MODULAR VERIFICATION OF CONCURRENCY-AWARE LINEARIZABILITY

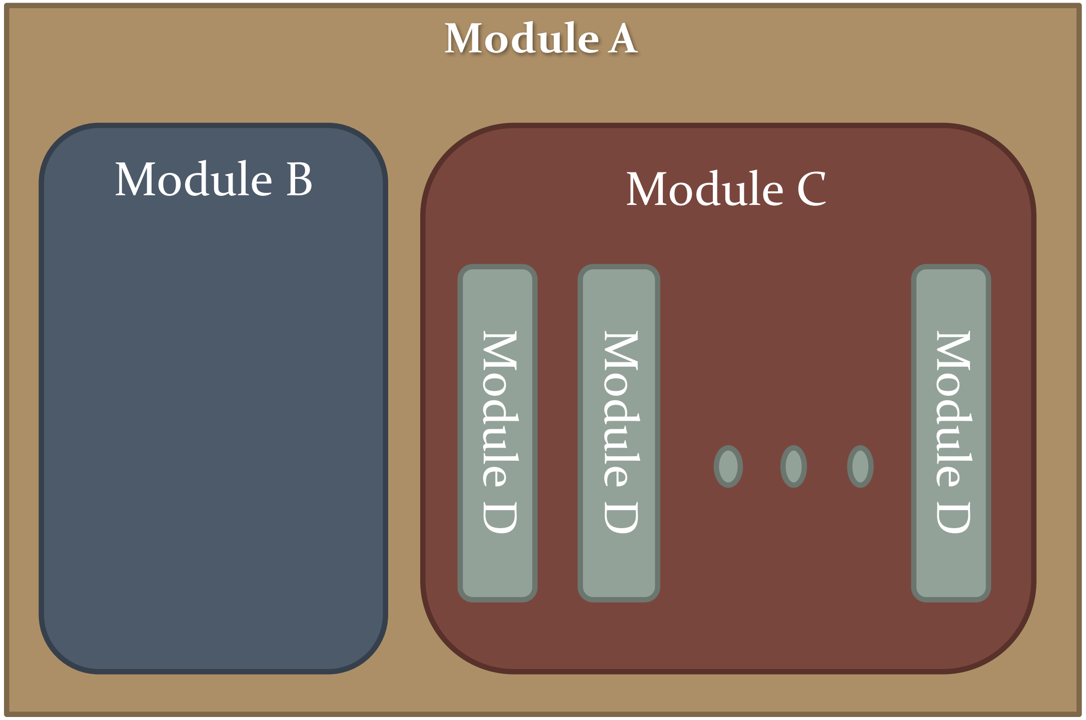
Nir Hemed[1], Noam Rinetzky[1], and Viktor Vafeiadis[2]

[1] Tel Aviv University, Israel
[2] MPI-SWS, Germany

DISC 2015

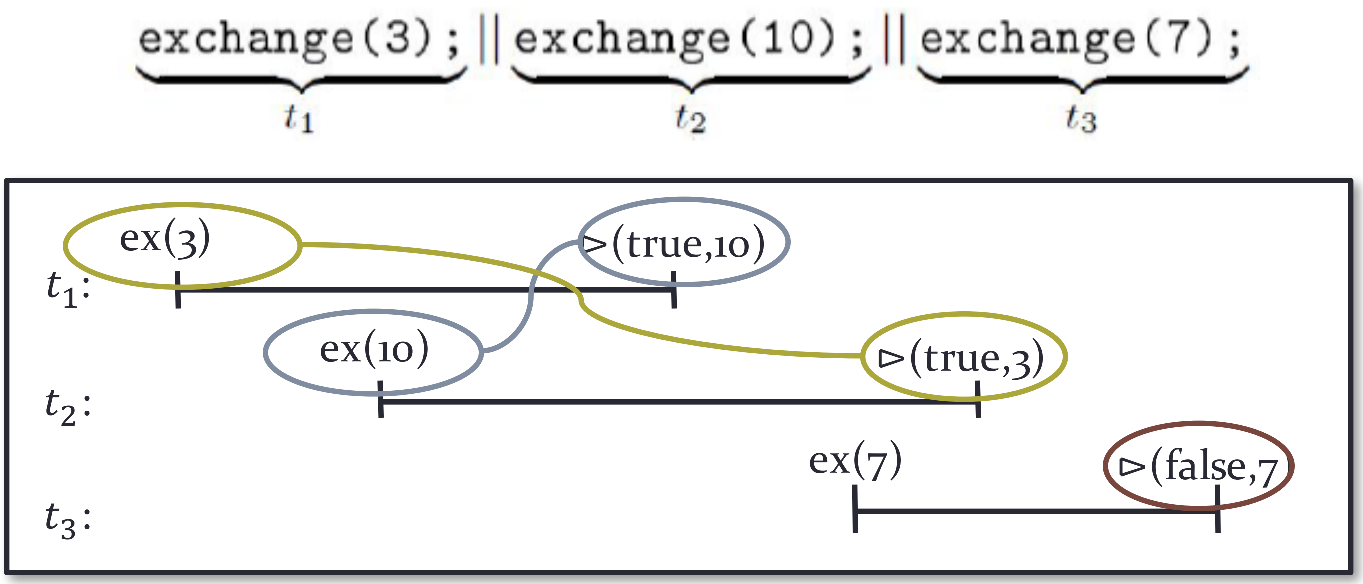
Goal I: Modular Reasoning

- Modular verification technique for concurrent objects



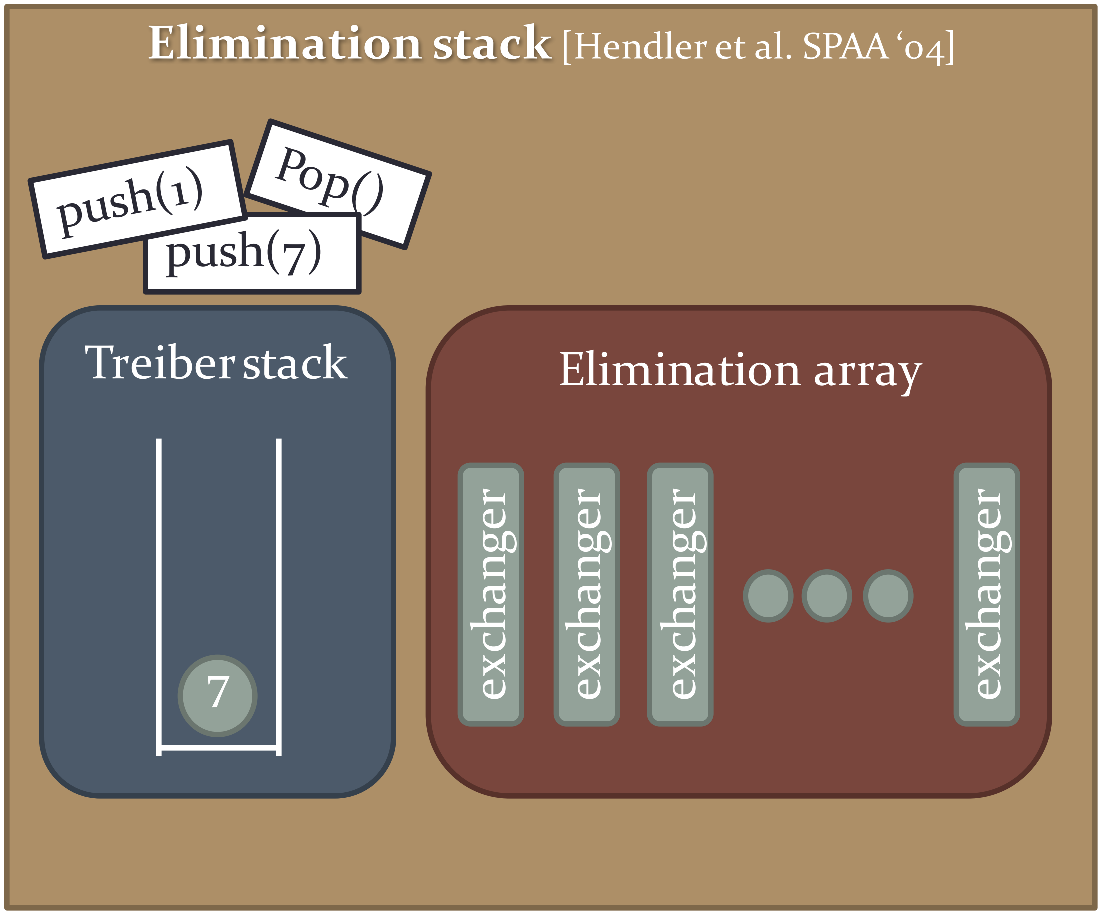
Goal II: Handle Concurrency-Aware objects

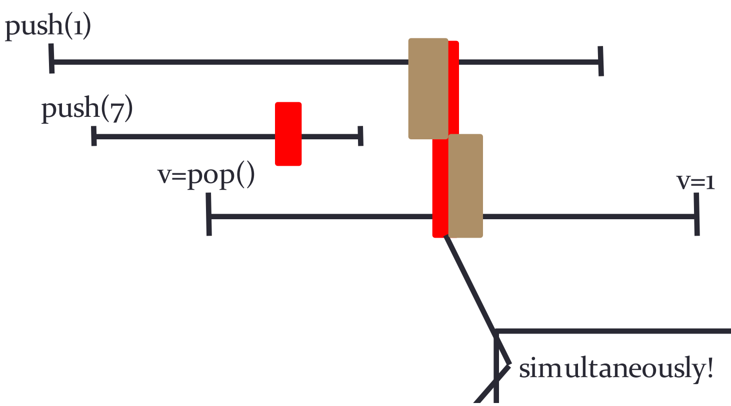
- Specifying Concurrency-Aware objects
  - Multiple operations linearize at the same point in time
- Example: java.util.concurrent.Exchanger
  - allow threads to pair up and swap elements



Verification challenge: Elimination Stack

push(1) || push(7) || v=pop()



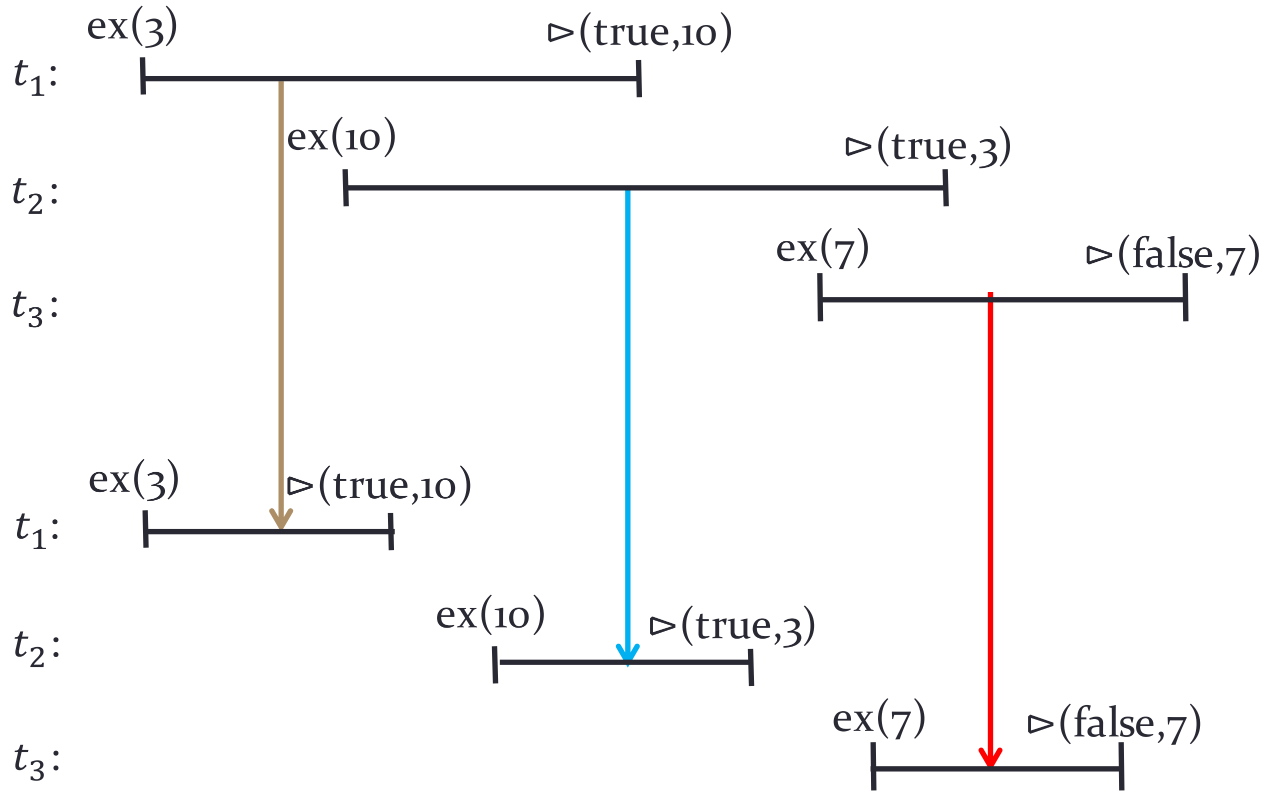


Research problems

- How do we define the behavior of **Concurrency-Aware** objects?
  - Multiple operations linearize at the same point in time
  - E.g., Exchanger, elimination array,...
- How do we provide a specification which is amenable for formal proofs?
- How do we reason about composed concurrent objects?
  - Information hiding- compositional reasoning
  - Mixing CA-objects and linearizable objects

Challenge I: specifying CA-objects

Linearizabilty?



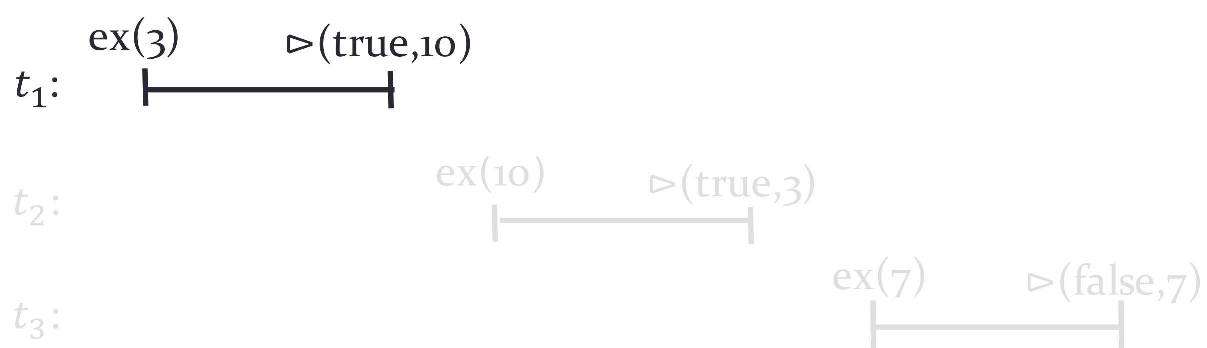
"Good specs.": intuitive, expressive,..., prefix-closed*, ...

* *SPEC is prefix-closed if* $\forall H, H_1, H_2 . H \in SPEC \wedge H = H_1 H_2 \Longrightarrow H_1 \in SPEC$

Sequential specification for Exchanger?

Sequential specifications for Exchanger are

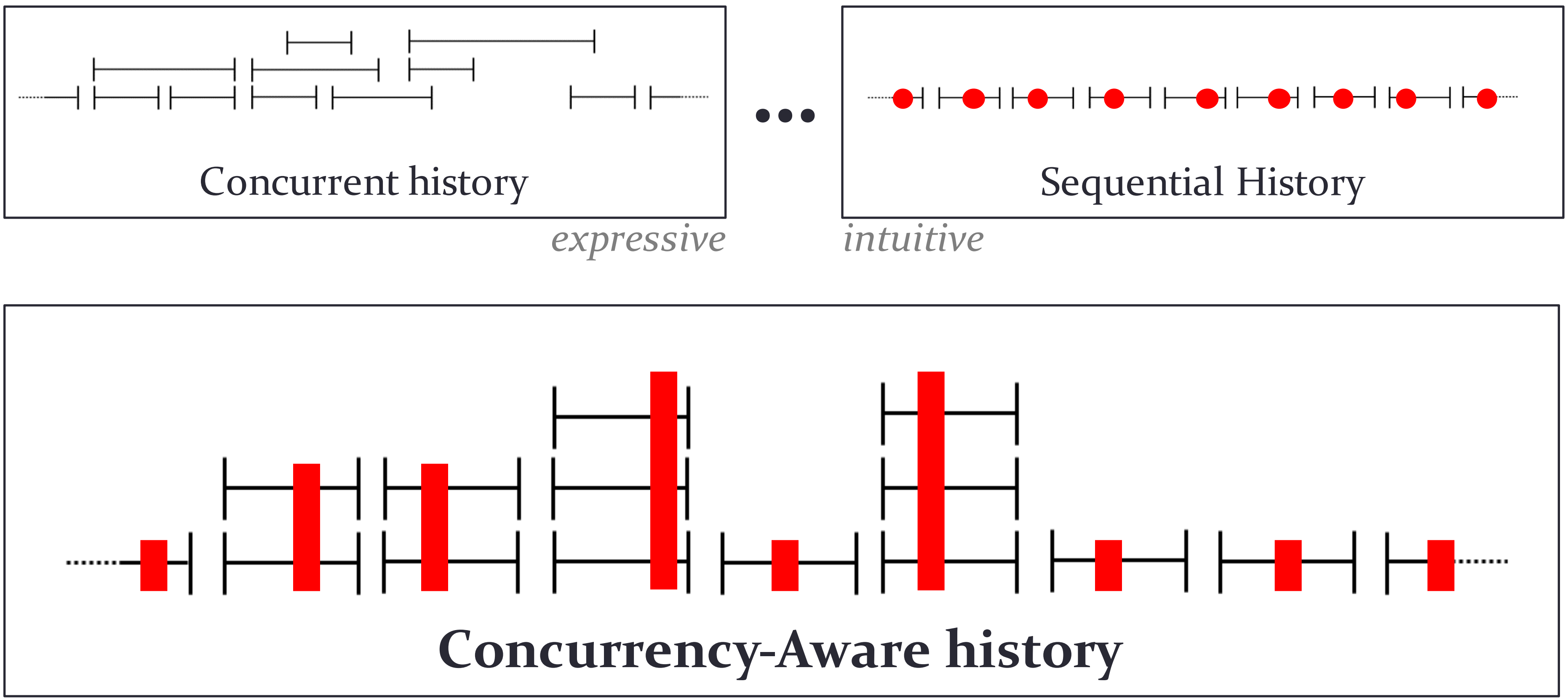
- *Too lax*
- *Too strict*



"Good specs.": intuitive, expressive,..., **prefix-closed***, ...

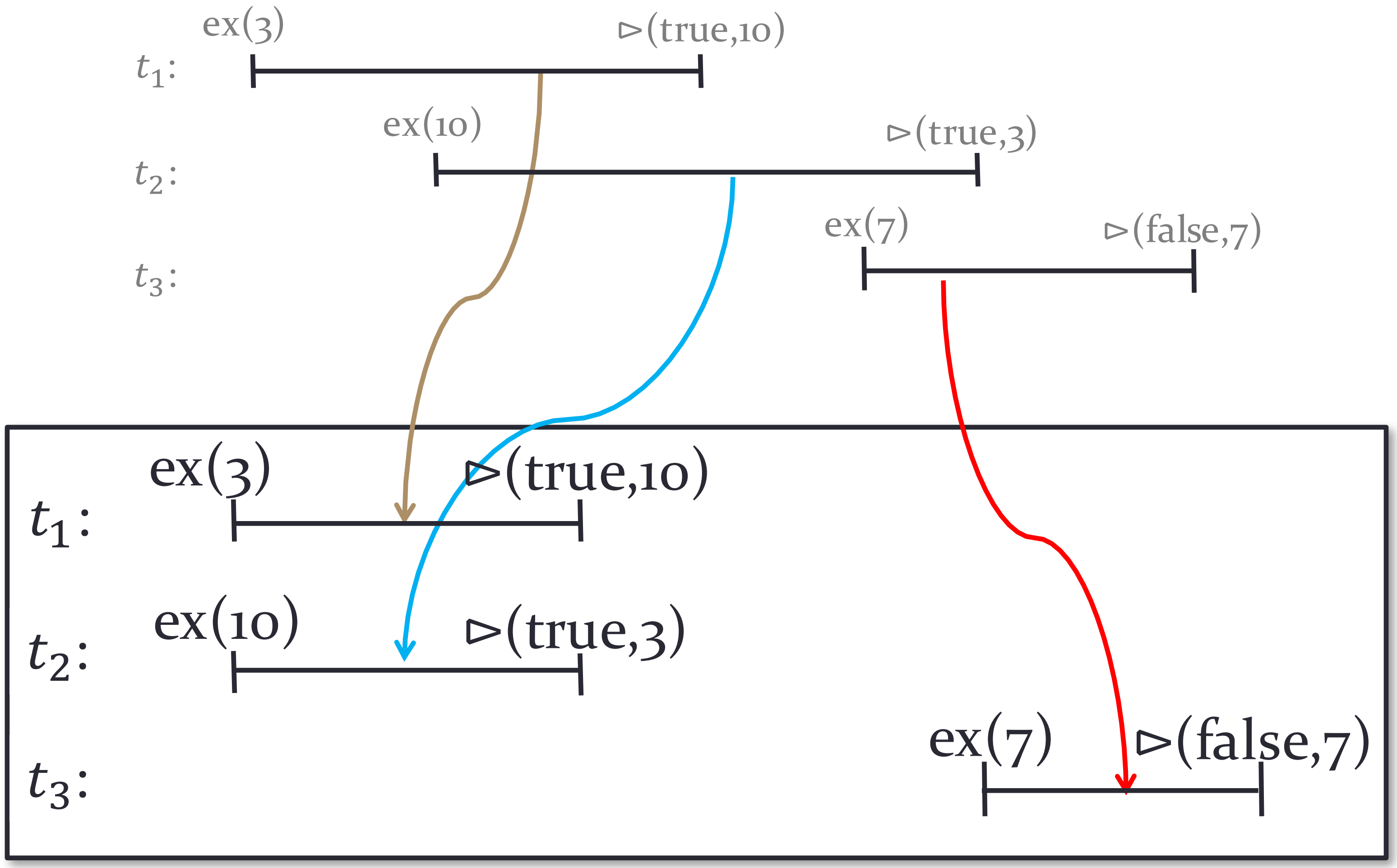
* *SPEC is prefix-closed if* $\forall H, H_1, H_2.\ H \in SPEC \land H = H_1 H_2 \Longrightarrow H_1 \in SPEC$

Concurrency-Aware Specifications



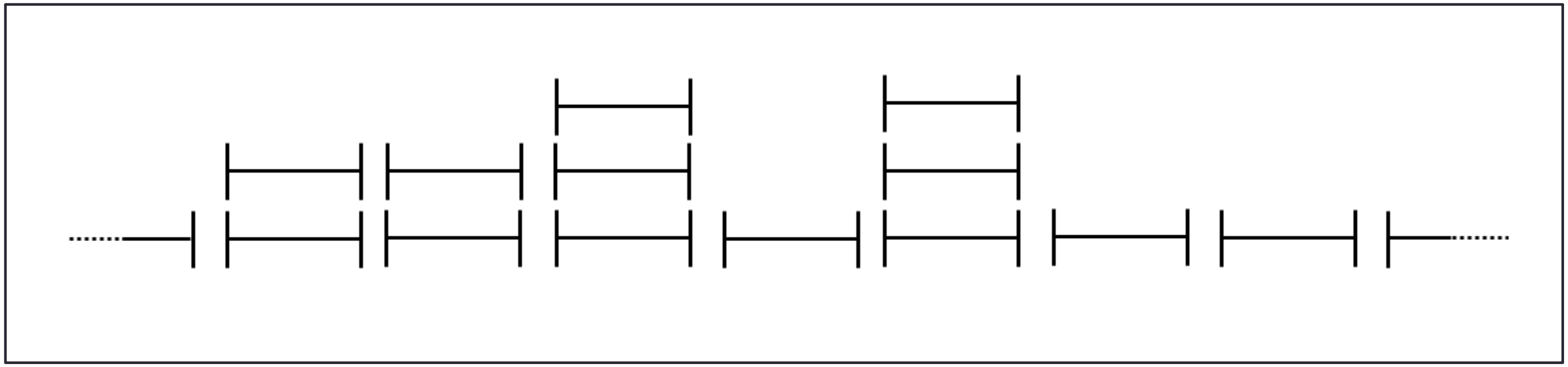
- **CA-specification**: a *prefix closed* set of *concurrency-aware histories*

Concurrency-Aware specification for Exchanger



Concurrency-Aware Linearizability

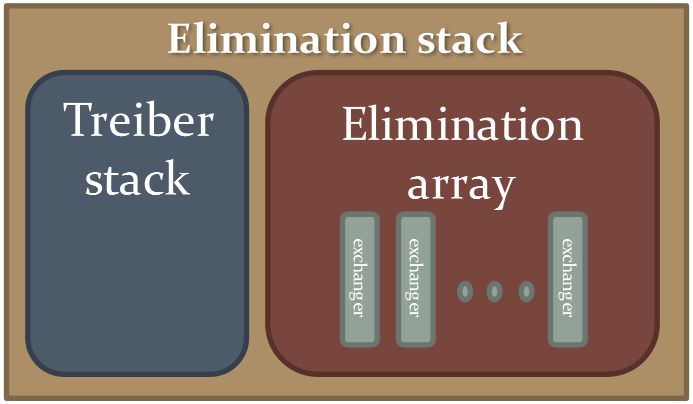
Generalizes linearizability by using CA-histories as the specification instead of sequential histories



Concurrency-Aware history

Goal: Modular Reasoning

- Linearizability allows for **compositional** reasoning
  - Reason about subcomponents in term of their interfaces

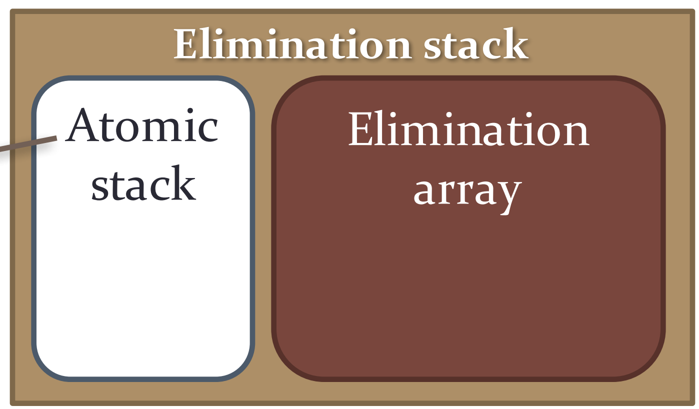


Goal: Modular Reasoning

- Linearizability allows for **compositional** reasoning
  - Reason about subcomponents in term of their interfaces

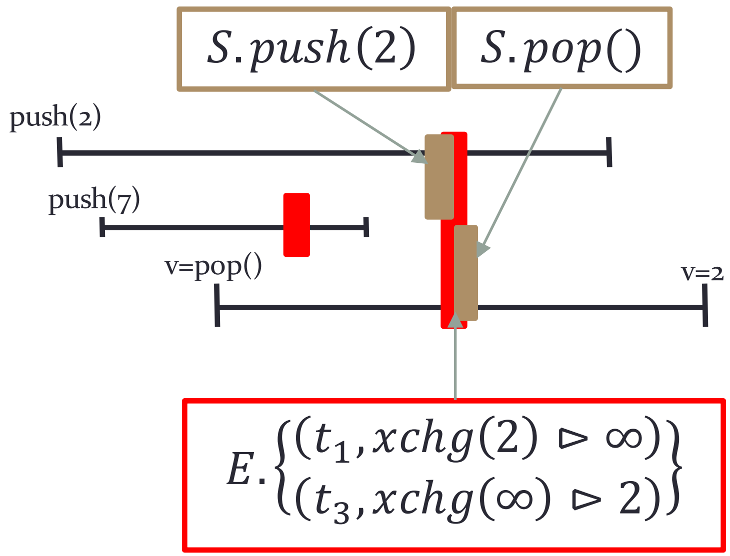
```
push(k) {
  atomic {
    ...
  }
}

pop() {
  atomic {
    ...
  }
}
```



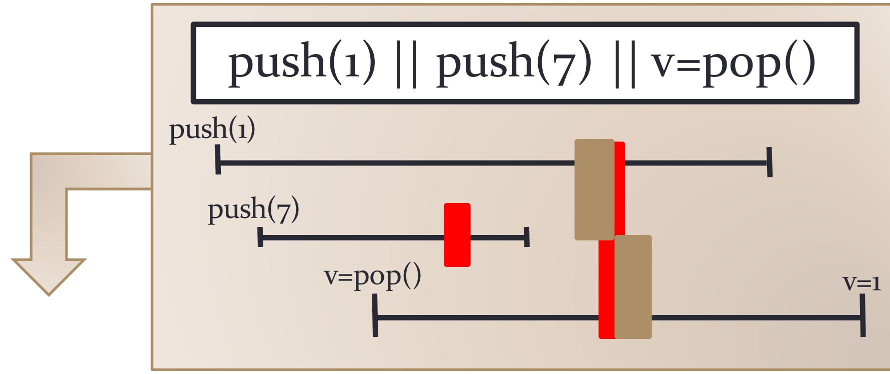
Goal: Modular Reasoning

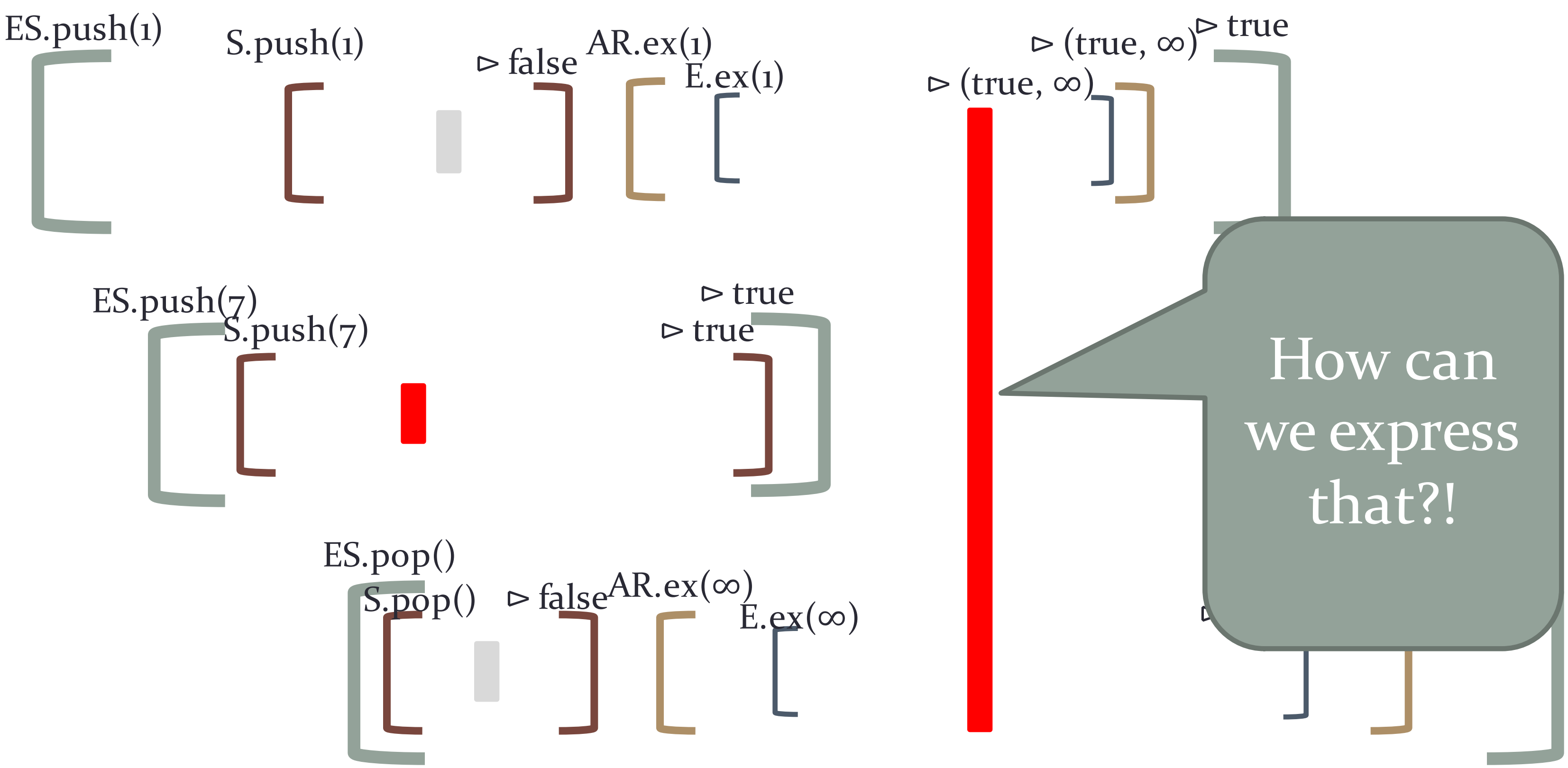
- **Compositional reasoning with CA-objects**
  - Reason about subcomponents in term of their interfaces
  - Reason about concurrency-aware subcomponent of “standard” linearizable objects



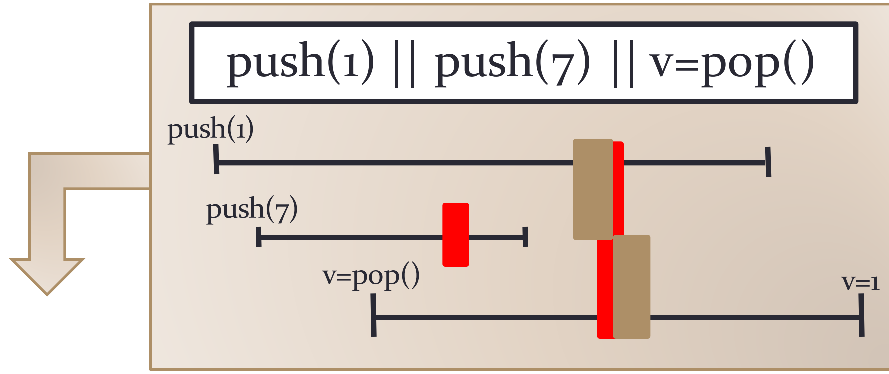
∞ is a dummy value used to indicate pop() operation

Challenge: Handling joint linearization points

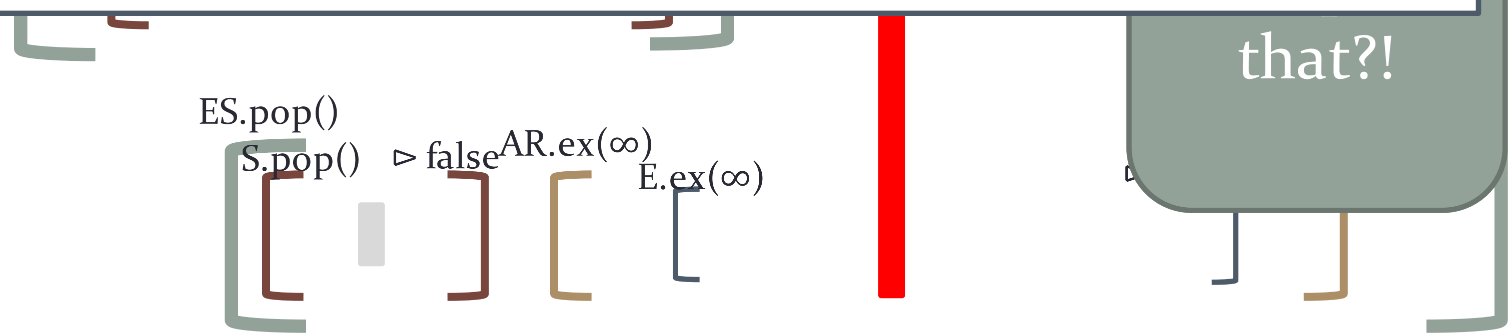




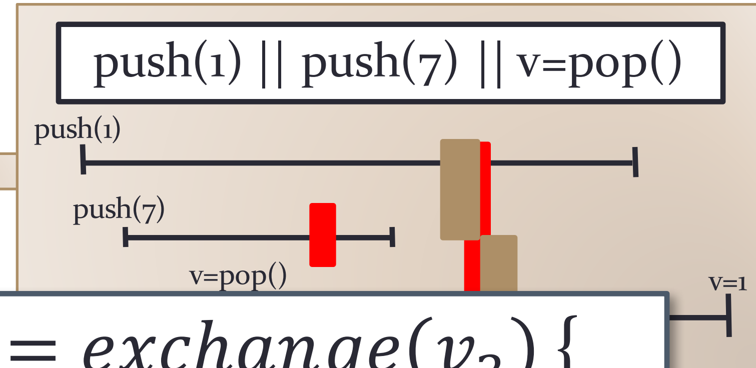
Challenge: Handling joint linearization points



$t_1 : r_1 = exchange(v_1)$ {
  atomic {
    ...
  }
}



Challenge: Handling joint linearization points



$t_1 : r_1 = exchange(v_1) \;||\; t_2 : r_2 = exchange(v_2)$ {

atomic {

…

$r_1 = (\text{true}, v_2);$

…

$r_2 = (\text{true}, v_1);$

…

}

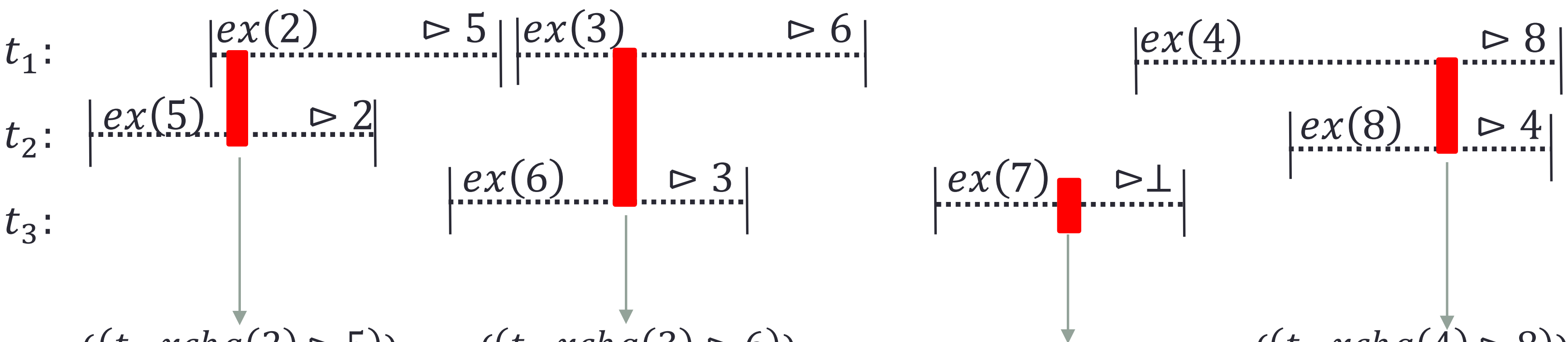
}

Our solution

- Auxiliary variable $\mathcal{T}$: logs the sequence of operations
- Adaptation function F_o: adapt operations on subcomponents of object o to their affect on o

Our solution

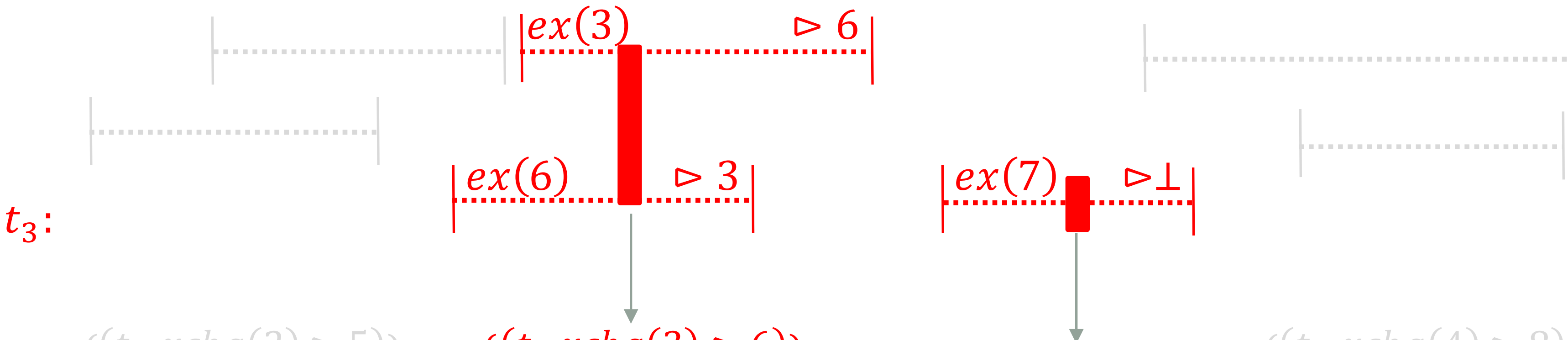
- Record interaction using a "history" auxiliary variable $\mathcal{T}$



$$\mathcal{T} = \quad E.\begin{Bmatrix}(t_1, xchg(2) \rhd 5) \\ (t_2, xchg(5) \rhd 2)\end{Bmatrix} \; E.\begin{Bmatrix}(t_1, xchg(3) \rhd 6) \\ (t_3, xchg(6) \rhd 3)\end{Bmatrix} \; E.\{(t_3, xchg(7) \rhd \bot)\} \; E.\begin{Bmatrix}(t_1, xchg(4) \rhd 8) \\ (t_2, xchg(8) \rhd 4)\end{Bmatrix}$$

Our solution

- Record interaction using a "history" auxiliary variable $\mathcal{T}$

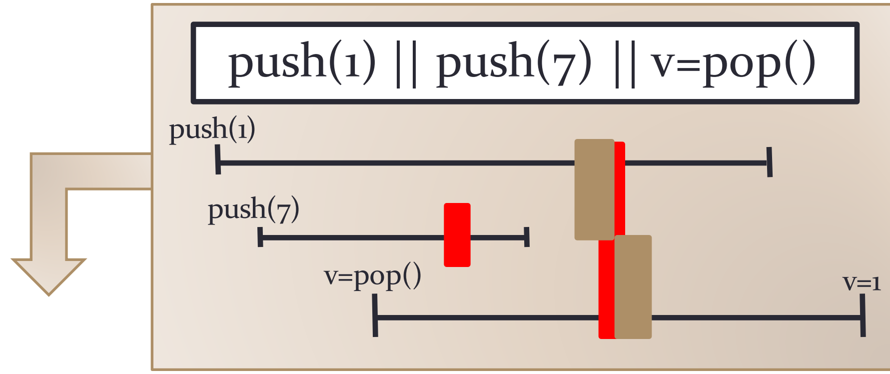


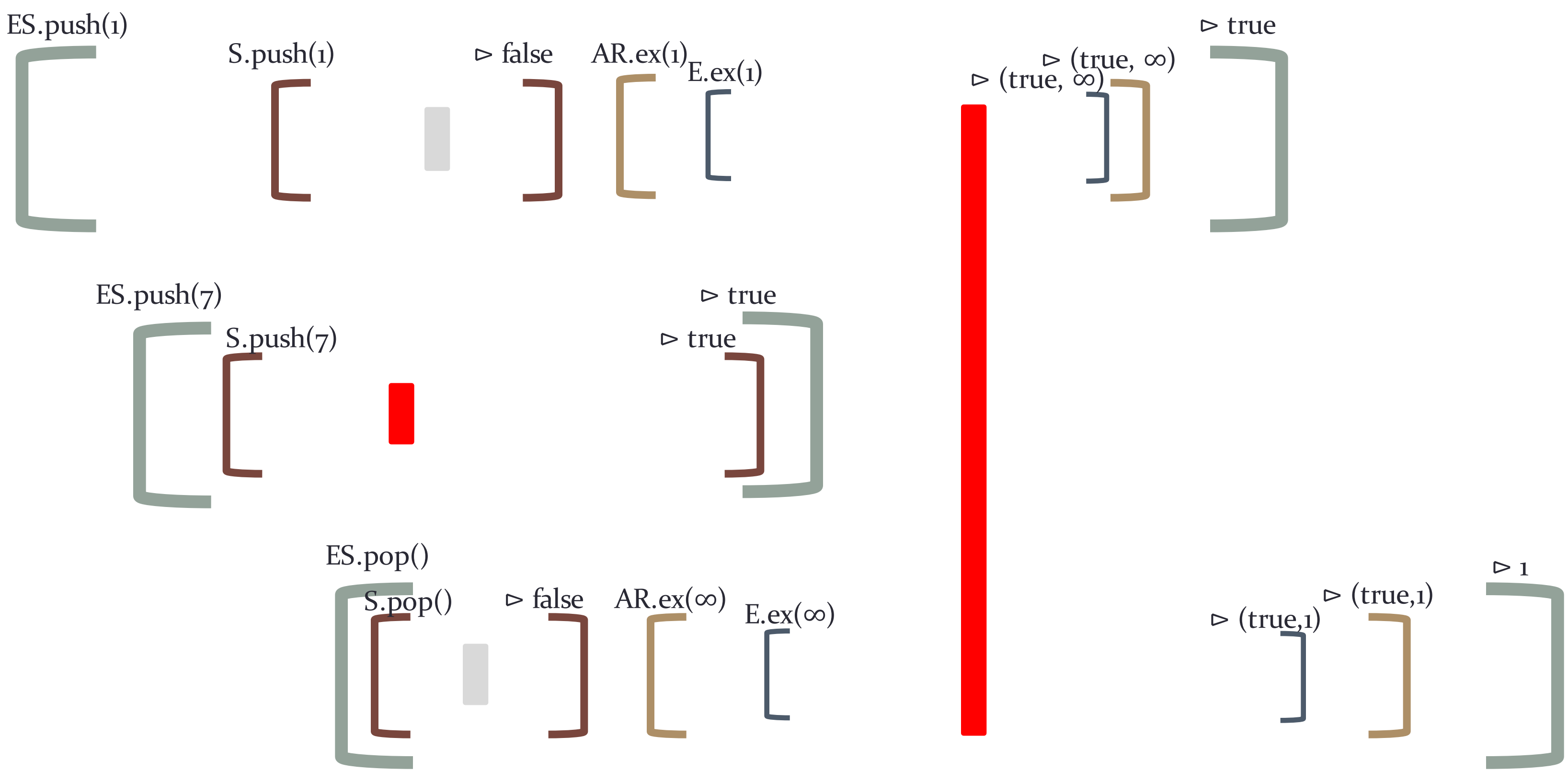
$$\mathcal{T} = \quad E.\begin{Bmatrix}(t_1, xchg(2) \rhd 5) \\ (t_2, xchg(5) \rhd 2)\end{Bmatrix} \quad E.\begin{Bmatrix}(t_1, xchg(3) \rhd 6) \\ (t_3, xchg(6) \rhd 3)\end{Bmatrix} \quad E.\{(t_3, xchg(7) \rhd \bot)\} \quad E.\begin{Bmatrix}(t_1, xchg(4) \rhd 8) \\ (t_2, xchg(8) \rhd 4)\end{Bmatrix}$$

$$\{\mathcal{T}_{\text{tid}} = T\}\ \text{tid}: r = xchg(v)\ \{\exists t', v'. \left(r = (true, v') \wedge \mathcal{T}_{\text{tid}} = T \cdot E.\begin{Bmatrix}(\text{tid}, xchg(\text{v}) \rhd v') \\ (t', xchg(\text{v}') \rhd v)\end{Bmatrix} \wedge t' \neq \text{tid} \right)$$
$$\vee\ (r = (false, v) \wedge \mathcal{T}_{\text{tid}} = T \cdot E.\{(\text{tid}, xchg(v) \rhd v)\})\}$$

Our solution

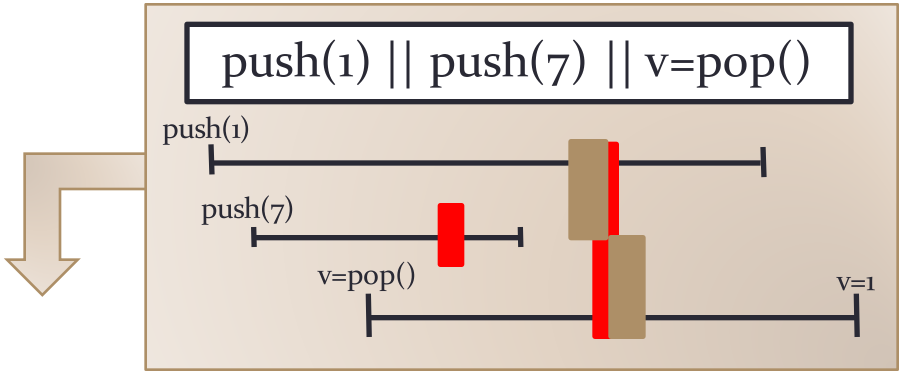
- **Adaptation function F_0**

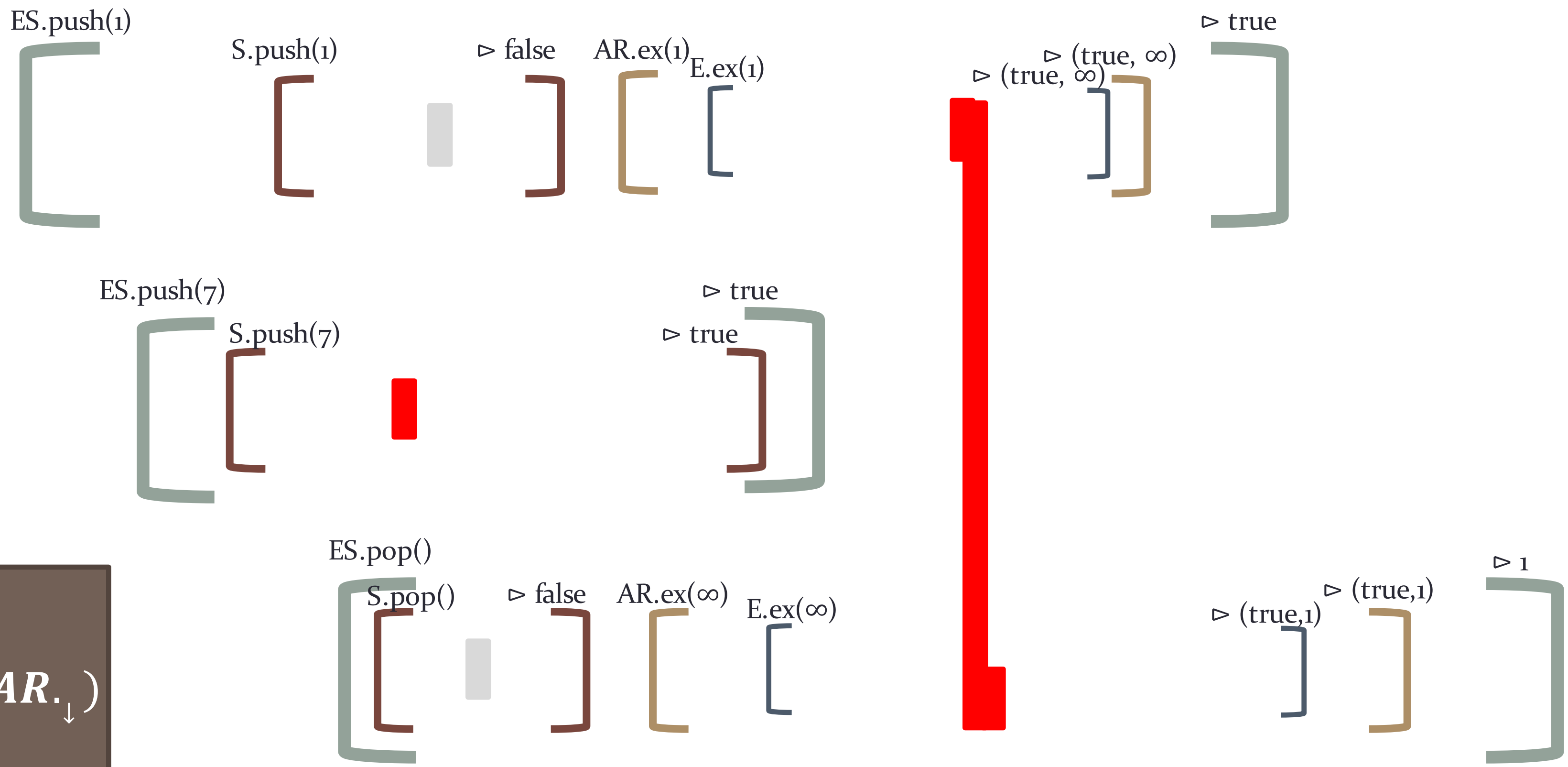


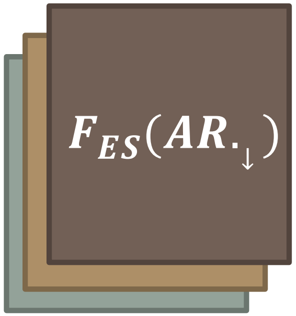


Our solution

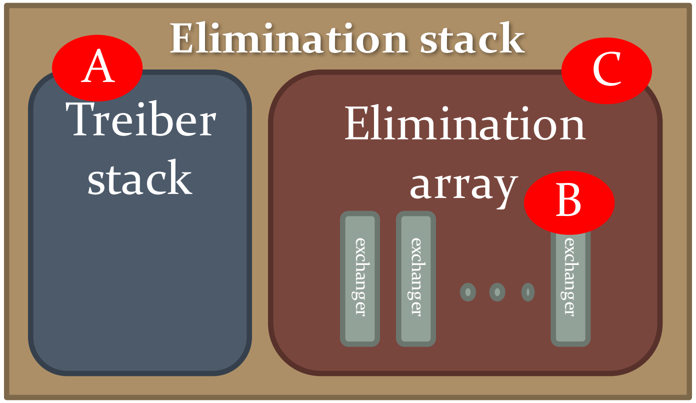
- **Adaptation function F_0**







Our solution



- **Adaptation function F_O**

- Each component defines how subcomponents adapt:
  - Elimination stack to Treiber stack (A):

$$F_{ES}(S.\{t, push(n) \rhd true\}) \triangleq (ES.\{t, push(n) \rhd true\})$$
$$F_{ES}(S.\{t, pop() \rhd (true, n)\}) \triangleq (ES.\{t, pop() \rhd n\})$$

  - Elimination array to Exchanger (B):

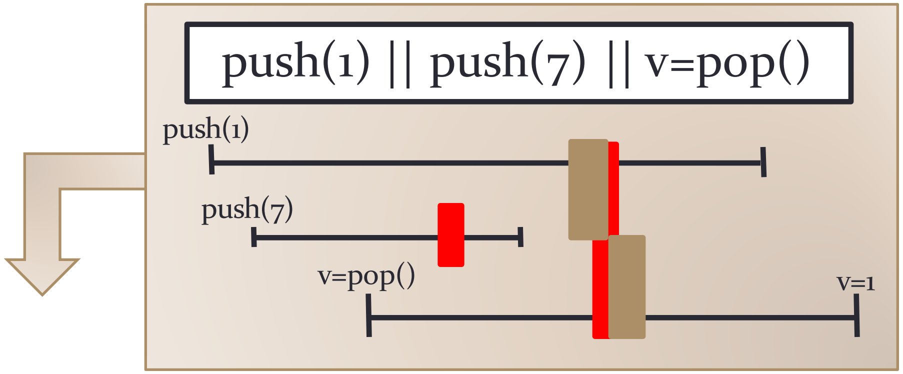
$$F_{AR}(E[i].\mathbb{S}) \triangleq (AR.\mathbb{S})$$

  - Elimination stack to Elimination array (C):

$$F_{ES}\left(AR.\begin{Bmatrix} t, ex(n) \rhd (true, \infty) \\ t', ex(\infty) \rhd (true, n) \end{Bmatrix}\right) \qquad (n \neq \infty)$$
$$\triangleq (ES.\{t, push(n) \rhd true\}) \cdot (ES.\{t', pop() \rhd (true, n)\})$$

Our solution

- **Adaptation function F_O**



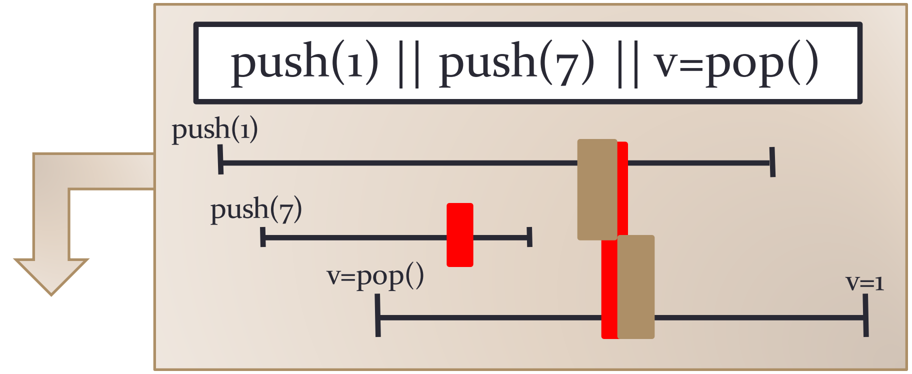
$$\underbrace{ES.\{(t_2, push(7) \rhd \mathrm{T})\}}_{F_{ES}(\mathcal{T}|_S)} \; \underbrace{ES.\{(t_1, push(2) \rhd \mathrm{T})\} \cdot ES.\{(t_3, pop() \rhd 2)\}}_{F_{ES}(\mathcal{T}|_{AR})}$$

$$\underbrace{S.\{(t_2, push(7) \rhd \mathrm{T})\} \; S.\{(t_1, push(2) \rhd \mathrm{F})\} \; S.\{(t_3, pop() \rhd \mathrm{F})\}}_{\mathcal{T}|_S} \quad \underbrace{AR.\begin{Bmatrix}(t_1, xchg(2) \rhd \infty)\\(t_3, xchg(\infty) \rhd 2)\end{Bmatrix}}_{\mathcal{T}|_{AR} = F_{AR}(\mathcal{T}|_E)}$$

$$\underbrace{E.\begin{Bmatrix}(t_1, xchg(2) \rhd \infty)\\(t_3, xchg(\infty) \rhd 2)\end{Bmatrix}}_{\mathcal{T}|_E}$$

Our solution

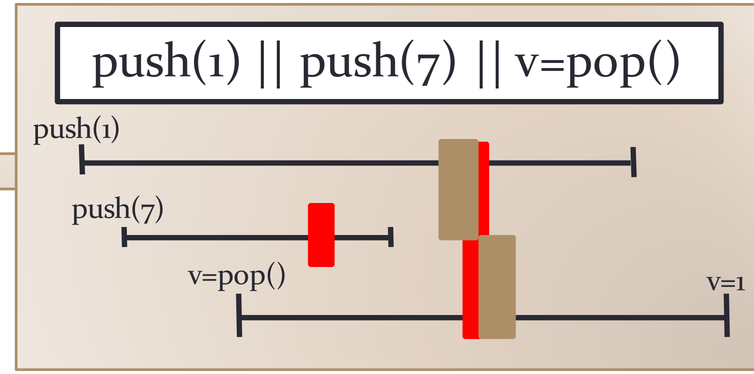
Object-local views of the trace



$$\underbrace{S.\{(t_2, push(7) \rhd \mathrm{T})\}\ S.\{(t_1, push(2) \rhd \mathrm{F})\}\ S.\{(t_3, pop() \rhd \mathrm{F})\}}_{\mathcal{T}_S} \quad \underbrace{E.\left\{\begin{matrix}(t_1, xchg(2) \rhd \infty) \\ (t_3, xchg(\infty) \rhd 2)\end{matrix}\right\}}_{\mathcal{T}_E}$$

Our solution

Object-local views of the trace



$F_{AR}(E._{\downarrow})$

$$AR.\left\{\begin{matrix}(t_1, xchg(2) \rhd \infty)\\(t_3, xchg(\infty) \rhd 2)\end{matrix}\right\}$$

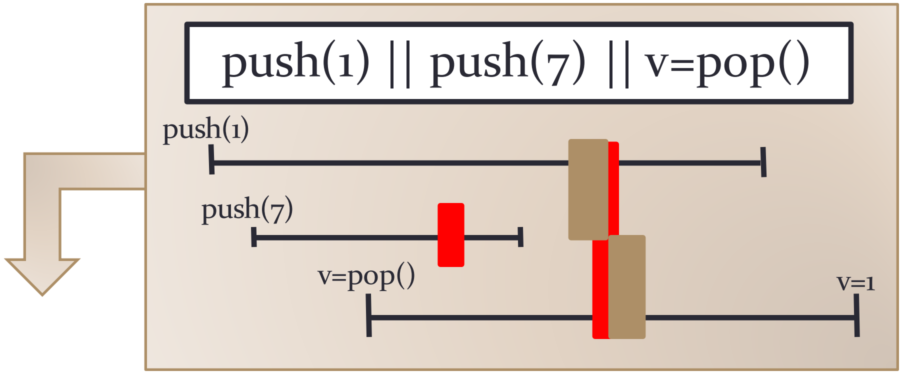
$$\mathcal{T}_{AR} = F_{AR}(\mathcal{T}_E)$$

$$\underbrace{S.\{(t_2, push(7) \rhd \mathrm{T})\}\ S.\{(t_1, push(2) \rhd \mathrm{F})\}\ S.\{(t_3, pop() \rhd \mathrm{F})\}}_{\mathcal{T}_S}$$

$$\underbrace{E.\left\{\begin{matrix}(t_1, xchg(2) \rhd \infty)\\(t_3, xchg(\infty) \rhd 2)\end{matrix}\right\}}_{\mathcal{T}_E}$$

Our solution

Object-local views of the trace



$$ES.\{(t_2, push(7) \rhd \mathrm{T})\}\ ES.\{(t_1, push(2) \rhd \mathrm{T})\} \cdot ES.\{(t_3, pop() \rhd 2)\}$$

$$\mathcal{T}_{ES} = F_{ES}(\mathcal{T}_S) \quad F_{ES}(\mathcal{T}_{AR})$$

$$F_{ES}(S._{\downarrow}) \quad F_{ES}(AR._{\downarrow})$$

$$AR.\begin{Bmatrix}(t_1, xchg(2) \rhd \infty)\\(t_3, xchg(\infty) \rhd 2)\end{Bmatrix}$$

$$\mathcal{T}_{AR} = F_{AR}(\mathcal{T}_E)$$

$$S.\{(t_2, push(7) \rhd \mathrm{T})\}\ S.\{(t_1, push(2) \rhd \mathrm{F})\}\ S.\{(t_3, pop() \rhd \mathrm{F})\}$$

$$\mathcal{T}_S$$

$$E.\begin{Bmatrix}(t_1, xchg(2) \rhd \infty)\\(t_3, xchg(\infty) \rhd 2)\end{Bmatrix}$$

$$\mathcal{T}_E$$

Modular reasoning

```
class ElimArray {
  Exchanger[] E = new Exchanger[K];
  (bool, int) exchange(int data) {
    int slot = random(0,K-1);
    return E[slot].exchange(data);
} }
class Stack {
  class Cell {Cell next; int data;}
  Cell top = null;
  bool push(int data){
    Cell h = top;
    Cell n = new Cell(data, h);
    return CAS(&top, h, n);
  }
  (bool, int) pop(){
    Cell h = top;
    if (h == null)
      return (false, 0); // EMPTY
    Cell n = h.next;
    if (CAS(&top, h, n))
      return (true, h.data);
    else
      return (false, 0);
} }
class EliminationStack {
  final int POP_SENTINAL = INFINITY;
  Stack S = new Stack();
  ElimArray AR = new ElimArray();
```

```
  {WF_ES(T_ES) ∧ T_ES|_t = T ∧ v < ∞}
  bool push(int v) { int d;
    while(true){
      {WF_ES(T_ES) ∧ T_ES|_t = T ∧ v < ∞}
      bool b = S.push(v);
      {WF_ES(T_ES) ∧ T_ES|_t = T·F_ES(S.(tid,push(v) ▷ b)) ∧ v < ∞}
      if (b) return true;
      {WF_ES(T_ES) ∧ T_ES|_t = T ∧ v < ∞}
      (b,d) = AR.exchange(v);
      {b ∧ d = ∞ ∧ T_ES|_t = T·(ES.(t,push(v) ▷ true)) ∧ WF_ES(T_ES)
       ∨ d ≠ ∞ ∧ WF_ES(T_ES) ∧ T_ES|_t = T ∧ v < ∞}
      if (d == POP_SENTINAL) return true;
  } }
  {WF_ES(T_ES) ∧ T_ES|_t = T·(ES.(t,push(v) ▷ ret))}

  {WF_ES(T_ES) ∧ T_ES|_t = H}
  (bool,int) pop() { (bool, int) (b,v);
    while(true){
      (b,v) = S.pop();
      {WF_ES(T_ES) ∧ T_ES|_t = T·F_ES(S.(t,pop() ▷ b,v))}
      if (b) return (true,v);
      {WF_ES(T_ES) ∧ T_ES|_t = T}
      (b,v) = AR.exchange(POP_SENTINAL);
      {b ∧ v ≠ ∞ ∧ T_ES|_t = T·(ES.(t,pop() ▷ true,v)) ∧ WF_ES(T_ES)
       ∨ v = ∞ ∧ WF_ES(T_ES) ∧ T_ES|_t = T}
      if (v != POP_SENTINAL) return (true,v);
  } }
  {WF_ES(T_ES) ∧ T_ES|_t = T·(ES.(t,pop() ▷ ret))}
}
```

```
{T_E|_tid = T}
(bool,int) exchange(int v) {
  Offer n = new Offer(tid,v);
  {A}
  if (CAS(g, null, n)){ // INIT
    {(T_E|_tid = T ∧ n ↦ tid,v,null ∧ g = n) ∨ B(n.hole)}
    sleep(50);
    if (CAS(n.hole, null, fail)) // PASS
      {T_E|_tid = T}
      return (false,v); // FAIL
    else {B(n.hole)}
      return (true,n.hole.data);
  }
  {A}
  Offer cur = g;
  {A ∧ (g = cur ∨ cur.hole ≠ null)}
  if (cur != null) {
    {A ∧ (g = cur ∨ cur.hole ≠ null) ∧ cur ≠ null ∧ ¬s}
    bool s = CAS(cur.hole, null, n); // XCHG
    {(¬s ∧ A ∨ s ∧ B(cur)) ∧ cur ≠ null ∧ cur.hole ≠ null}
    CAS(g, cur, null); // CLEAN
    if (s) {B(cur)}
      return (true,cur.data);
  }
  return (false,v); // FAIL
}
{(∃t′,v′. ret = (true,v′) ∧ t′ ≠ tid ∧
     T_E|_tid = T·(E.{(tid,ex(v) ▷ true,v′),(t′,ex(v′) ▷ true,v)}))
 ∨ (ret = (false,v) ∧ T_E|_tid = T·(E.{(tid,ex(v) ▷ false,v)}))}
}
```

Modular reasoning

$R, G \ \vdash$

$\{\mathrm{WF}_{\mathrm{ES}}(\mathcal{T}_{\mathrm{ES}}) \wedge \mathcal{T}_{\mathrm{ES}}|_t = T \wedge v < \infty\}$

```
bool push(int v) { int d;
  while(true){
```

$\{\mathrm{WF}_{\mathrm{ES}}(\mathcal{T}_{\mathrm{ES}}) \wedge \mathcal{T}_{\mathrm{ES}}|_t = T \wedge v < \infty\}$

```
    bool b = S.push(v);
```

$\{\mathrm{WF}_{\mathrm{ES}}(\mathcal{T}_{\mathrm{ES}}) \wedge \mathcal{T}_{\mathrm{ES}}|_t = T \cdot F_{\mathrm{ES}}(\mathrm{S}.(\mathtt{tid}, \mathtt{push}(v) \triangleright b)) \wedge v < \infty\}$

```
    if (b) return true;
```

$\{\mathrm{WF}_{\mathrm{ES}}(\mathcal{T}_{\mathrm{ES}}) \wedge \boxed{\mathcal{T}_{\mathrm{ES}}|_t = T} \wedge v < \infty\}$

```
    (b,d) = AR.exchange(v);
```

$$\left\{\begin{array}{l}\boxed{b \wedge d = \infty \wedge \mathcal{T}_{\mathrm{ES}}|_t = T \cdot (\mathrm{ES}.(t, \mathtt{push}(v) \triangleright \mathsf{true}))} \wedge \mathrm{WF}_{\mathrm{ES}}(\mathcal{T}_{\mathrm{ES}}) \\ \vee\, d \neq \infty \wedge \mathrm{WF}_{\mathrm{ES}}(\mathcal{T}_{\mathrm{ES}}) \wedge \mathcal{T}_{\mathrm{ES}}|_t = T \wedge v < \infty\end{array}\right\}$$

```
    if (d == POP_SENTINAL) return true;
} }
```

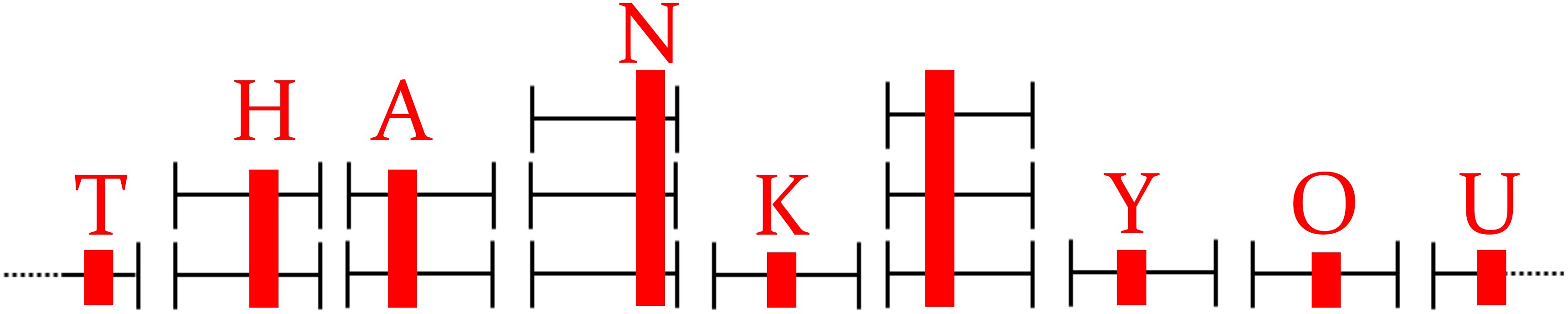
$\{\mathrm{WF}_{\mathrm{ES}}(\mathcal{T}_{\mathrm{ES}}) \wedge \mathcal{T}_{\mathrm{ES}}|_t = T \cdot (\mathrm{ES}.(t, \mathtt{push}(v) \triangleright ret))\}$

Related work

- Verifying the elimination stack
  - [Hendler et al. SPAA '04]
  - [Scherer & Scott, SCOOL '05]
  - [Vafeiadis, PHD thesis]
  - …
- Concurrency-aware linearizabilty
  - Set-Linearizabilty [Neiger, PODC '94]

Summary

- We identify the class of concurrency-aware objects
  - CAL (Concurrency-Aware Linearizability)
  - Syntactic specifications
- Verification method
  - Thread-modular
  - Compositional
  - Translate seemingly instantaneous operations into a sequence of indivisible actions of the clients
- First modular proof of linearizability of the elimination stack


T
H
A
N
K
Y
O
U