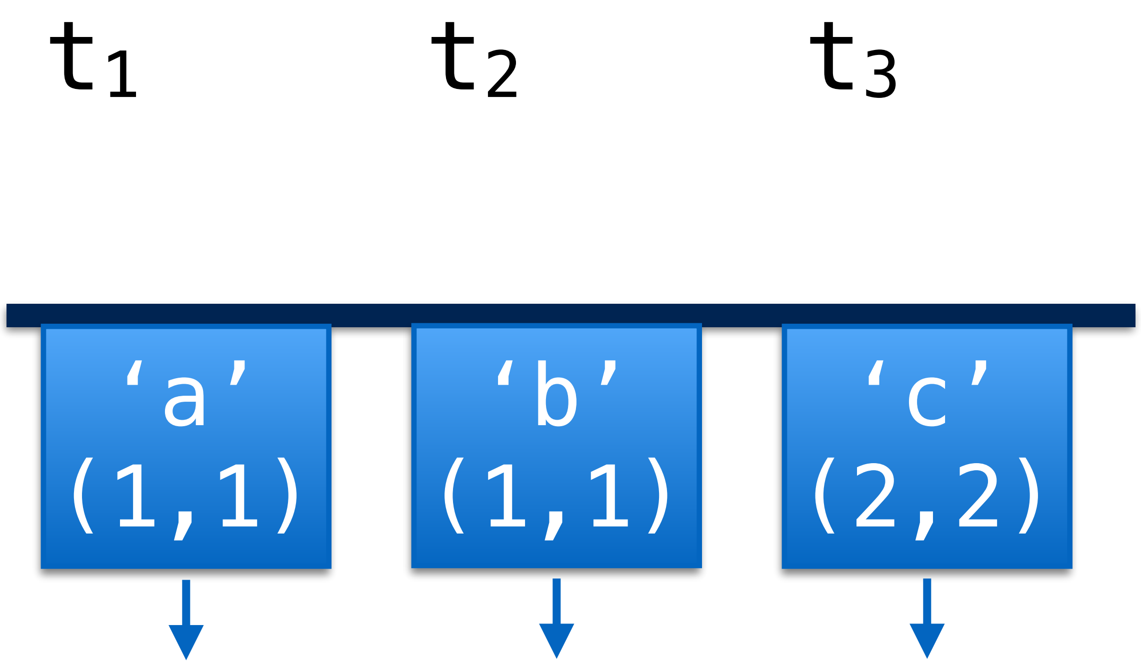
t1's pool
t2's pool
t3's pool
'a' (1,1)
'b' (1,1)
'c' (2,2)
'd' (3,3)
t1
t2
t3
e('a')
d(?)
e('b')
e('d')
e('c')
current step

The TS queue




```
Val dequeue() {
  do {
    for each pool do {
      look at the top node;
      update the candidate
              for removal;
    }
    if (there is a candidate)
      try removing it and
      returning its value;
  } while (true);
}
```

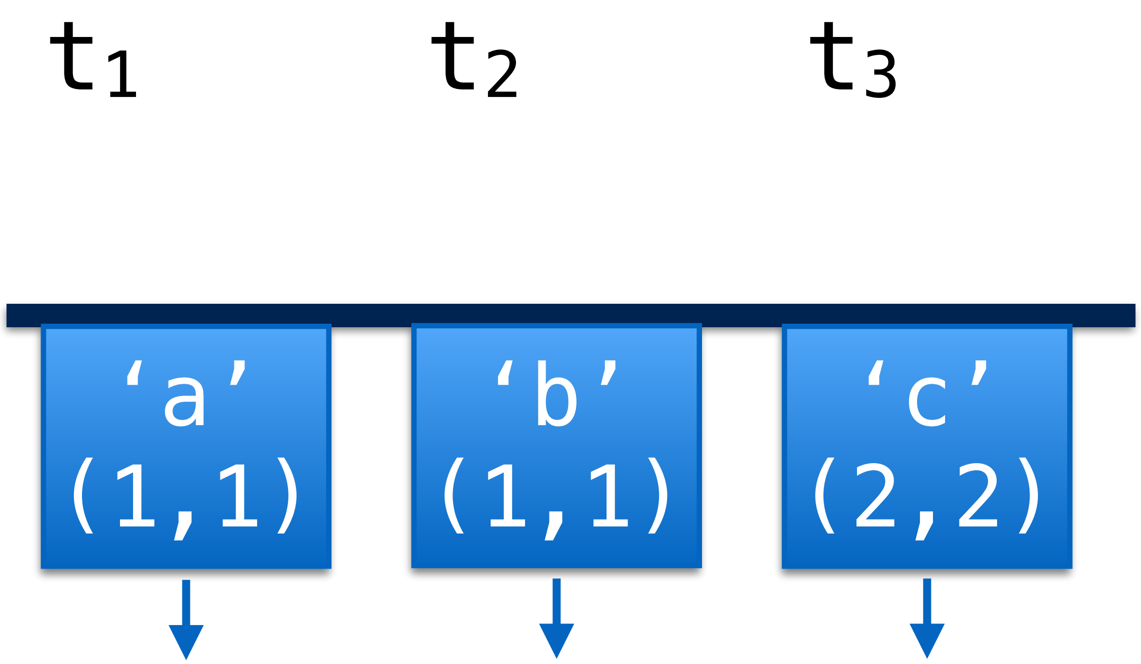
The TS queue




```
Val dequeue() {
  do {
    for each pool do {
      look at the top node;
      update the candidate
               for removal;
    }
    if (there is a candidate)
      try removing it and
      returning its value;
  } while (true);
}
```

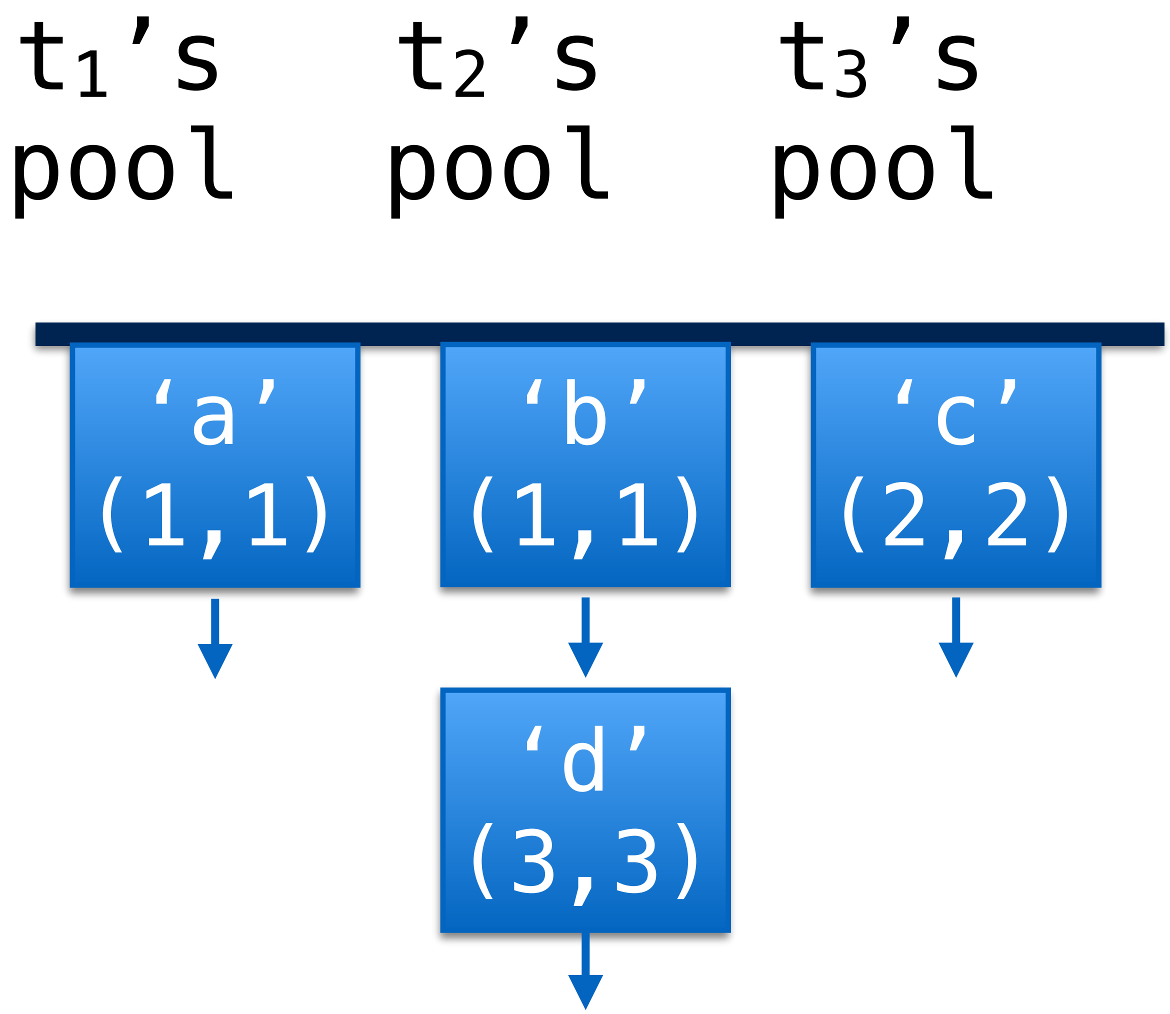
by choosing the smallest timestamp

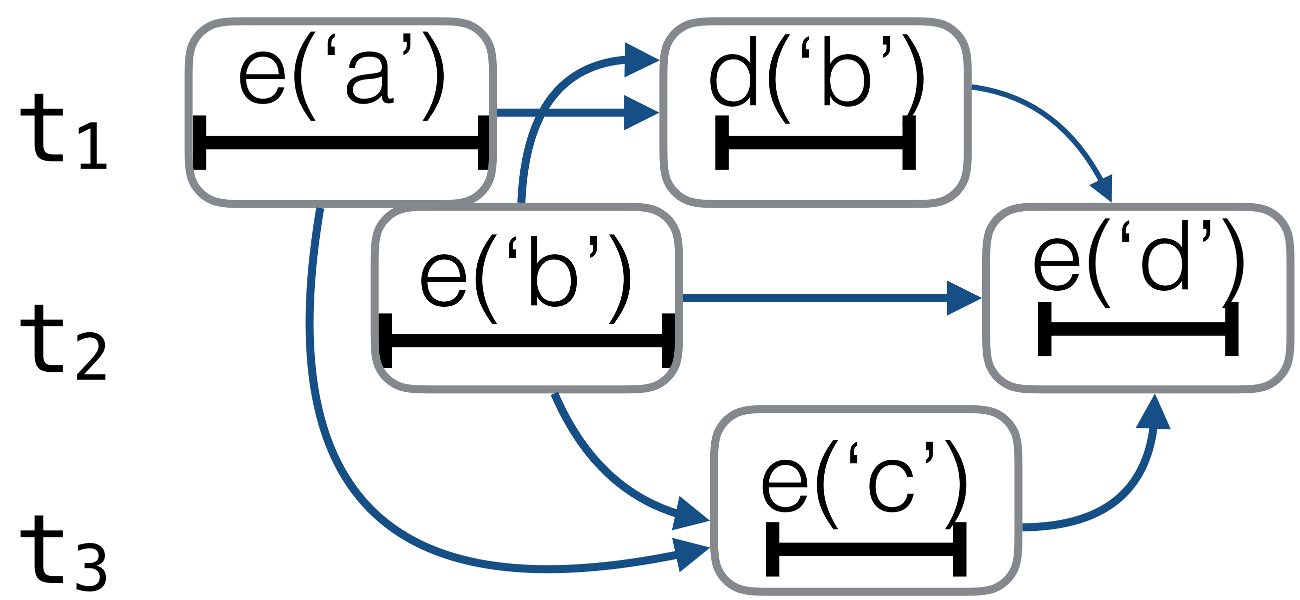
The TS queue




```
Val dequeue() {
  do {
    for each pool do {
      look at the top node;
      update the candidate
                for removal;
    }
    if (there is a candidate)
      try removing it and
      returning its value;
  } while (true);
}
```

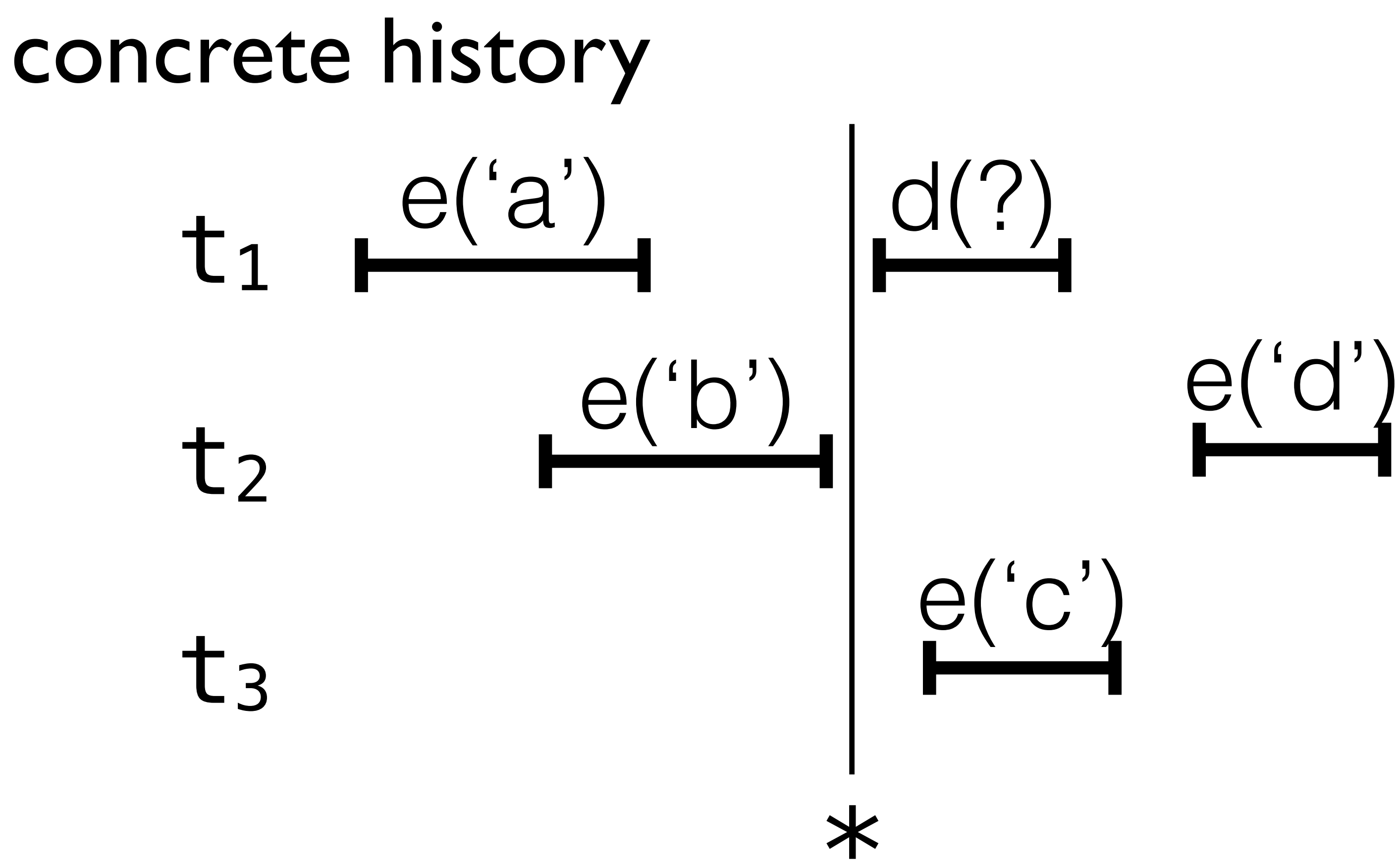
Why smallest timestamp?





- the connection with the real-time order
- e1 —rt—> e2 ==> timestamp(e1) < timestamp(e2)

Problem



- by the moment (*) we still may not know where the linearization points of e(‘a’) and e(‘b’) are

Our technique

- ✓ We alter the standard proof technique to support late choice via partial orders
- ✓ Our proof technique is implemented in a program logic
- ✓ Examples: the TS queue, the Herlihy-Wing queue, the Optimistic Set

Abstract histories

- For each history of a data structure we construct a matching abstract history (instead of a linearization)
- An “abstract” history (E, R):
  - E — a set of events [eid: (tid, op, arg, rval)]
  - R — a partial order

Abstract histories

- For each history of a data structure we construct a **matching** abstract history (instead of a linearization)

1. the abstract history extends the real-time order
2. all linearizations meet the sequential spec

Abstract histories

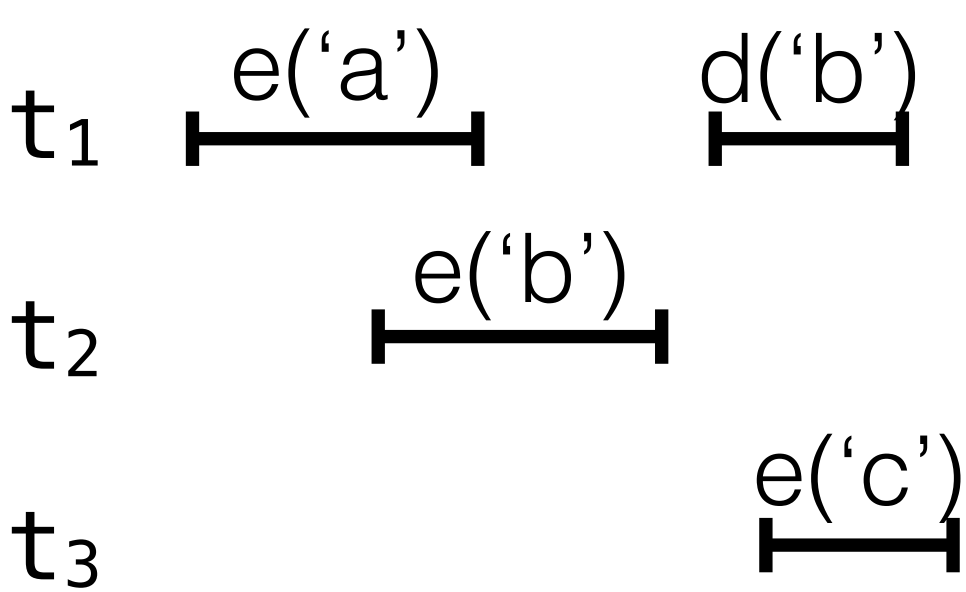
- We prove a number of invariant properties including:
  - all linearizations meet the sequential spec
  - for all enqueues e, e' with values in the data structure:
    - e —> e' ==> timestamp(e) < timestamp(e')
  - …

Commitment points

- Abstract histories are constructed:
  - by adding new events
  - by adding more edges into the partial order
  - by assigning a return value to an event

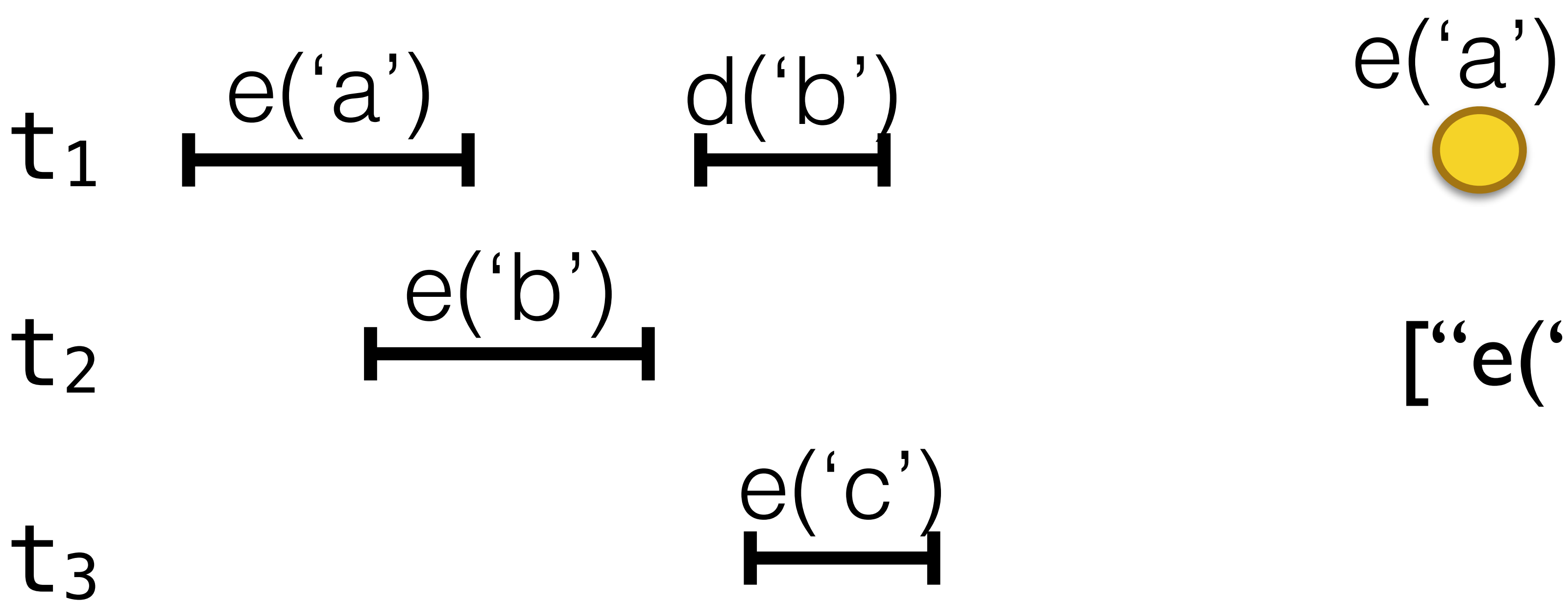
Commitment points

- At the beginning of each operation, a fresh event and real-time order edges are added to the abstract history



Commitment points

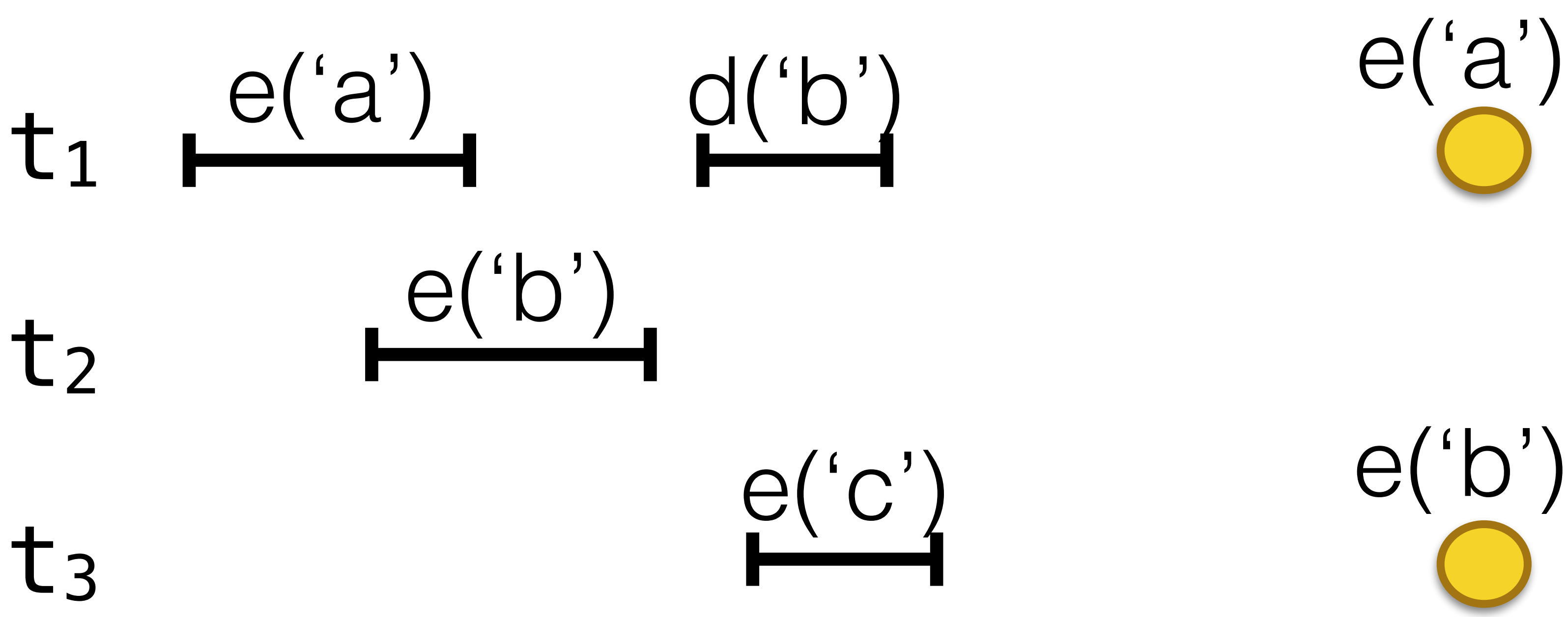
- At the beginning of each operation, a fresh event and real-time order edges are added to the abstract history



[“e(‘a’)” : t1, enq, ‘a’, TODO]

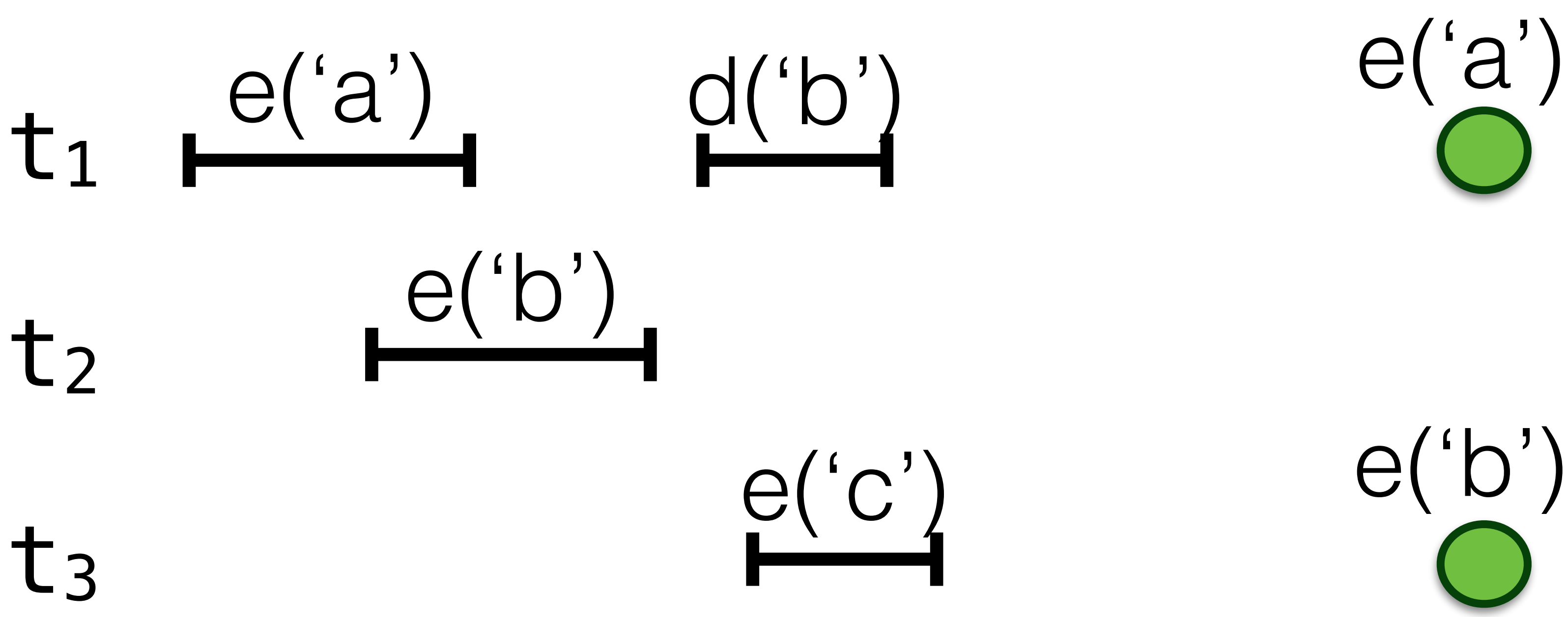
Commitment points

- At the beginning of each operation, a fresh event and real-time order edges are added to the abstract history



Commitment points

- At the beginning of each operation, a fresh event and real-time order edges are added to the abstract history



- events get completed by the end of the operations

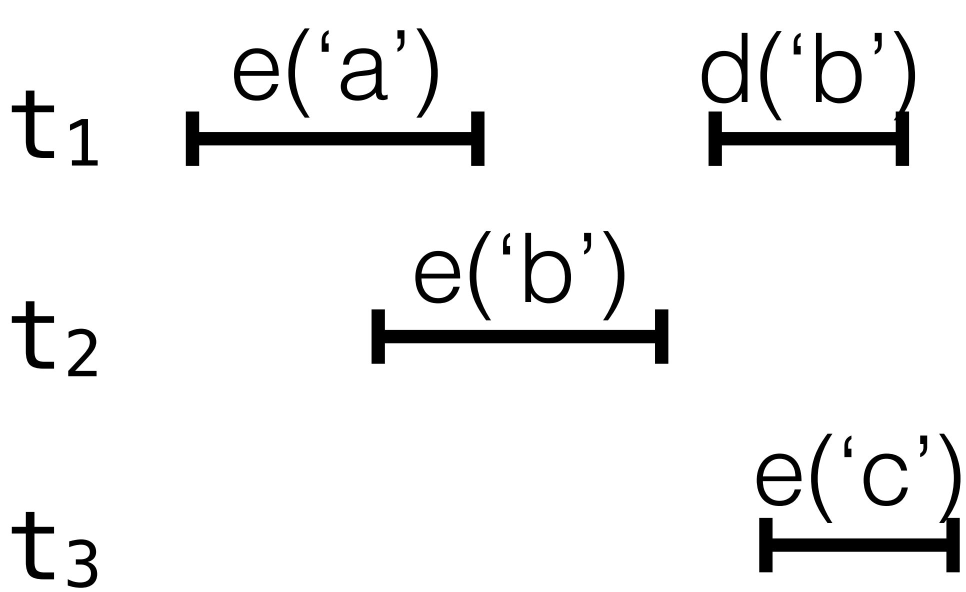
Enqueue’s commitment point

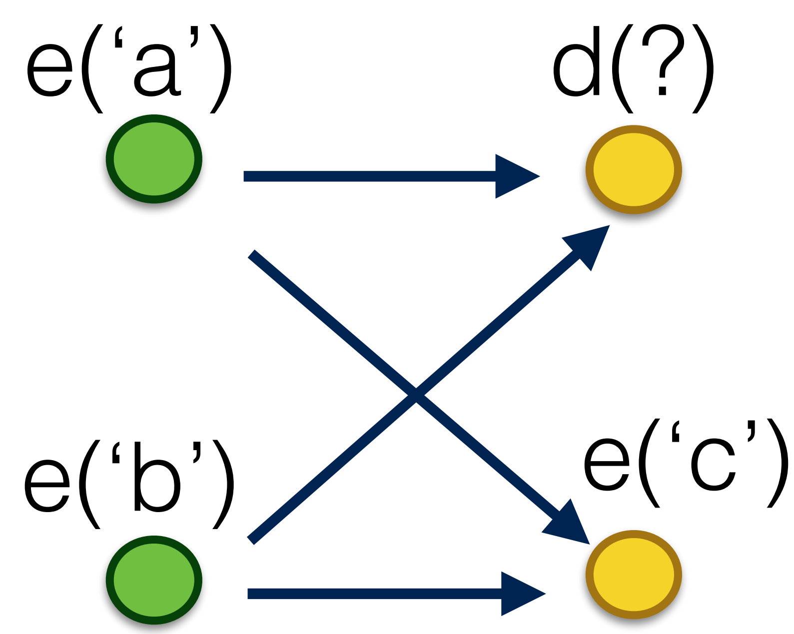
```
enqueue(Val v) {
  ts := newTimestamp();
  atomic {
    insert(this_thread, v, ts);
    E(this_event).rval := DONE;
    G[this_event] := ts;
  }
}
```

- Ghost state G is a map from events and timestamps
- Helps to establish a bijection between events and elements of the data structure

Commitment points

- At the beginning of each operation, a fresh event and real-time order edges are added to the abstract history

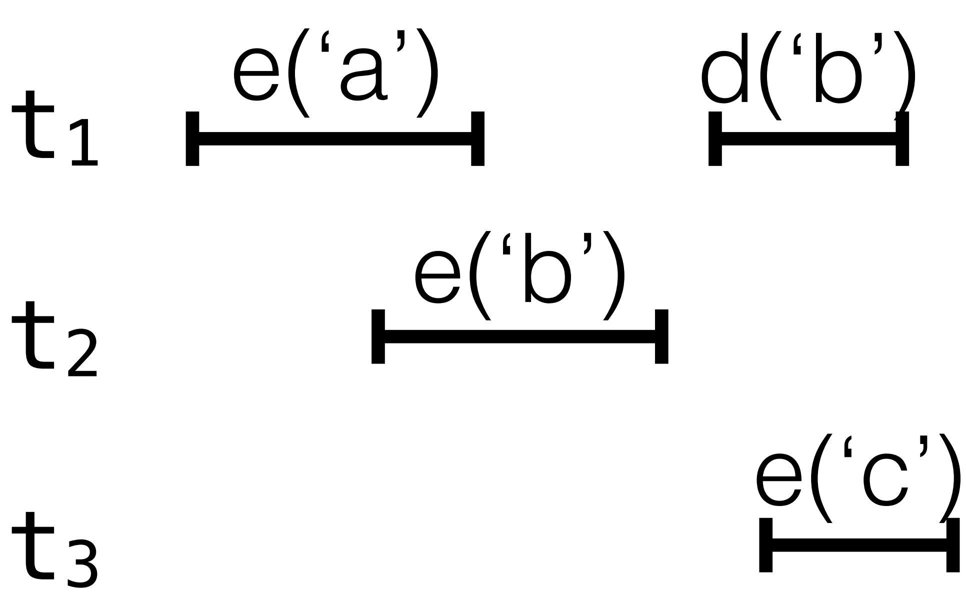


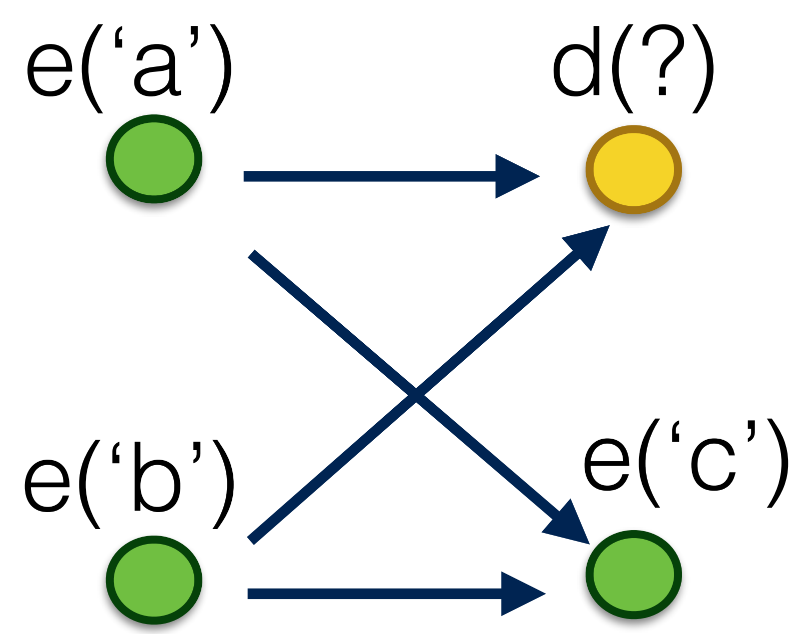


- edges from the completed events are added

Commitment points

- The late choice is resolved by a dequeue with the FIFO policy in mind

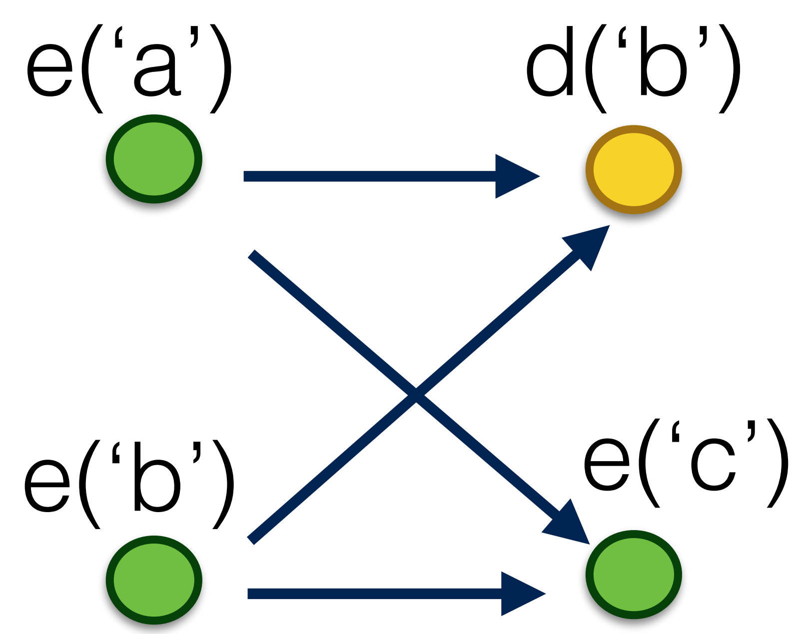




Commitment points

- The late choice is resolved by a dequeue with the FIFO policy in mind

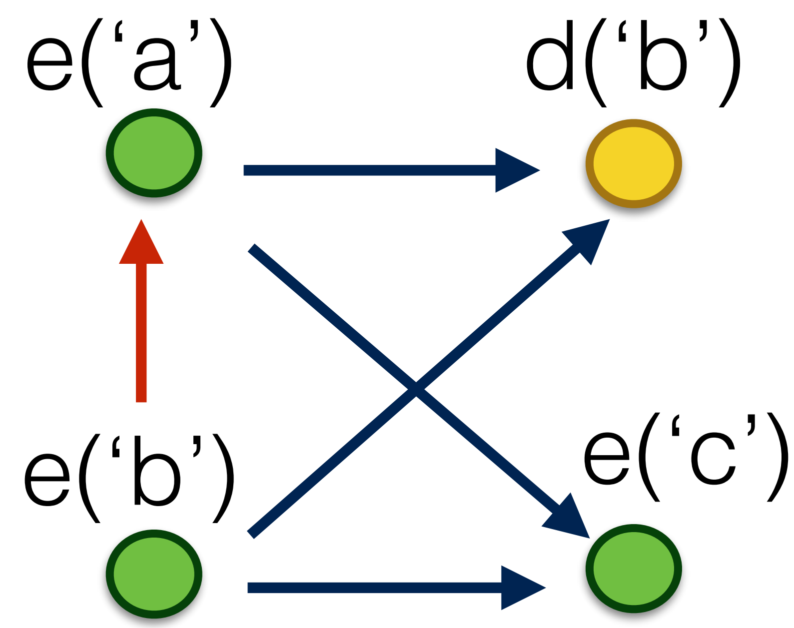
1. e(‘b’) e(‘a’) d(‘b’) e(‘c’)
2. e(‘b’) e(‘a’) e(‘c’) d(‘b’)
3. e(‘a’) e(‘b’) d(‘b’) e(‘c’)
4. e(‘a’) e(‘b’) e(‘c’) d(‘b’)



Commitment points

- The late choice is resolved by a dequeue with the FIFO policy in mind

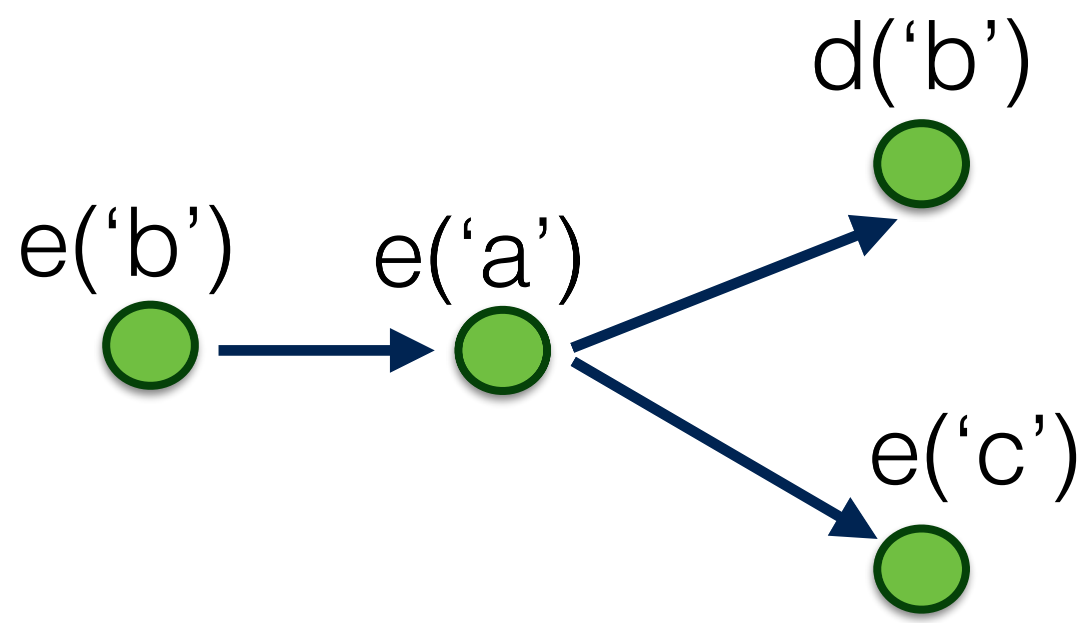
1. e(‘b’) e(‘a’) d(‘b’) e(‘c’)
2. e(‘b’) e(‘a’) e(‘c’) d(‘b’)
3. e(‘a’) e(‘b’) d(‘b’) e(‘c’)
4. e(‘a’) e(‘b’) e(‘c’) d(‘b’)



Commitment points

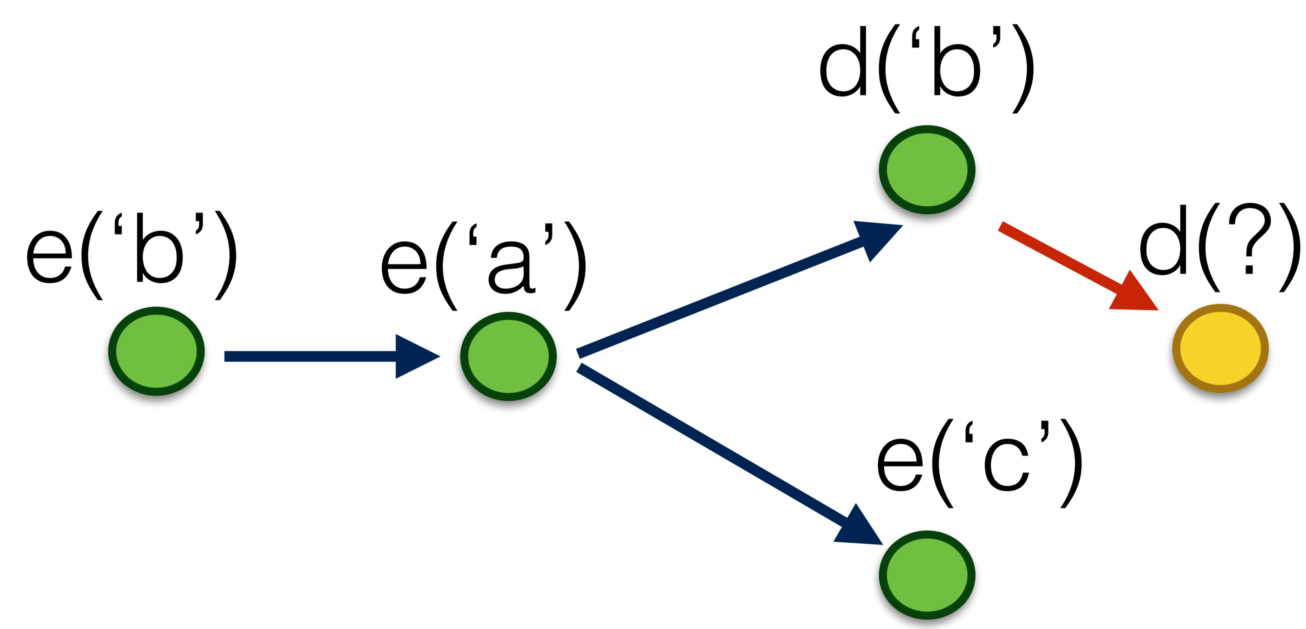
- The late choice is resolved by a dequeue with the FIFO policy in mind

1. e('b') e('a') d('b') e('c')
2. e('b') e('a') e('c') d('b')



Commitment points

- The late choice is resolved by a dequeue with the FIFO policy in mind



Dequeue’s commitment point

```
if (there is a candidate enq) {
  res := try removing enq;
  if (res != FAIL) {
    E(this_event).rval := res;
    R := (R
     U {(enq, this_event)}
     U {(enq, e’) | the value of e’ is in queue}
     U {(deq, d’} | d’ is an uncompleted dequeue})+;
 }
}
```

Proof obligations

- For each operation, come up with commitment points:
  - adding a new event and real-time order edges
  - by the end of operation, assign a return value
  - (optionally) extend the order
    - preserving "all linearizations meet seq. spec."
    - preserving acyclicity of the order

Conclusions

- The technique for proving linearizability of algorithms that are challenging with the linearization points method
- Examples: the TS queue, the Herlihy-Wing queue, the Optimistic Set
- Examples to do: the TS stack