Le Temps des Cerises: Efficient Temporal Stack Safety on Capability Machines using Directed Capabilities

Joint work with Alix Trieu and Lars Birkedal

Aïna Linn Georges - Maj 2022

What is Stack Safety?

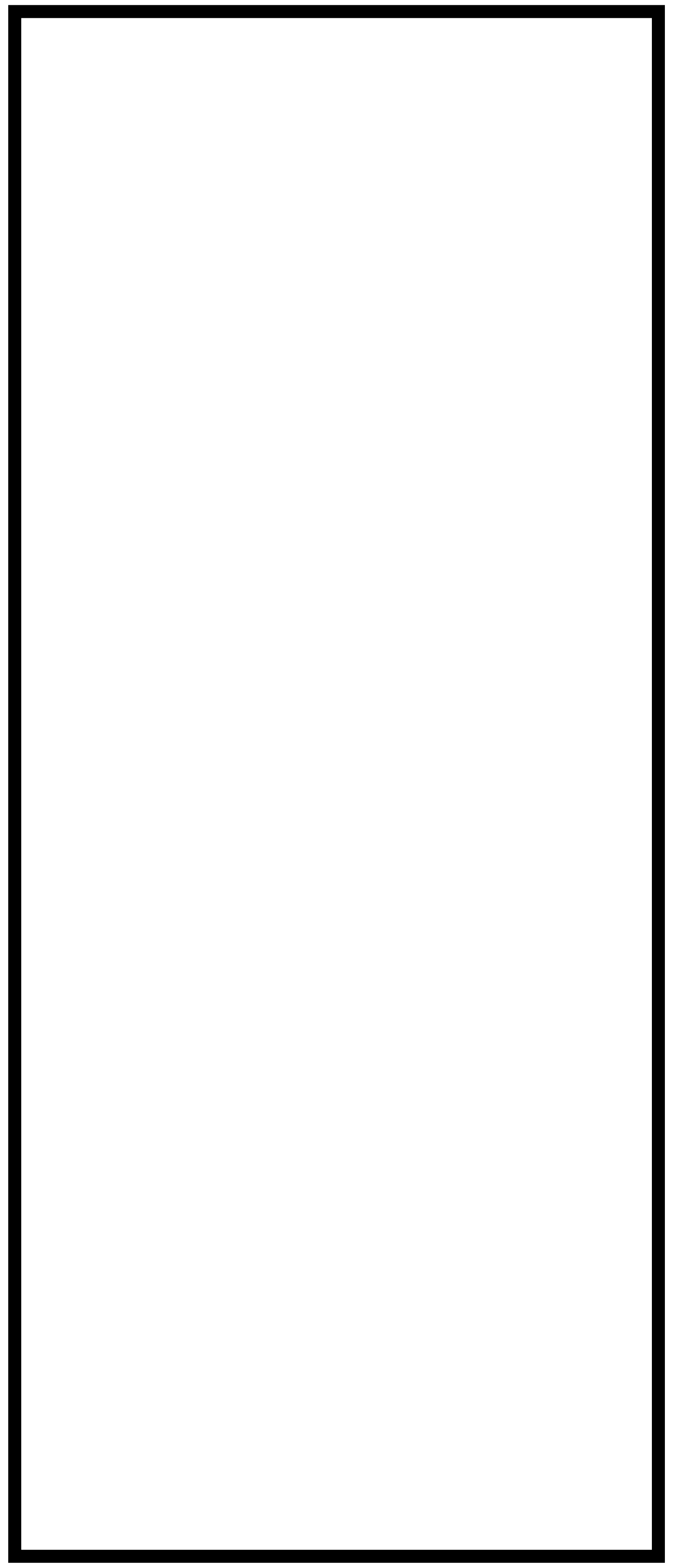
- The call stack is responsible for:
  - Local variables
  - Adheres to strict scope and lifetime rules
  - Enforces that calls are well bracketed
- A family of stack safety properties

A Family of Stack Safety Properties

- Local state encapsulation
  - Local variables/stack objects — **stack allocated**
  - Local state of a closure — **heap allocated**
- Well-bracketed control flow
- Temporal stack safety
  - No use after free of stack allocated data

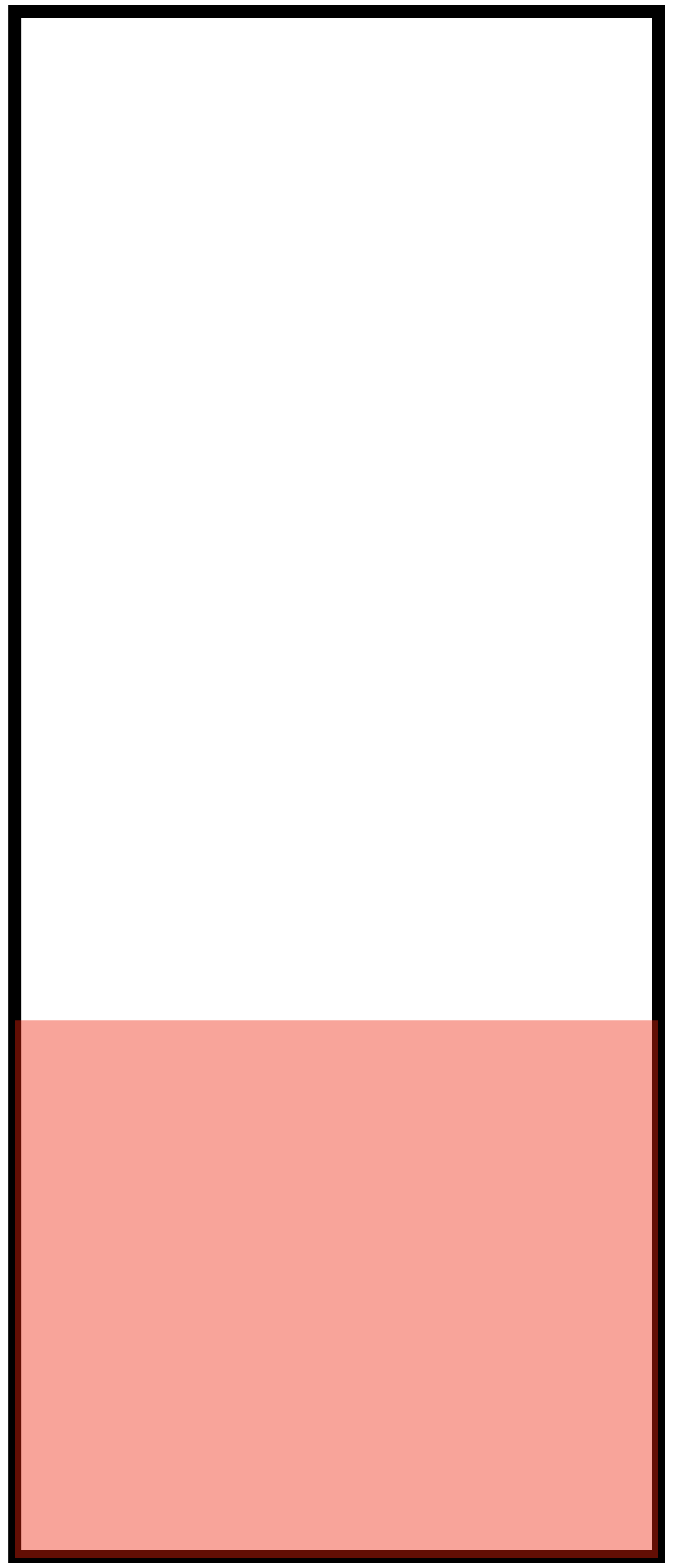
Stack Safety Properties

As stack frame discipline



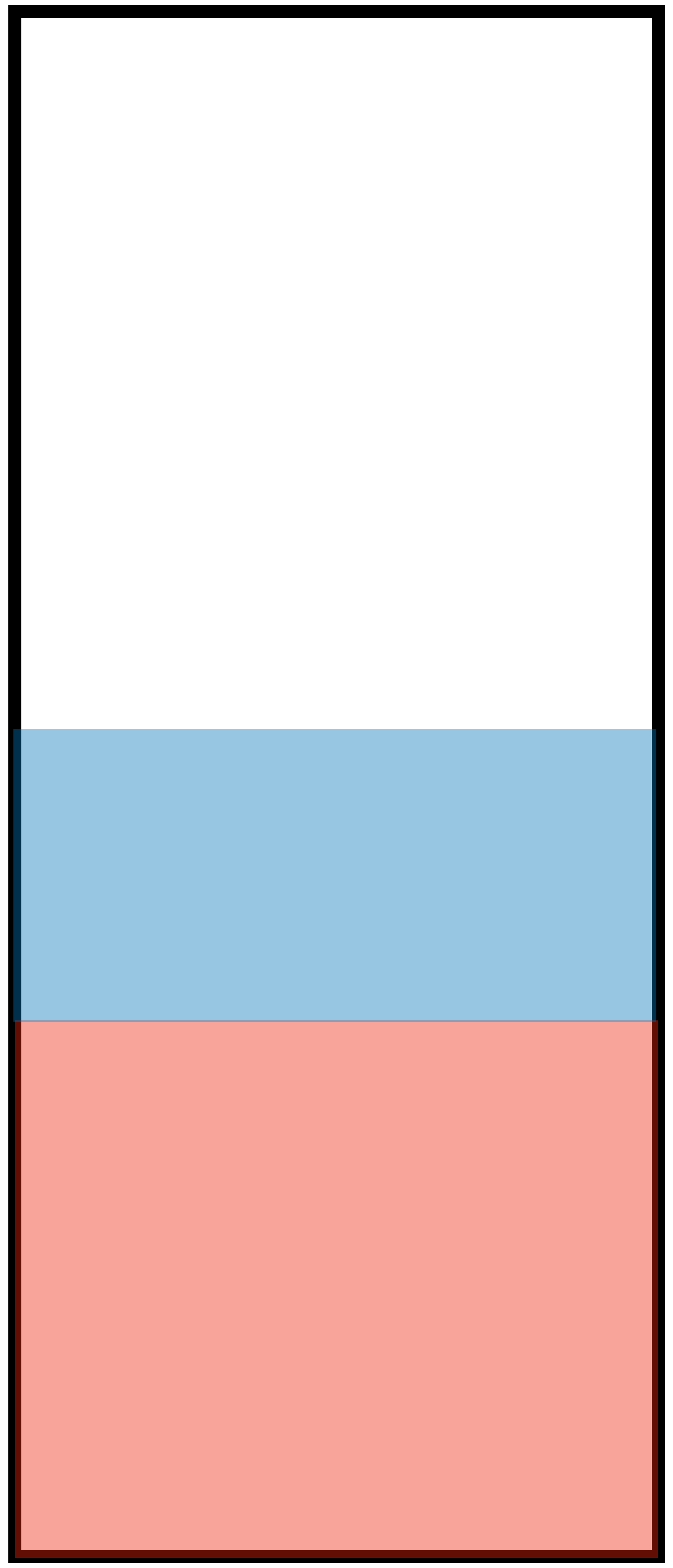
Stack Safety Properties

As stack frame discipline



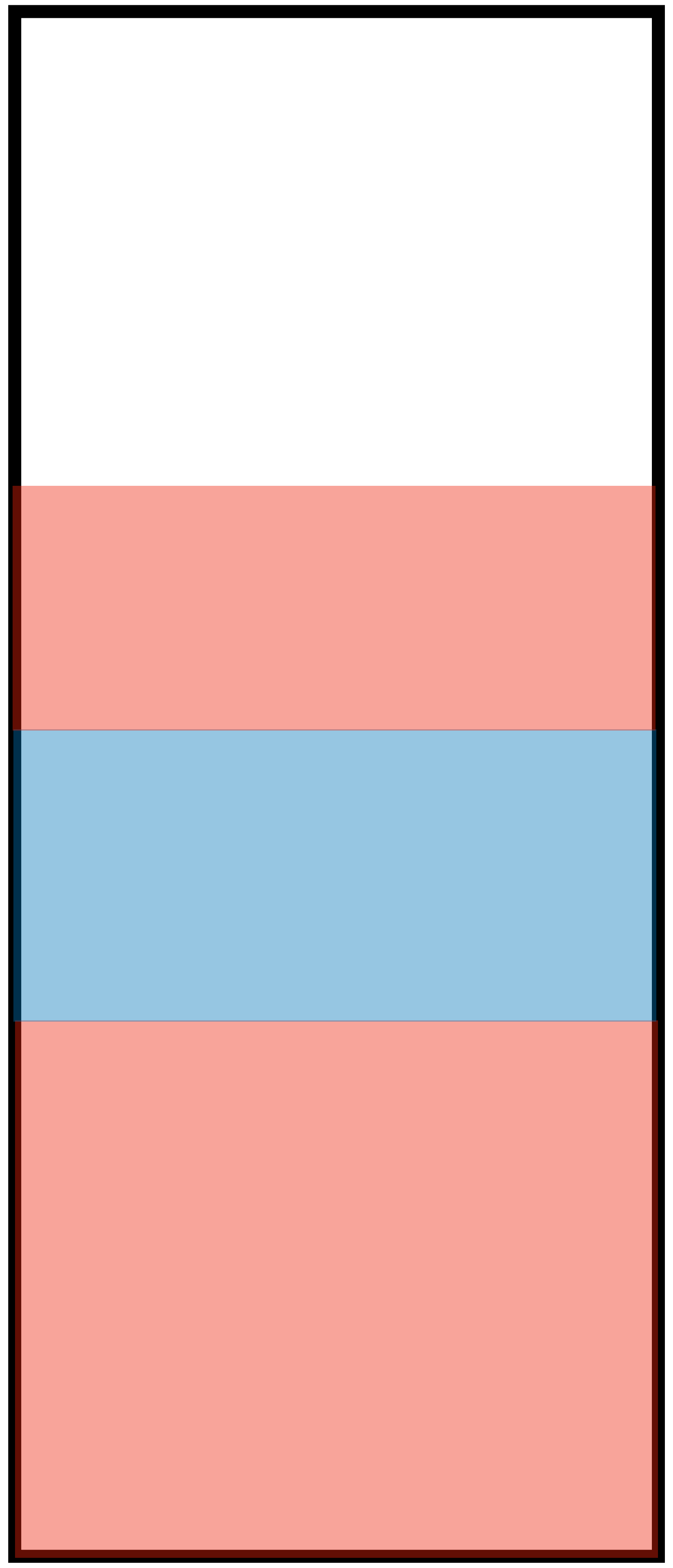
Stack Safety Properties

As stack frame discipline



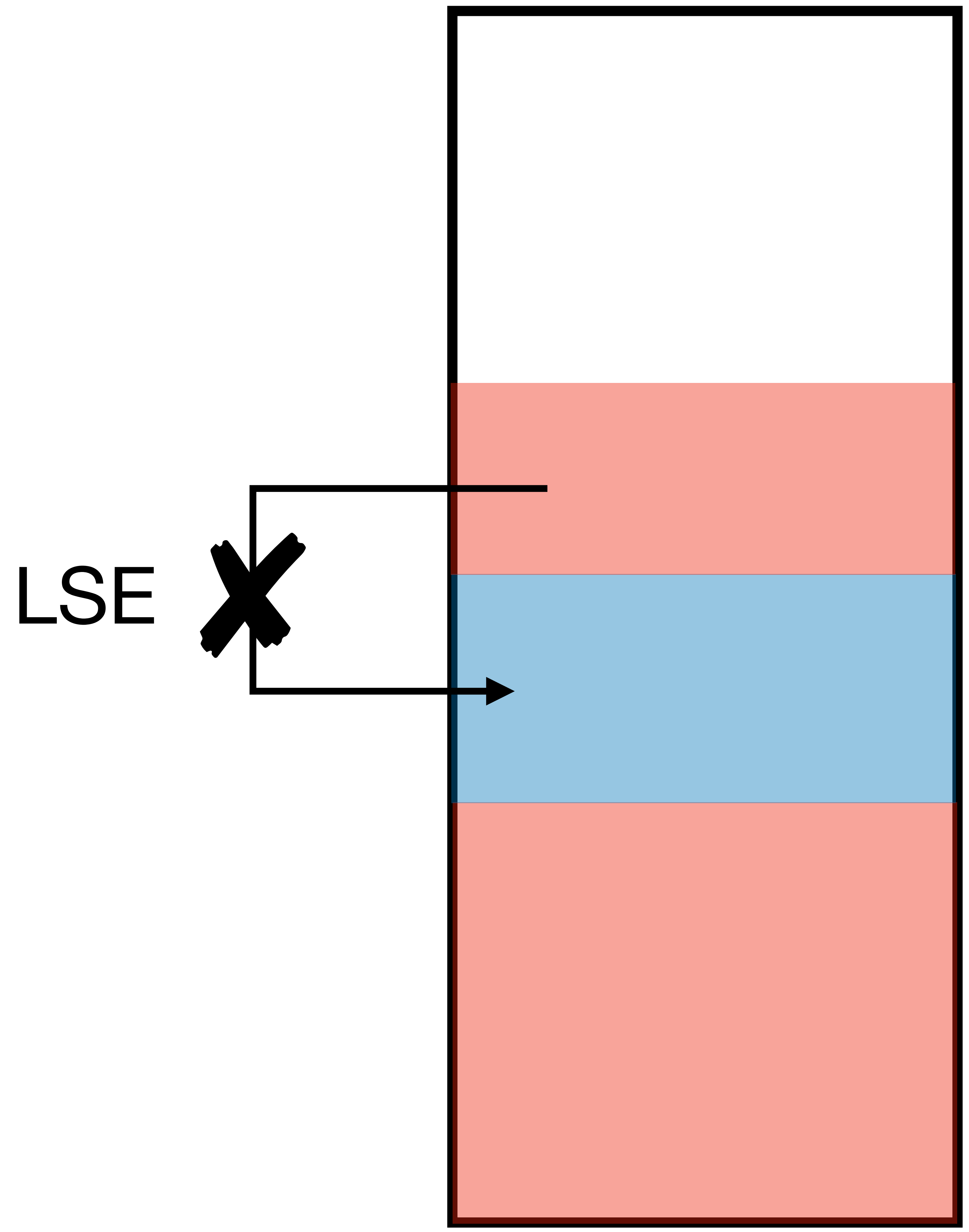
Stack Safety Properties

As stack frame discipline



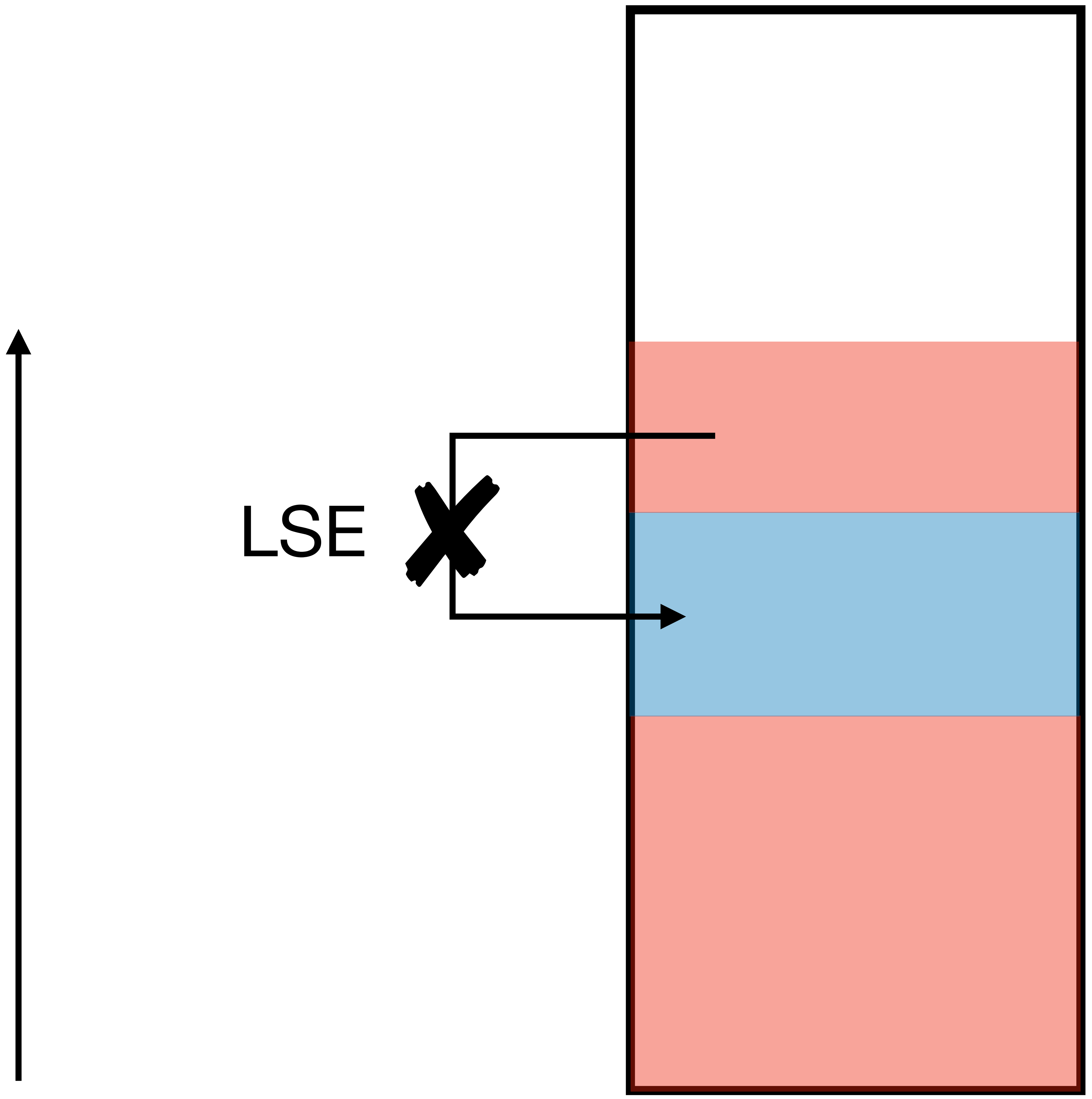
Stack Safety Properties

As stack frame discipline



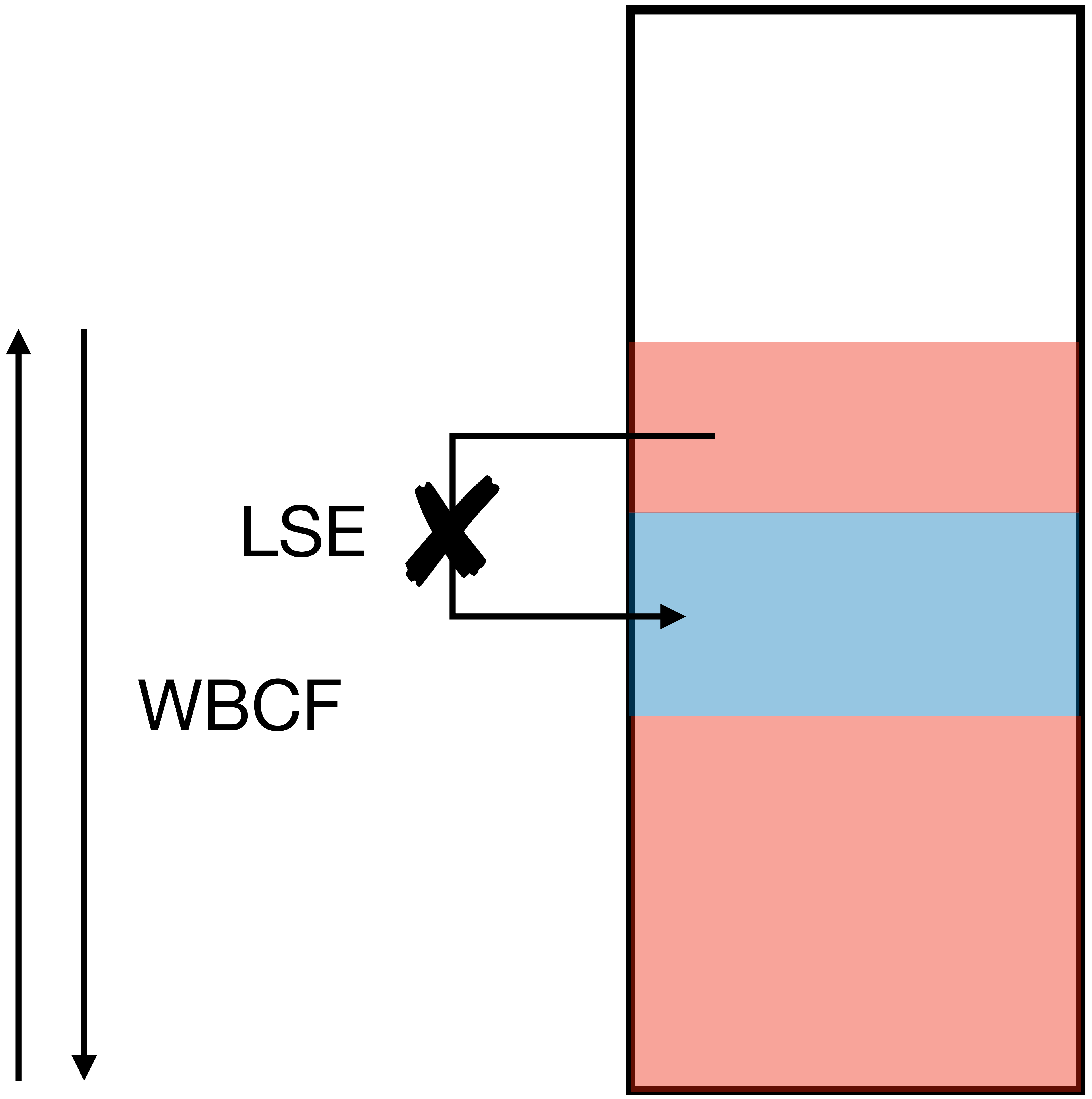
Stack Safety Properties

As stack frame discipline



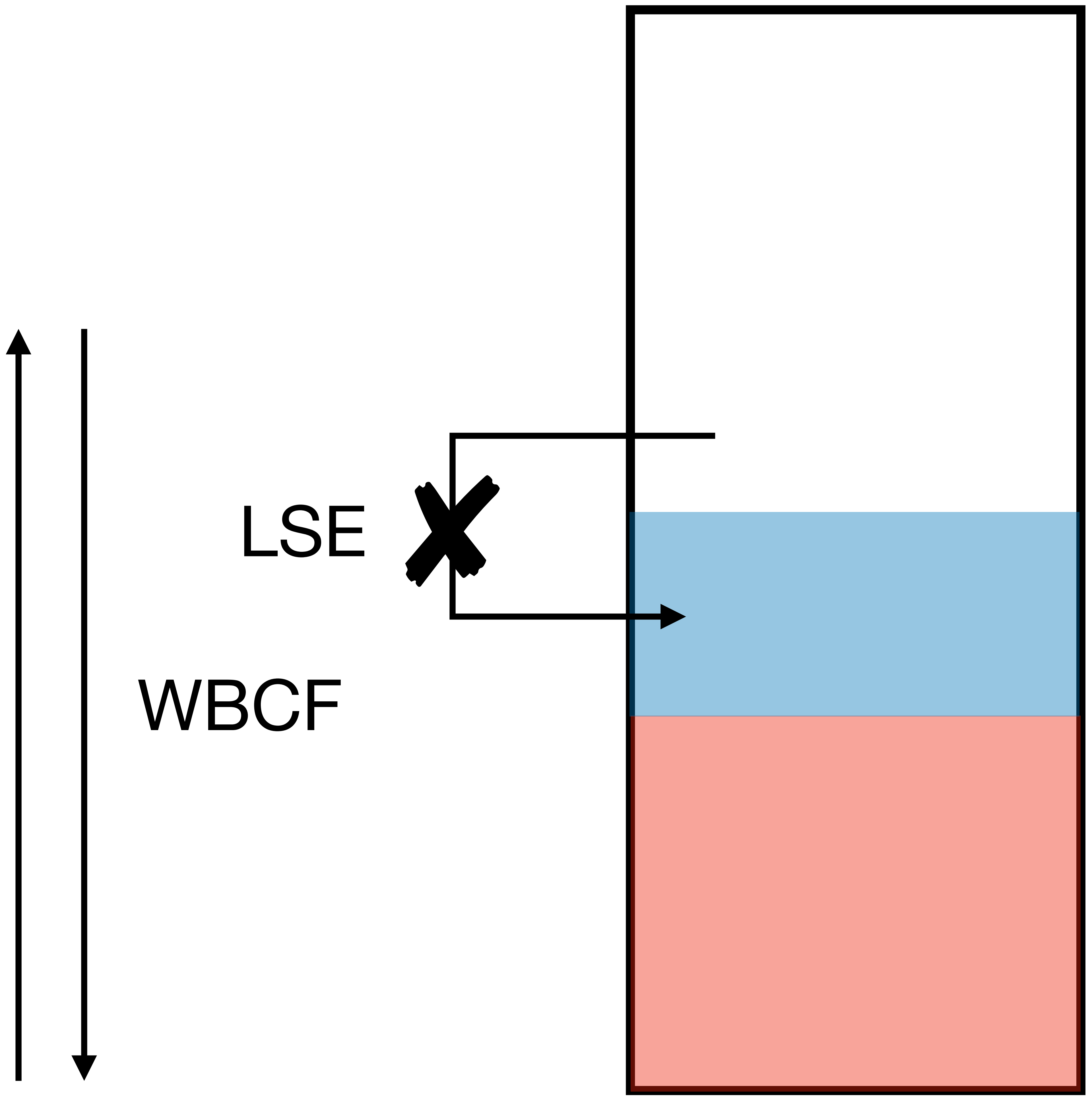
Stack Safety Properties

As stack frame discipline



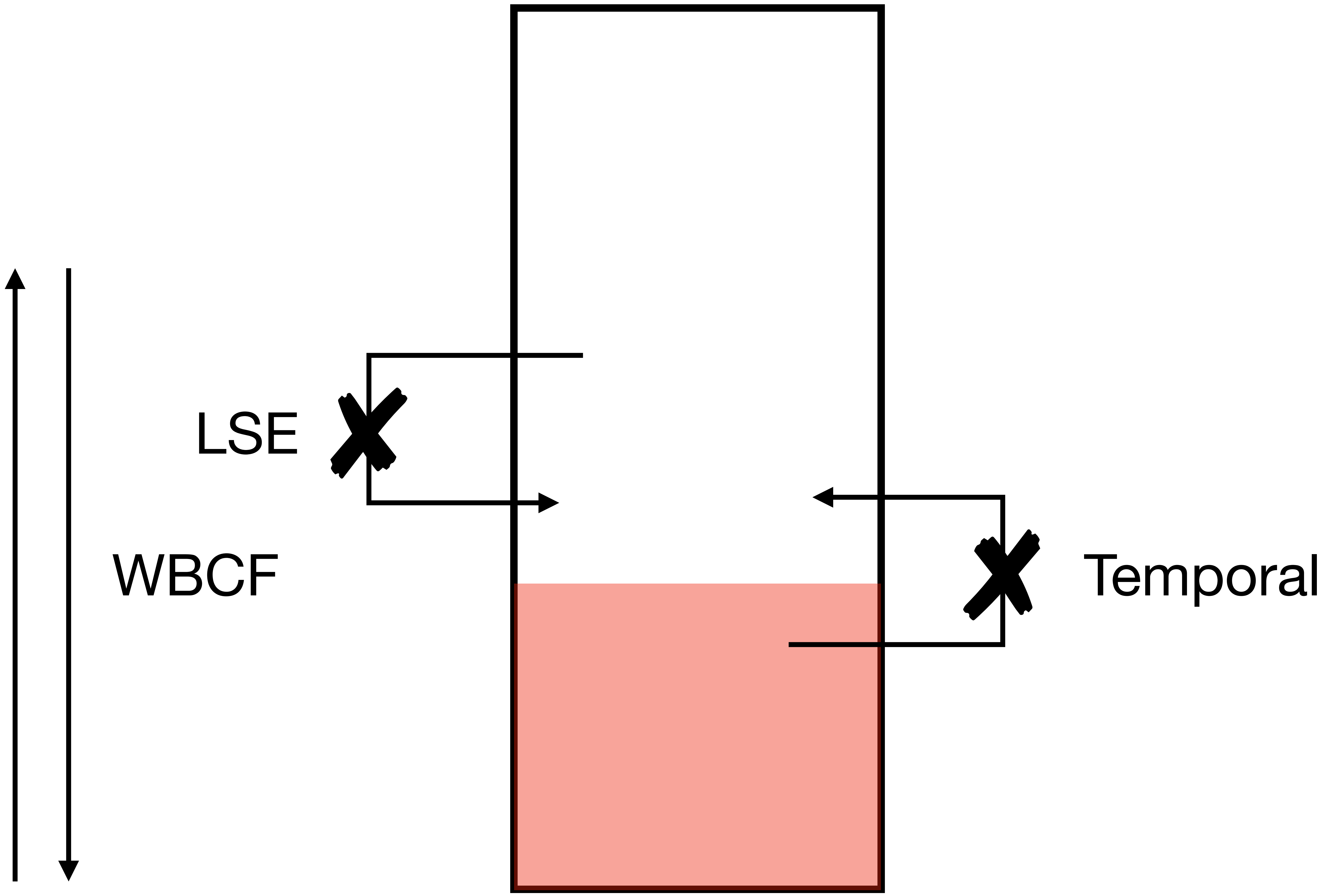
Stack Safety Properties

As stack frame discipline



Stack Safety Properties

As stack frame discipline



Enforcing Stack Safety using Capabilities

Background

Local Capabilities

[2018] Reasoning about a Machine with Local Capabilities
Skorstengaard et. al.

Linear Capabilities

[2019] Enforcing Well-Bracketed Control Flow and Local State Encapsulation using Linear Capabilities
Skorstengaard et. al.

<u>Uninitialized Capabilities</u>

[2021] Efficient and Provable Local Capability Revocation using Uninitialized Capabilities
Georges et. al.

Temporal Capabilities

[2019] Temporal Safety for Stack Allocated Memory on Capability Machines
Tsampas et. al.

Throughline

A safe stack enforces specific spatial and temporal properties to stack allocated memory

The authority granted by a stack capability must follow these exact properties, including the lifetime properties of stack frames

This requires some kind of “capability revocation” mechanism

Throughline

A safe stack enforces specific spatial and temporal properties to stack allocated memory

The authority granted by a stack capability must follow these exact properties, including the lifetime properties of stack frames

This requires some kind of “capability revocation” mechanism

… efficiently!

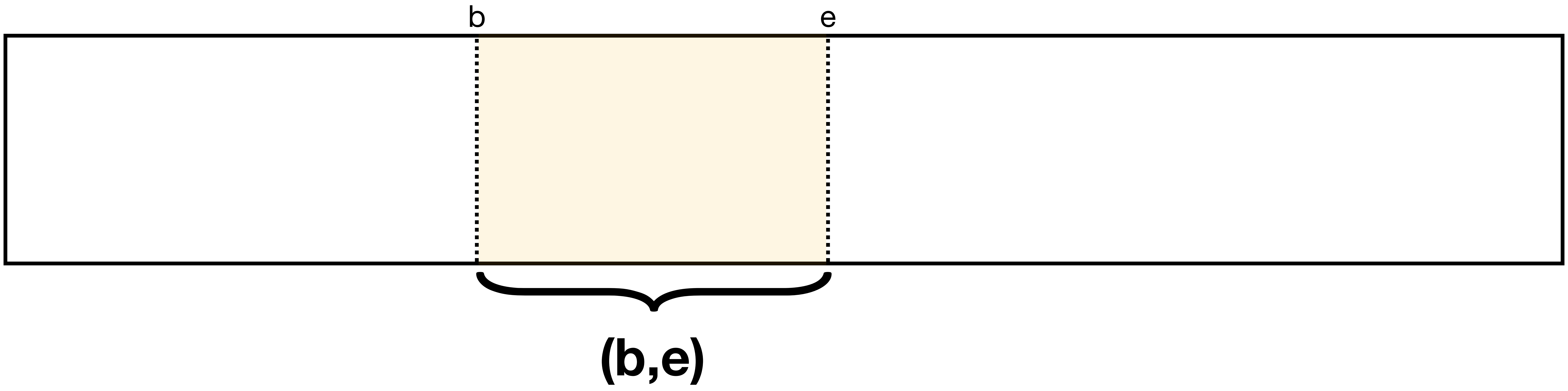
Capabilities for the heap and for the stack

Pointers as Capabilities

Pointers are replaced by hardware capabilities

Pointers as Capabilities

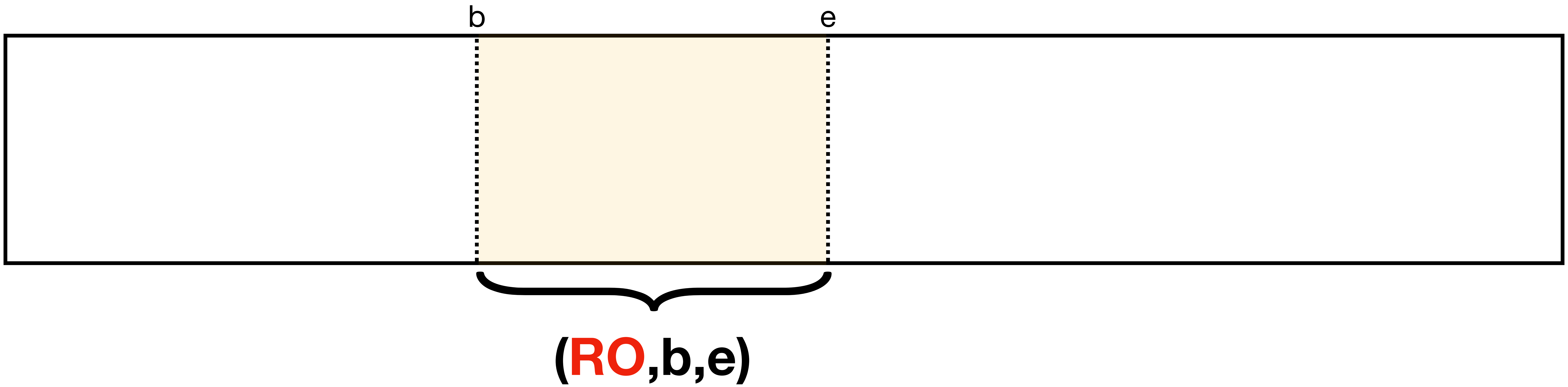
Pointers are replaced by hardware capabilities



- Bounds of authority

Pointers as Capabilities

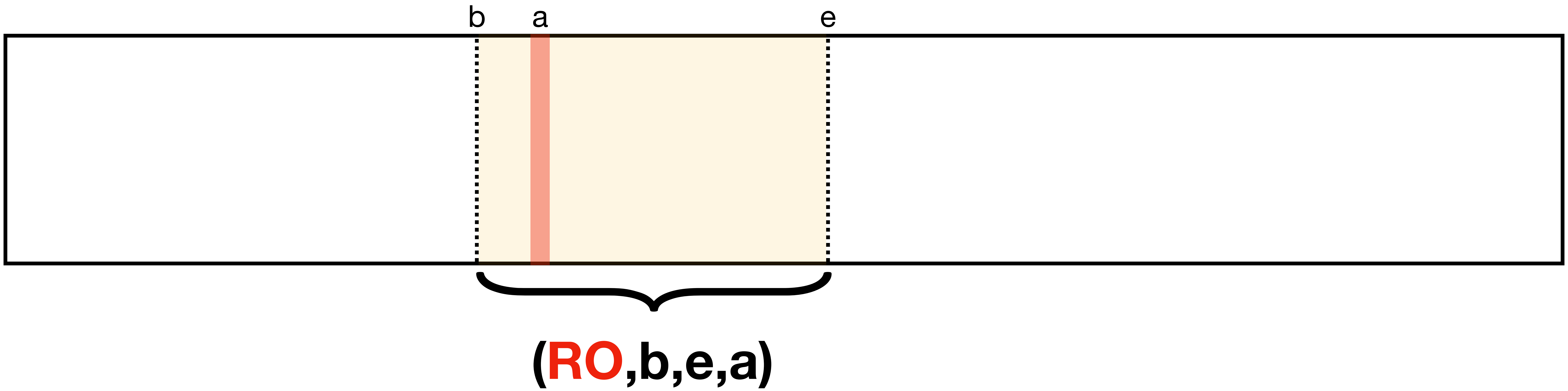
Pointers are replaced by hardware capabilities



- Bounds of authority
- Permission: RO/RW/etc

Pointers as Capabilities

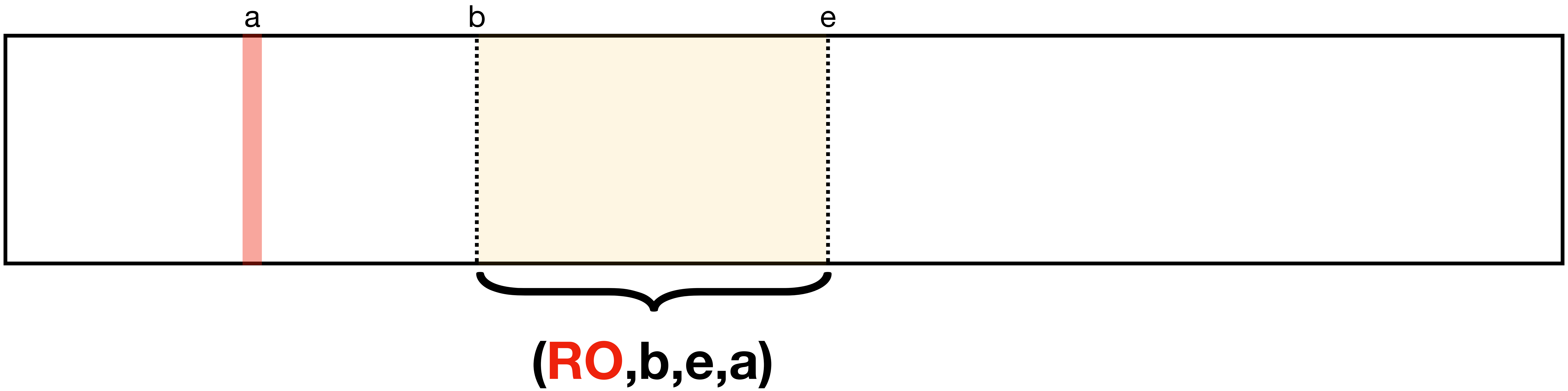
Pointers are replaced by hardware capabilities



- Bounds of authority
- Permission: RO/RW/etc
- Address

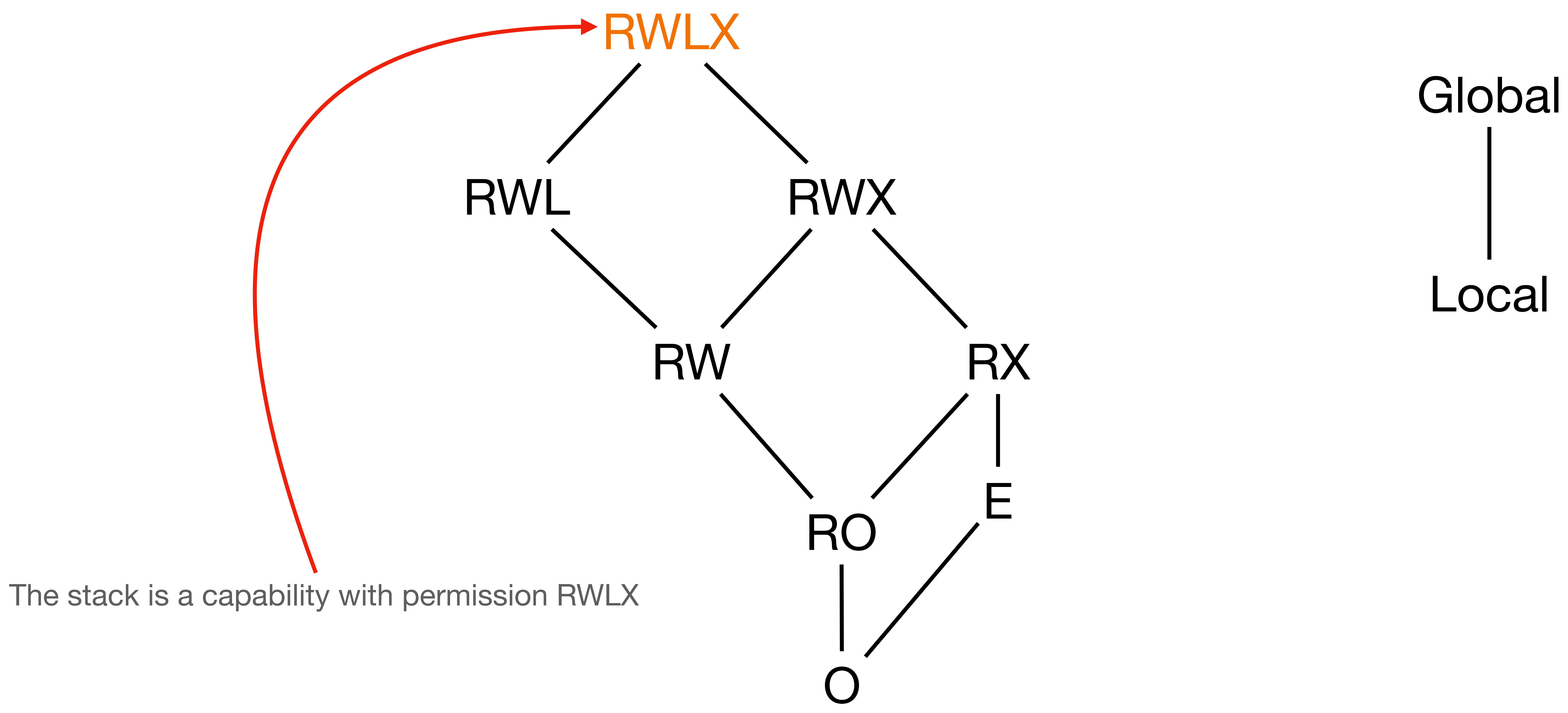
Pointers as Capabilities

Pointers are replaced by hardware capabilities

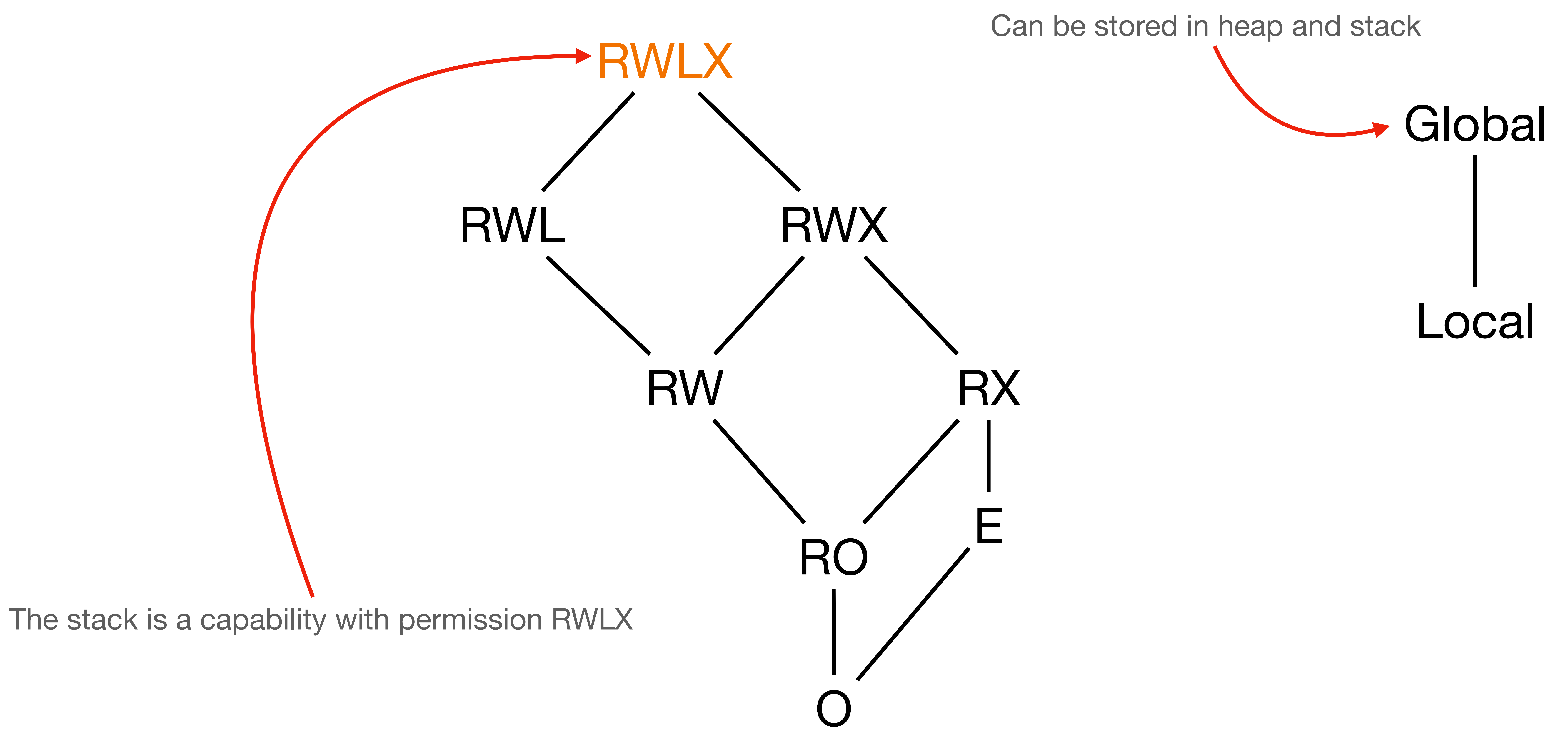


- Bounds of authority
- Permission: RO/RW/etc
- Address

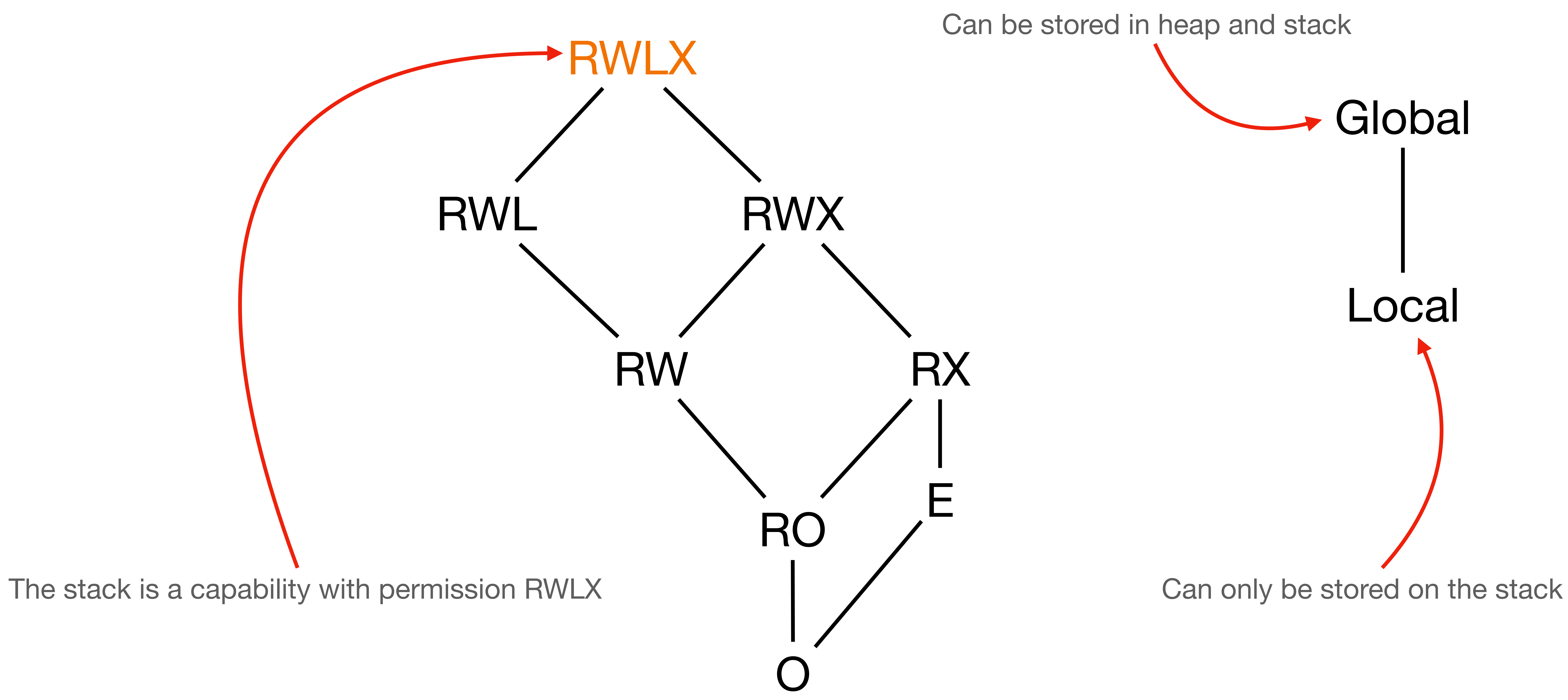
A Lattice of Permissions



A Lattice of Permissions

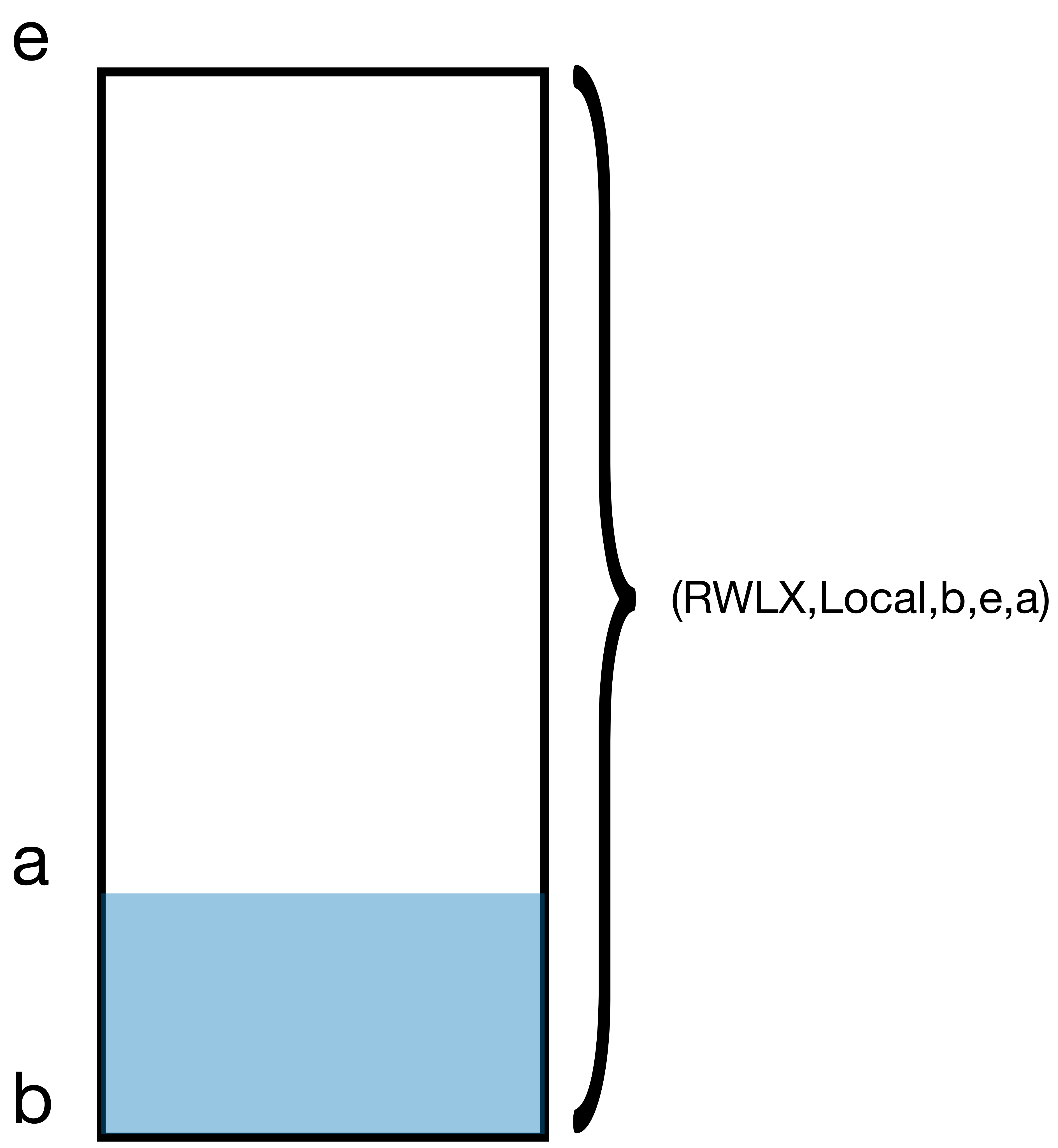


A Lattice of Permissions

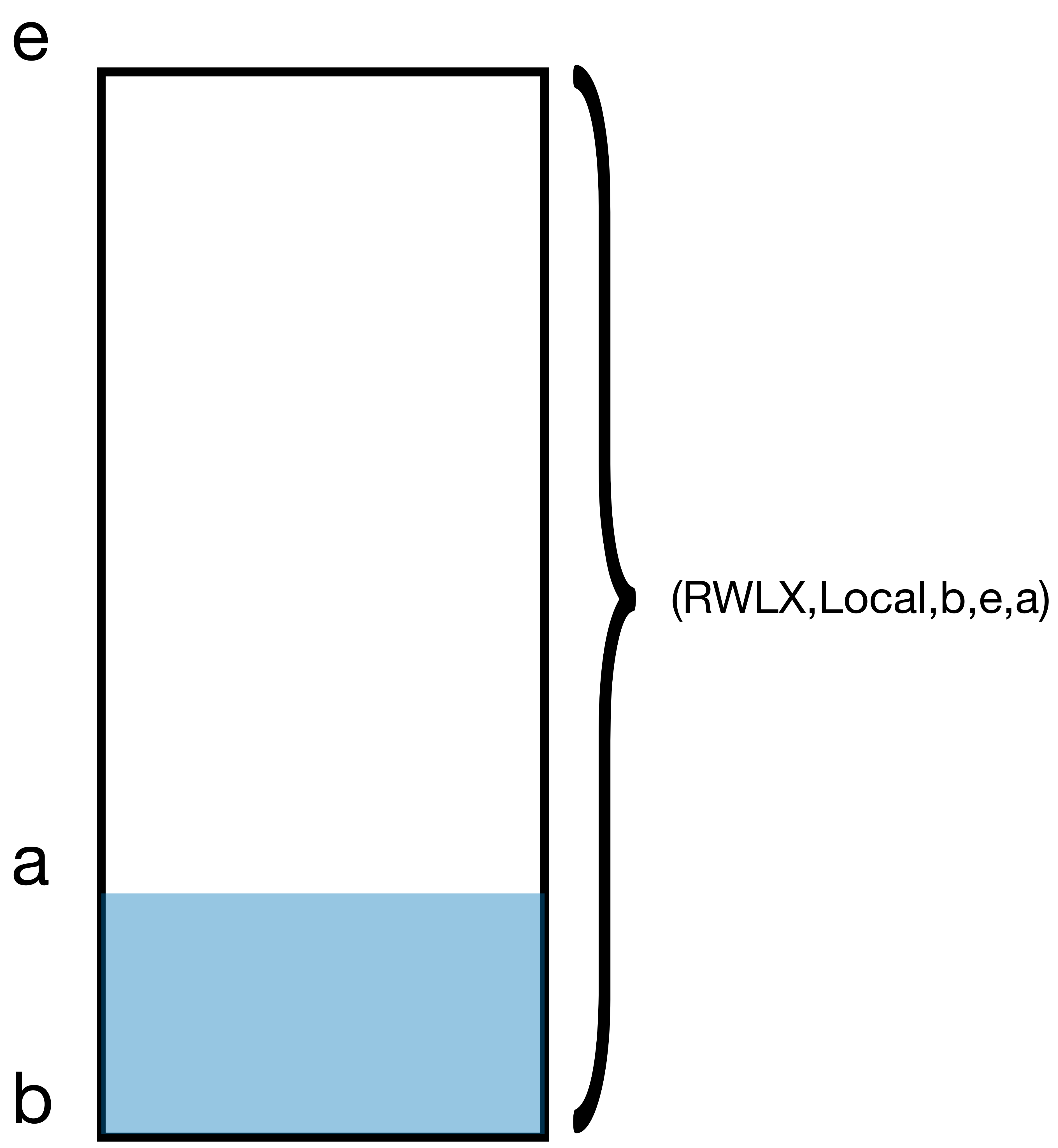


Capability Revocation

Calling Convention: Before Calling an Adversary

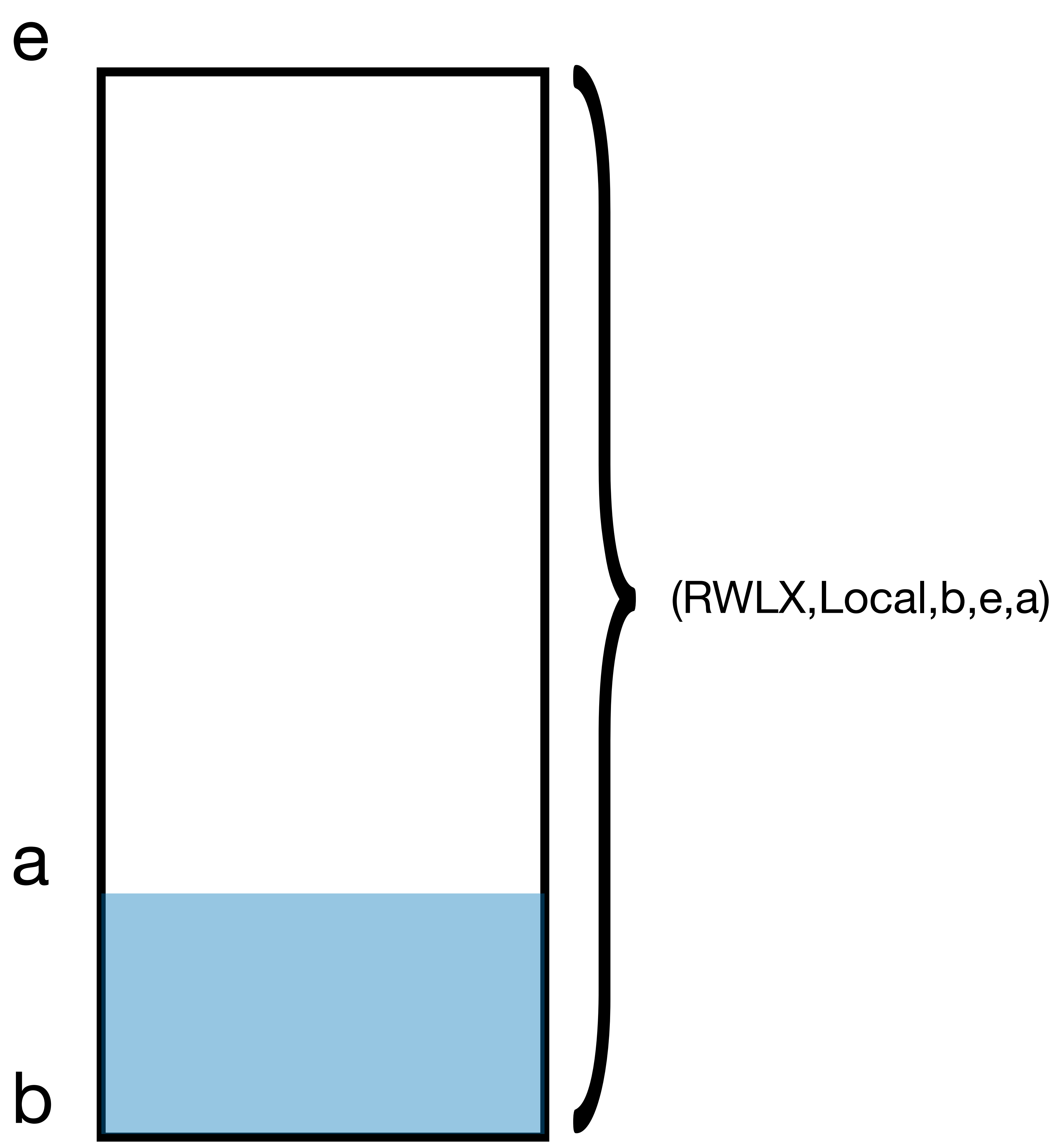


Calling Convention: Before Calling an Adversary



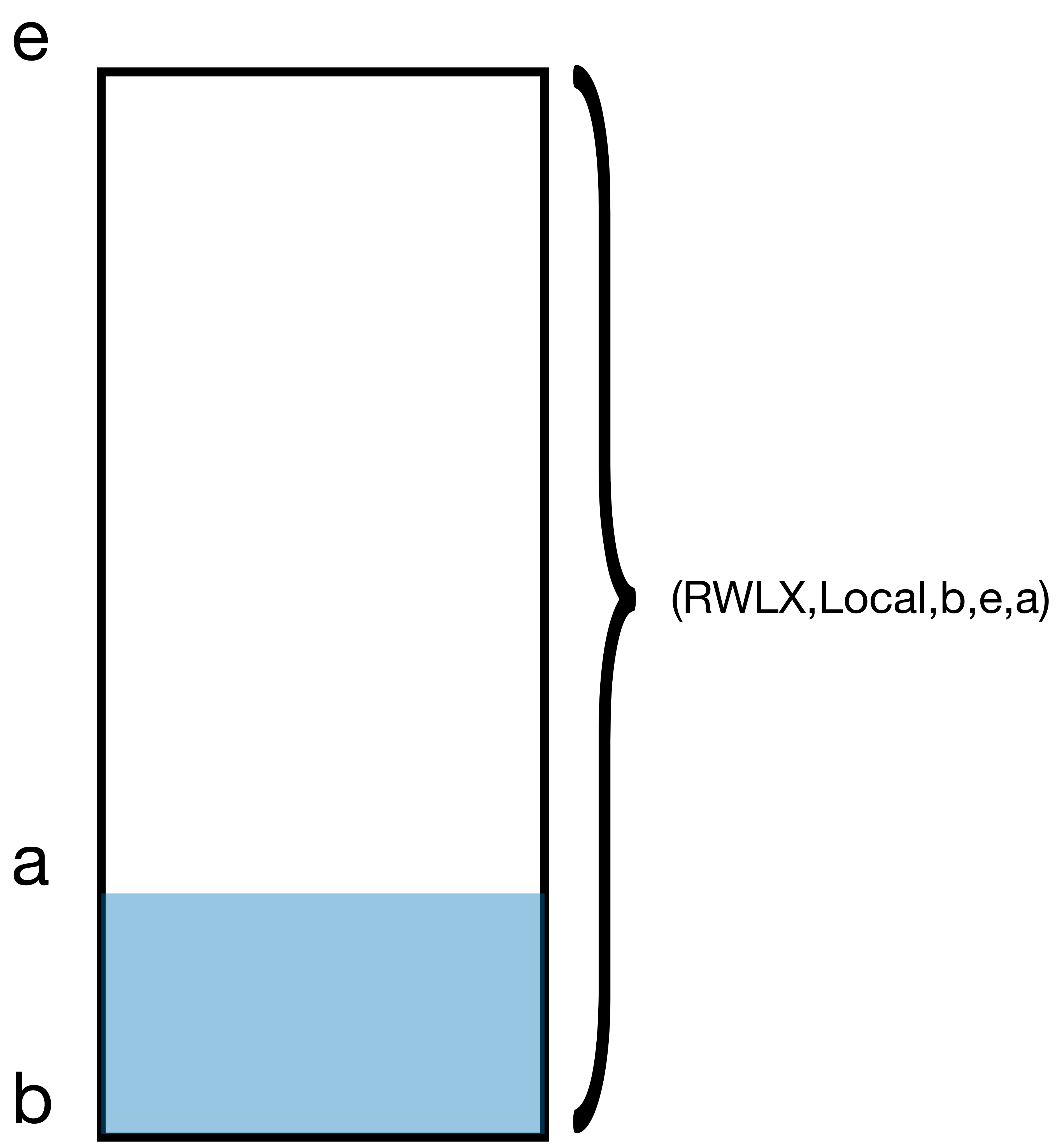
- Create an activation record that can:

Calling Convention: Before Calling an Adversary



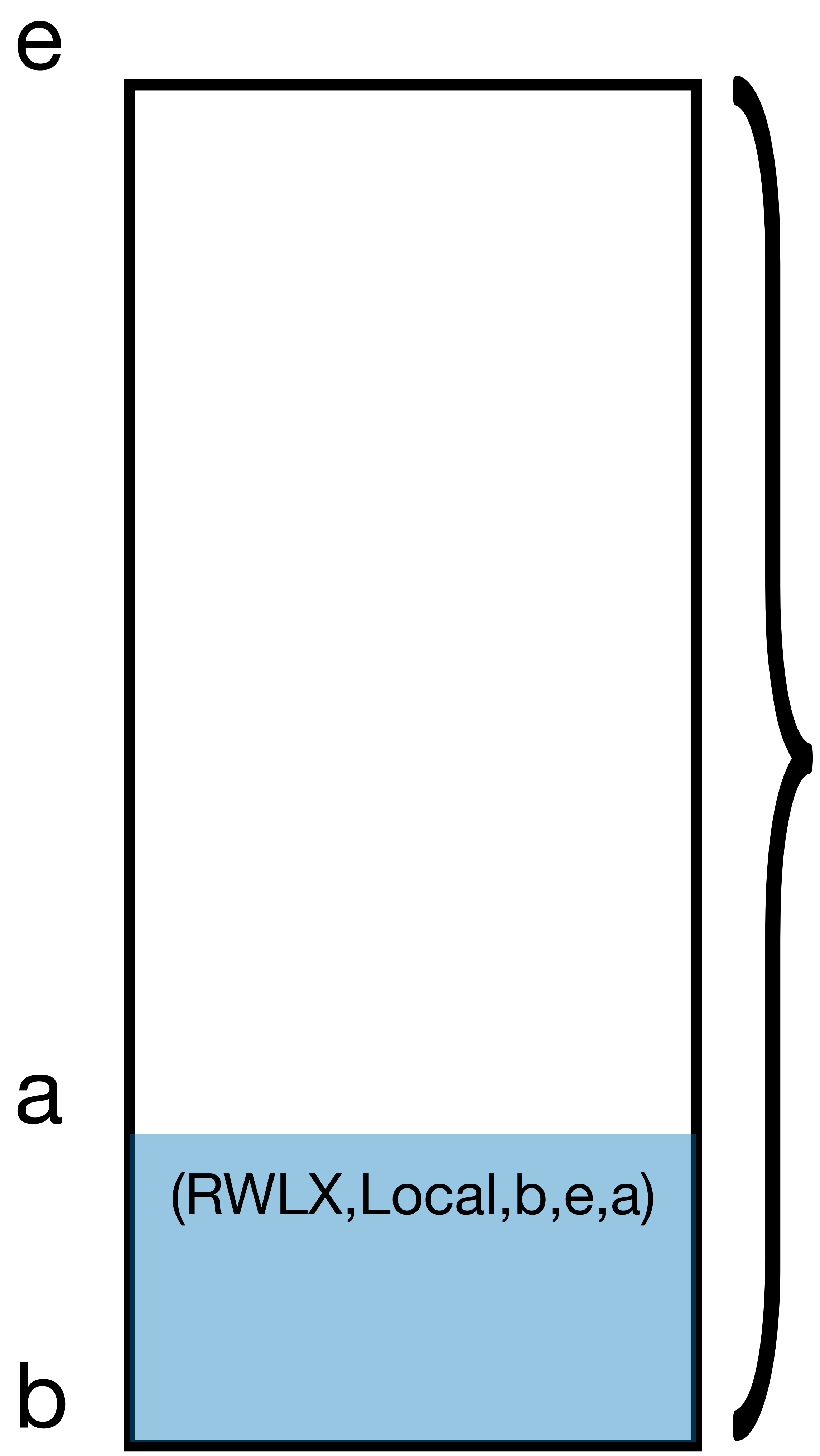
- Create an activation record that can:
  - Reinstate the old stack pointer

Calling Convention: Before Calling an Adversary



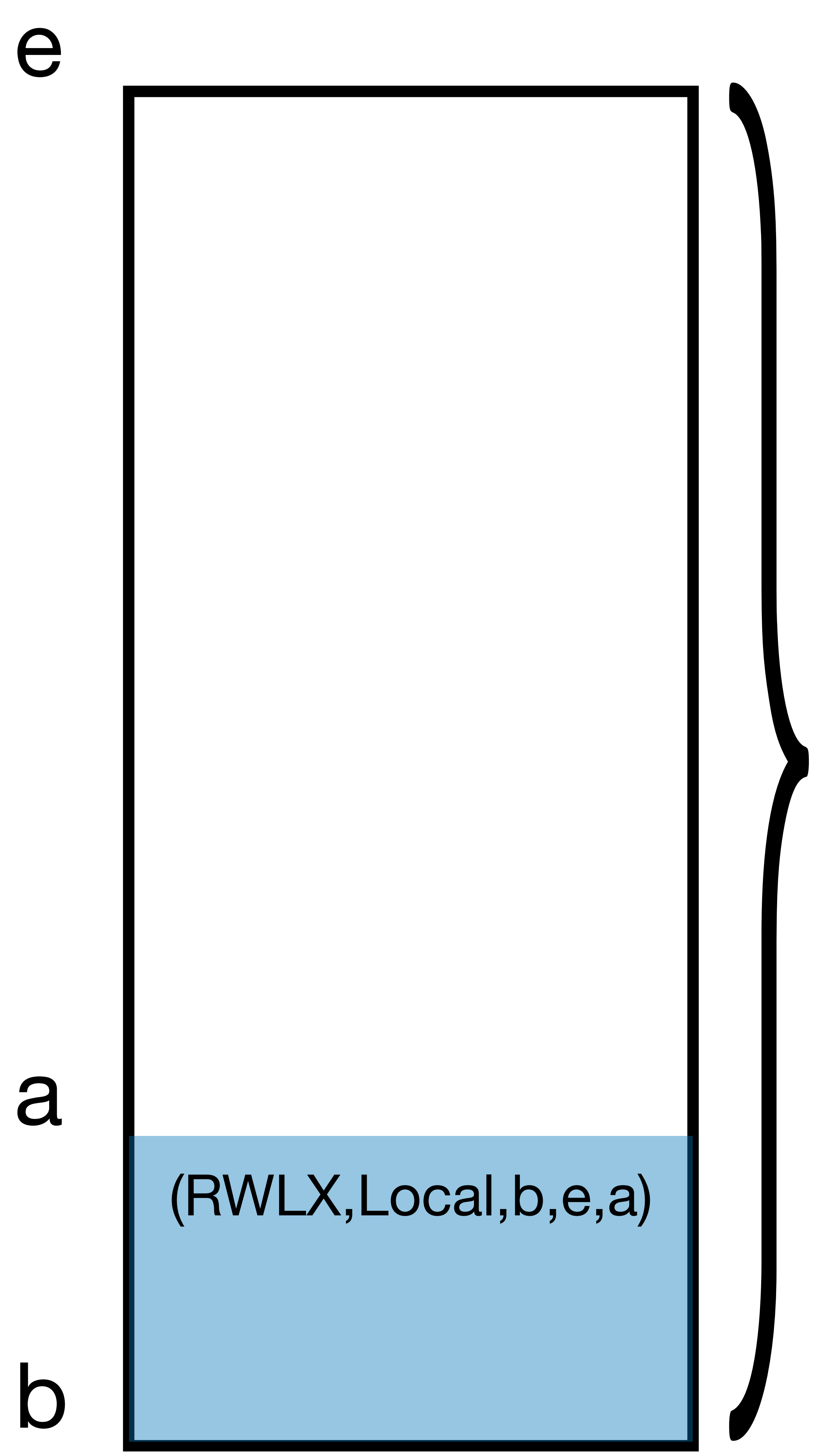
- Create an activation record that can:
    - Reinstate the old stack pointer
    - Update PC to the next instruction in program

Calling Convention: Before Calling an Adversary



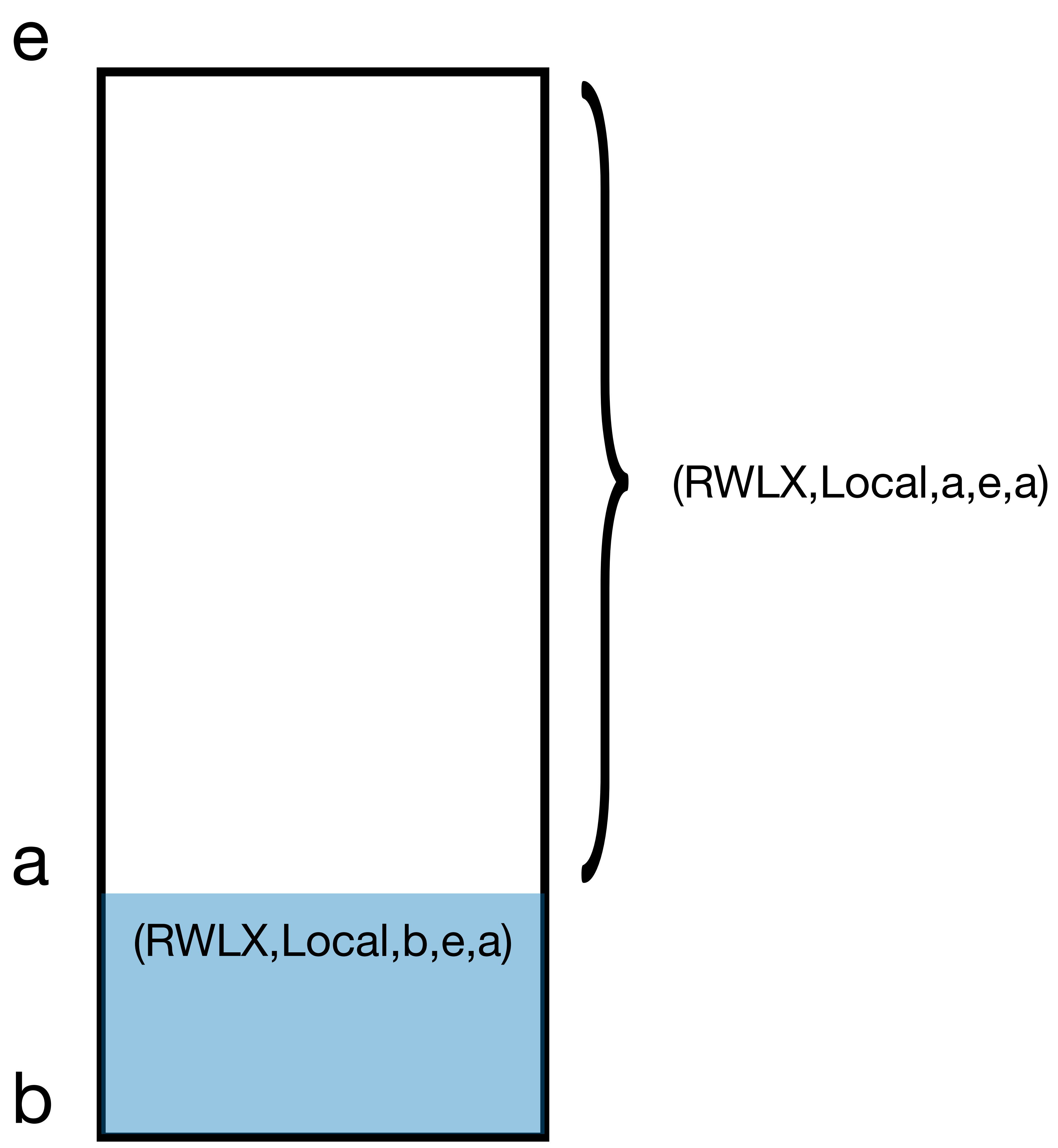
- Create an activation record that can:
  - Reinstate the old stack pointer
  - Update PC to the next instruction in program

Calling Convention: Before Calling an Adversary



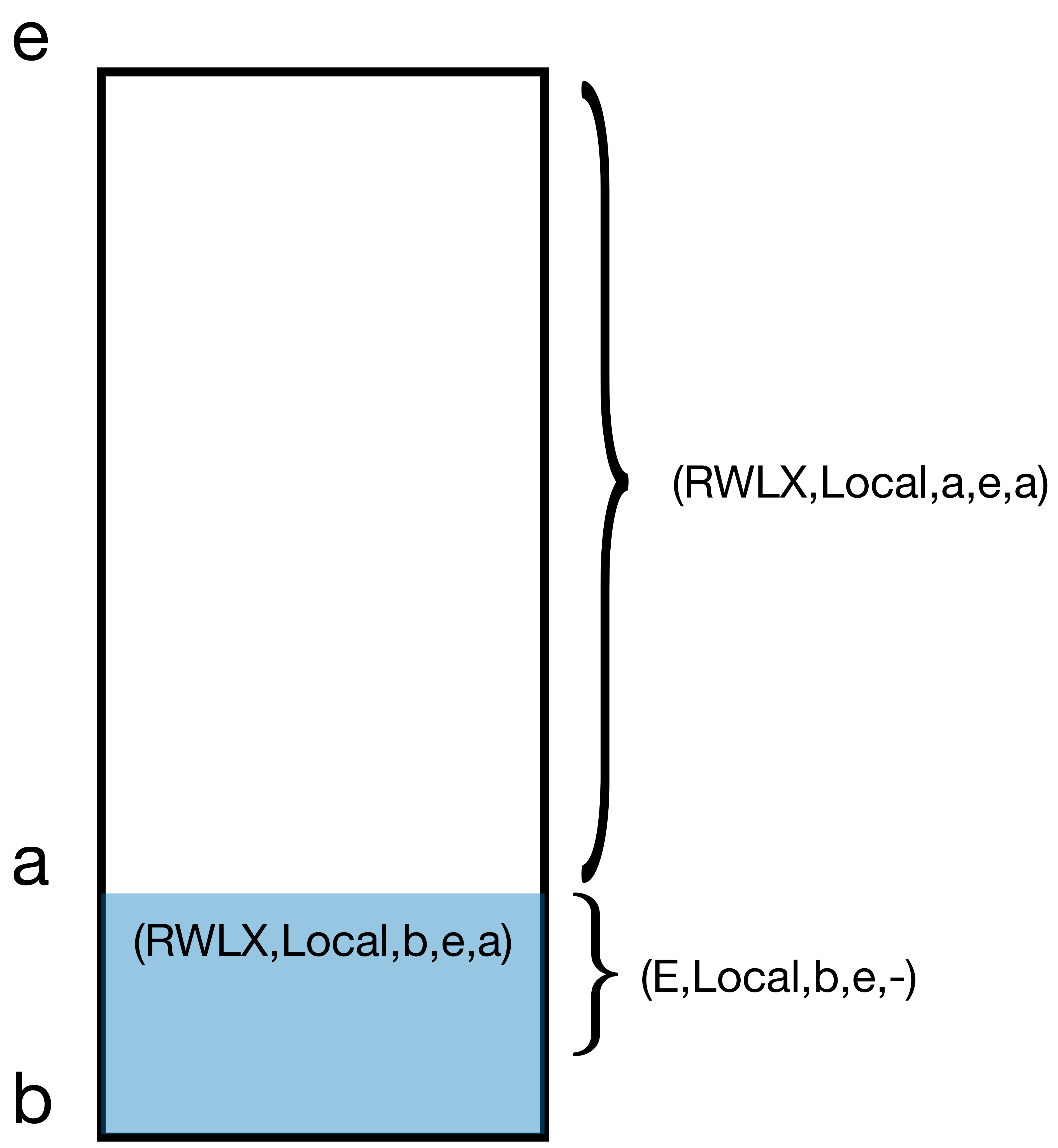
- Create an activation record that can:
    - Reinstate the old stack pointer
    - Update PC to the next instruction in program
- Activation record is stored on the stack

Calling Convention: Before Calling an Adversary



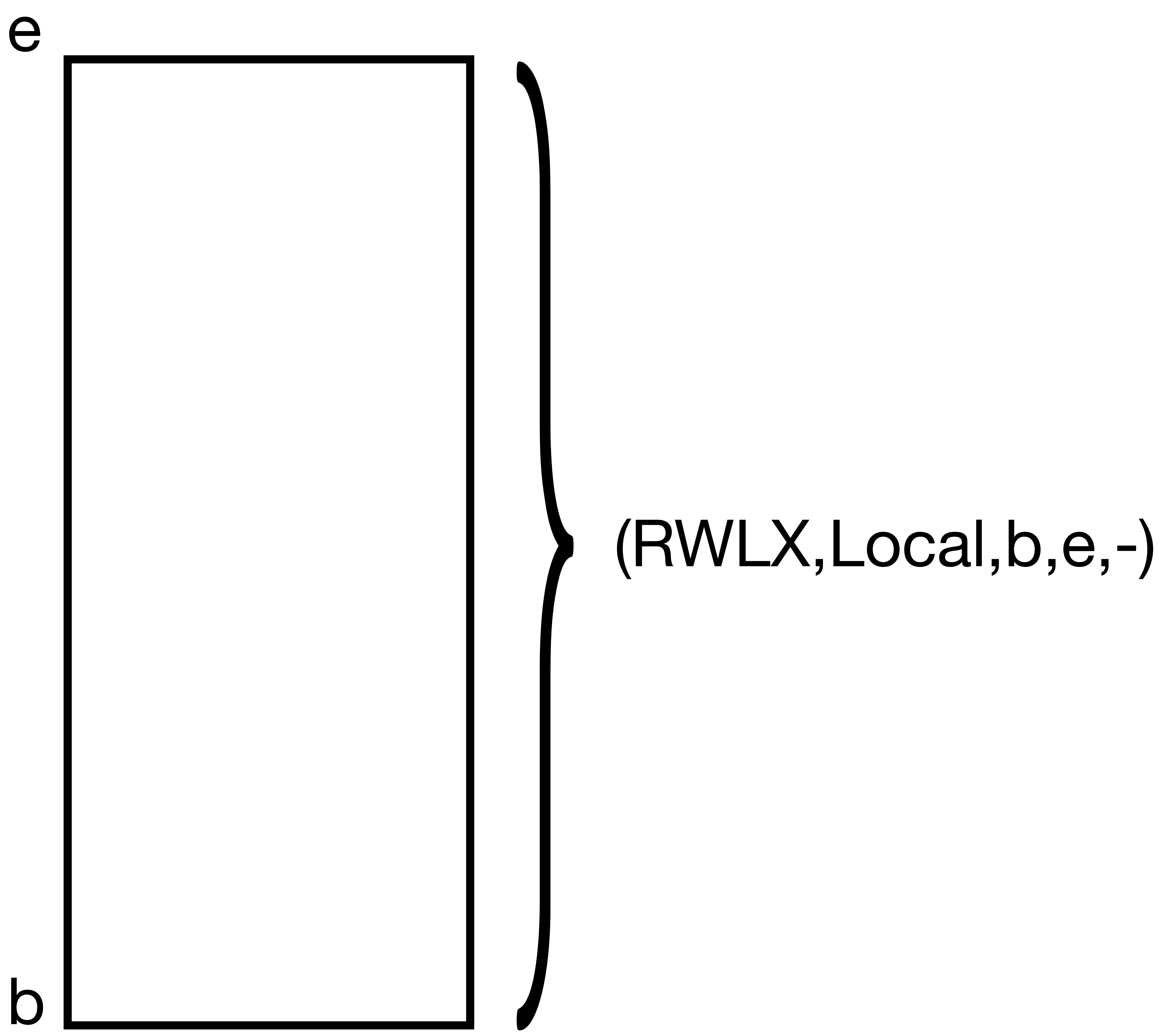
- Create an activation record that can:
  - Reinstate the old stack pointer
  - Update PC to the next instruction in program
- Activation record is stored on the stack

Calling Convention: Before Calling an Adversary

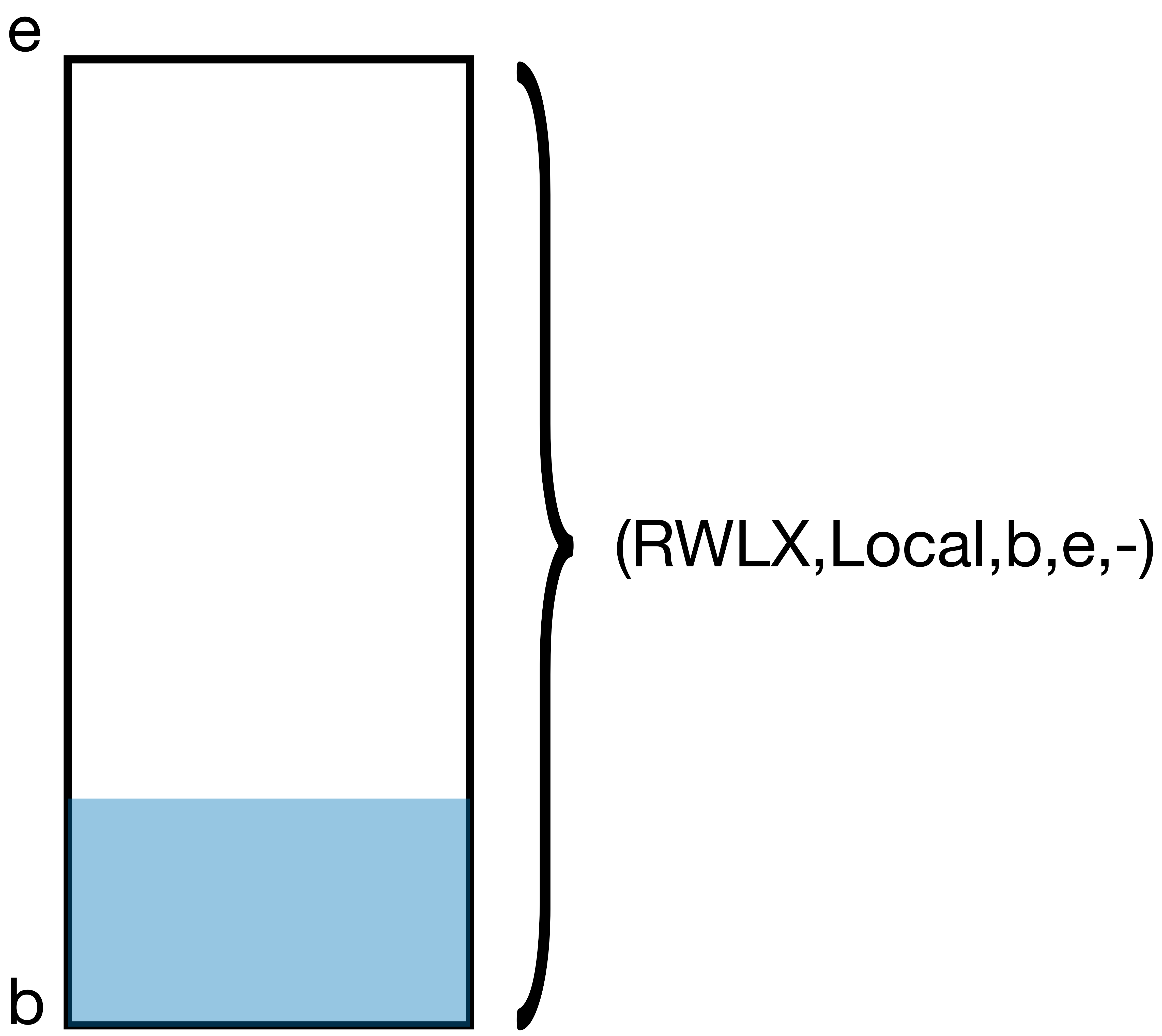


- Create an activation record that can:
  - Reinstate the old stack pointer
  - Update PC to the next instruction in program
- Activation record is stored on the stack

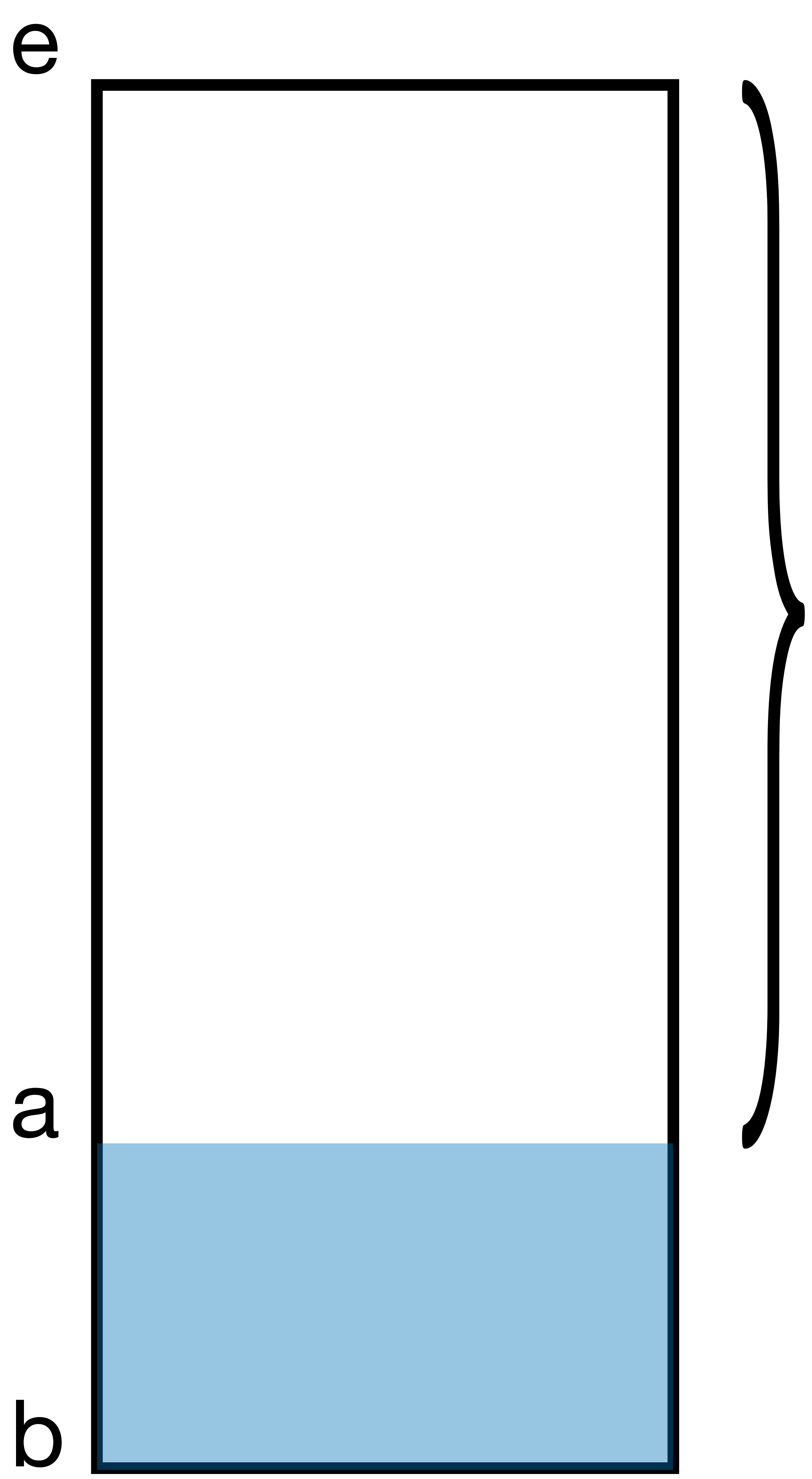
Motivating Revocation



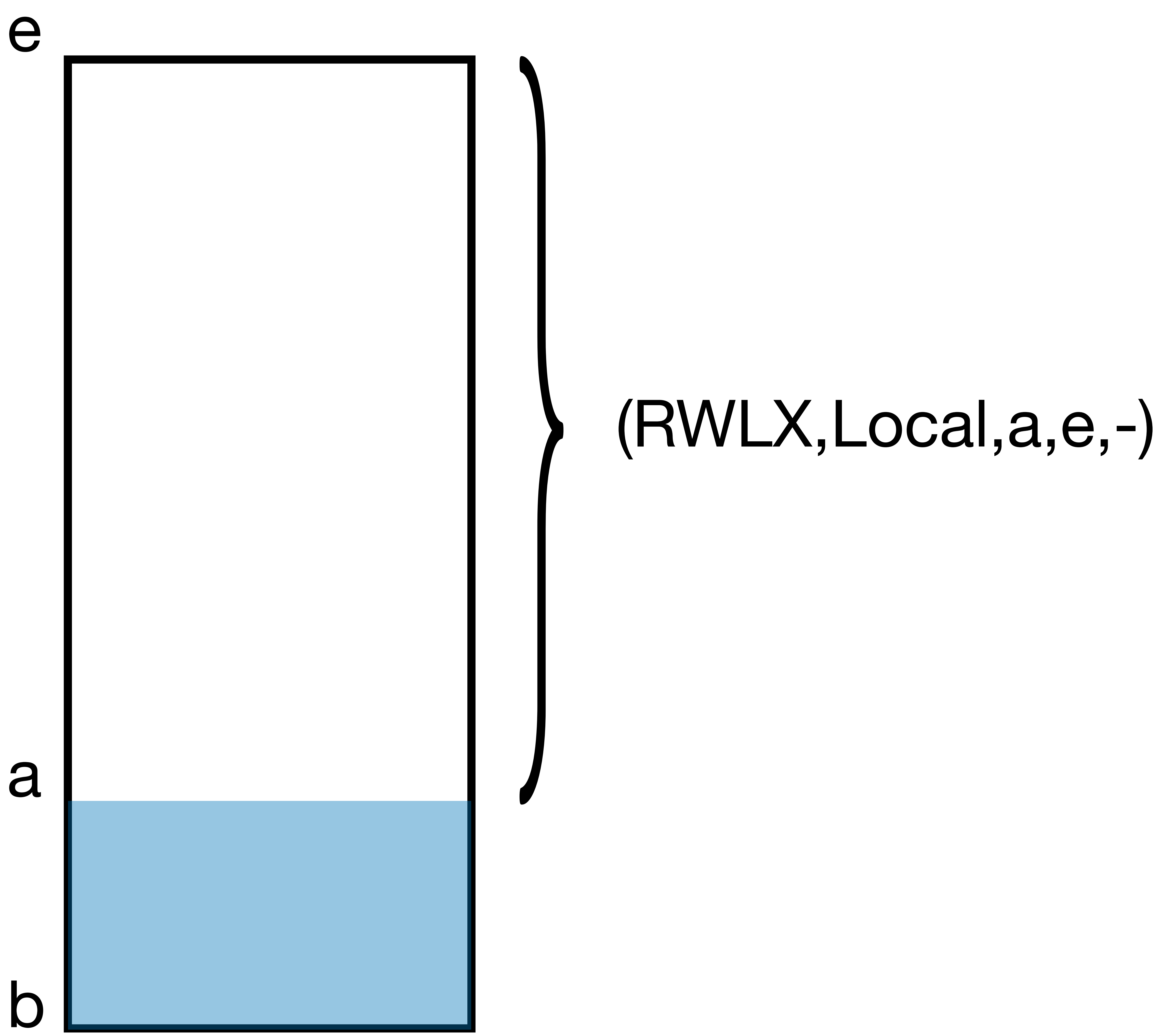
Motivating Revocation



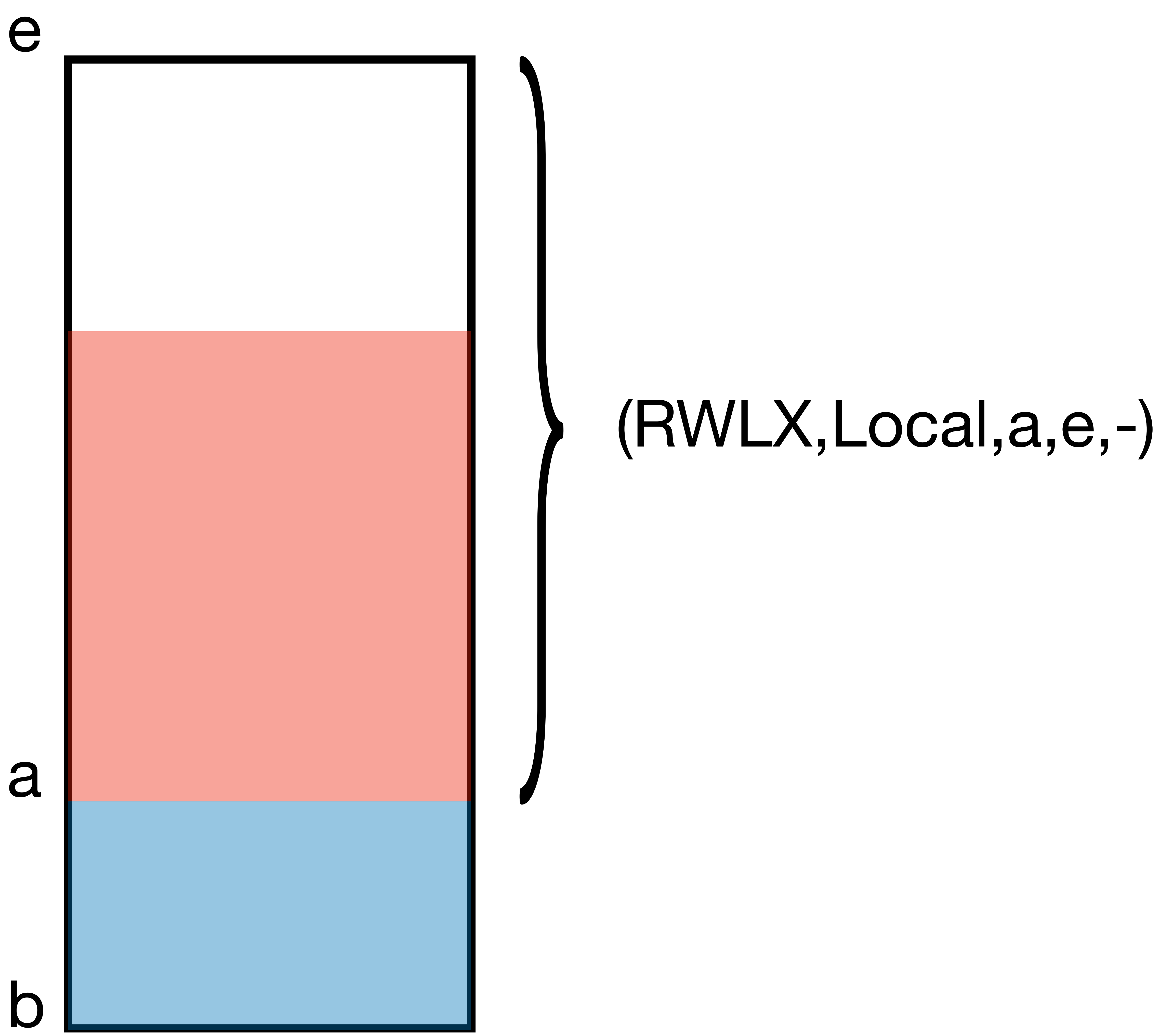
Motivating Revocation



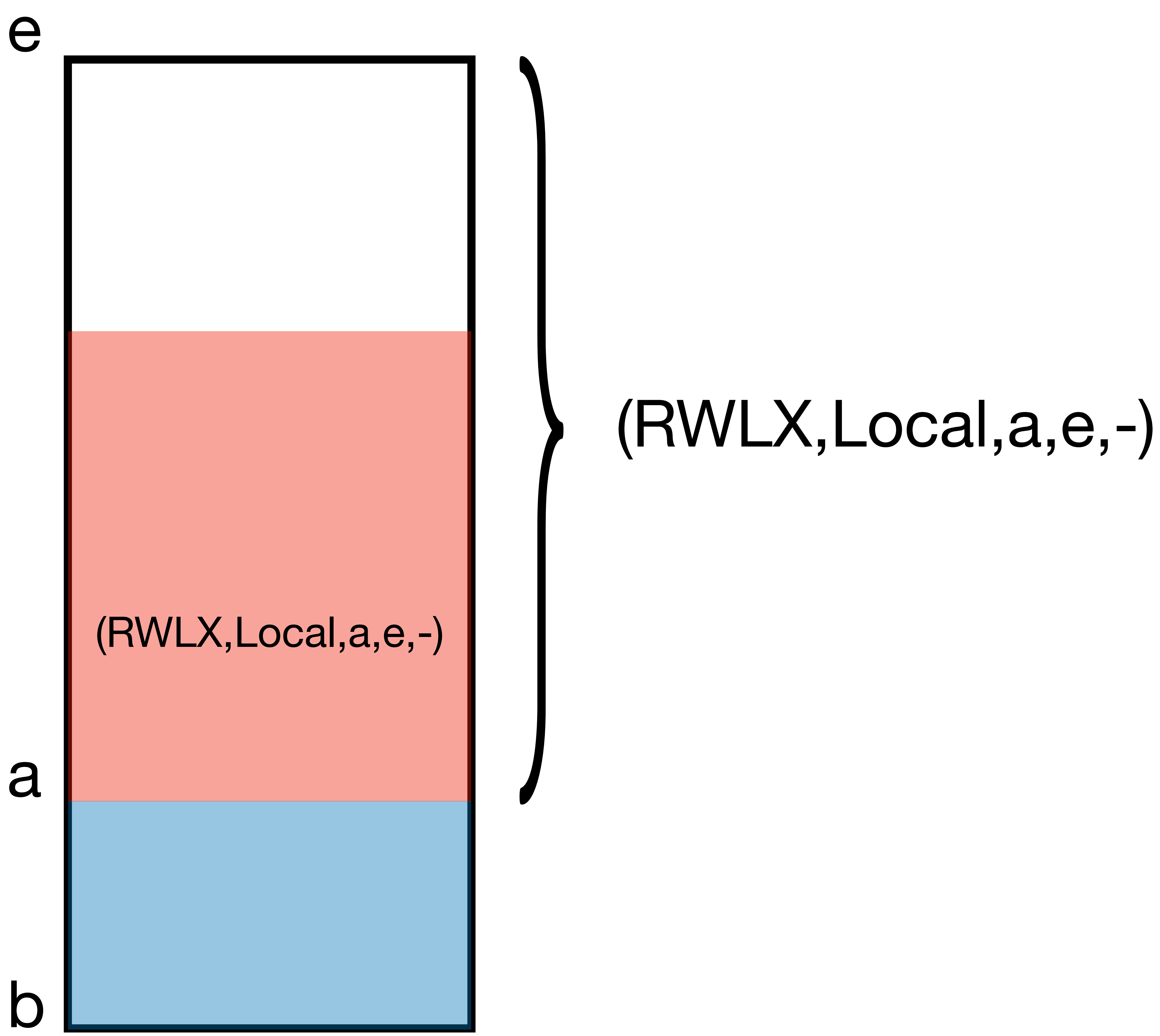
Motivating Revocation



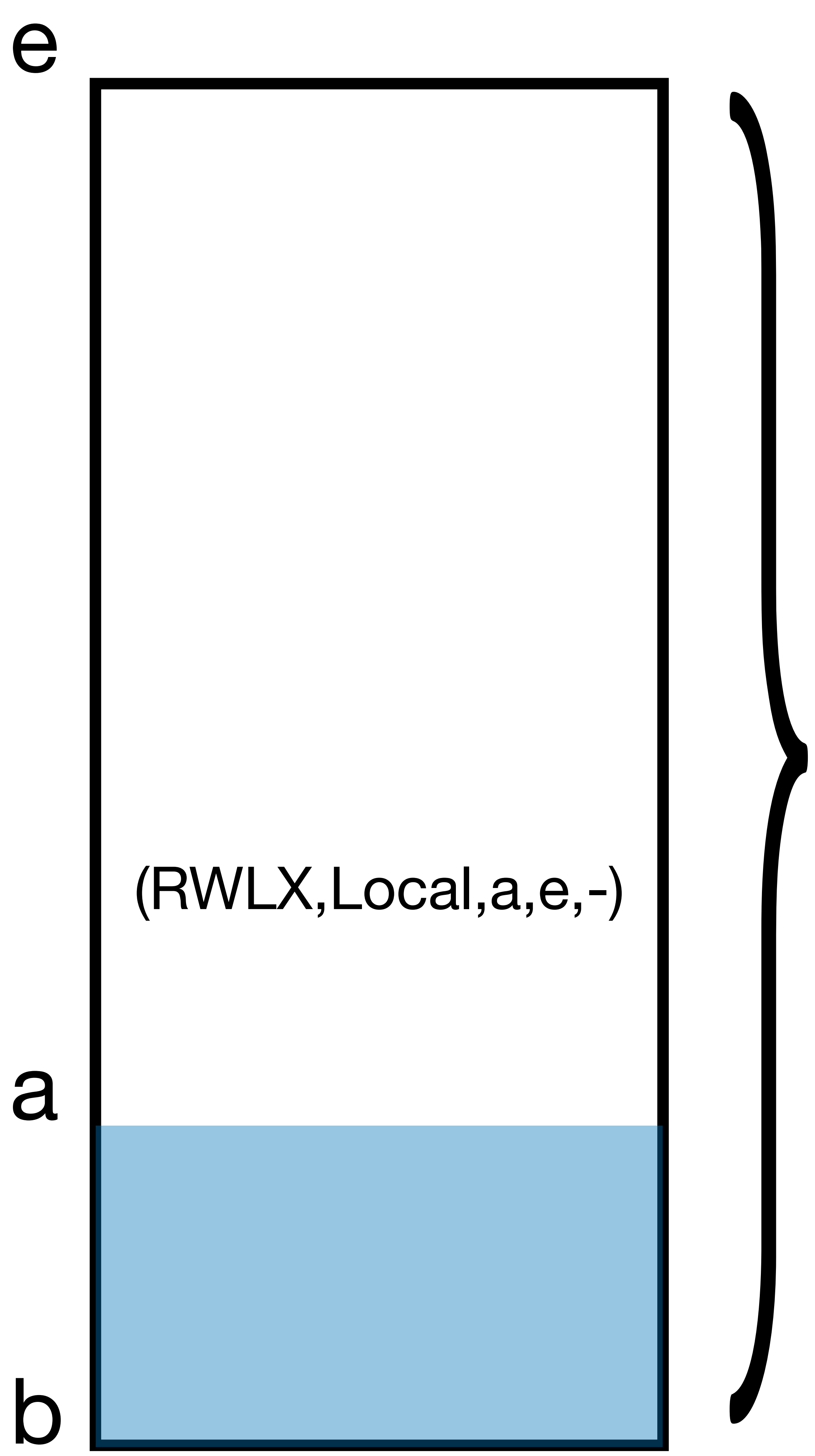
Motivating Revocation



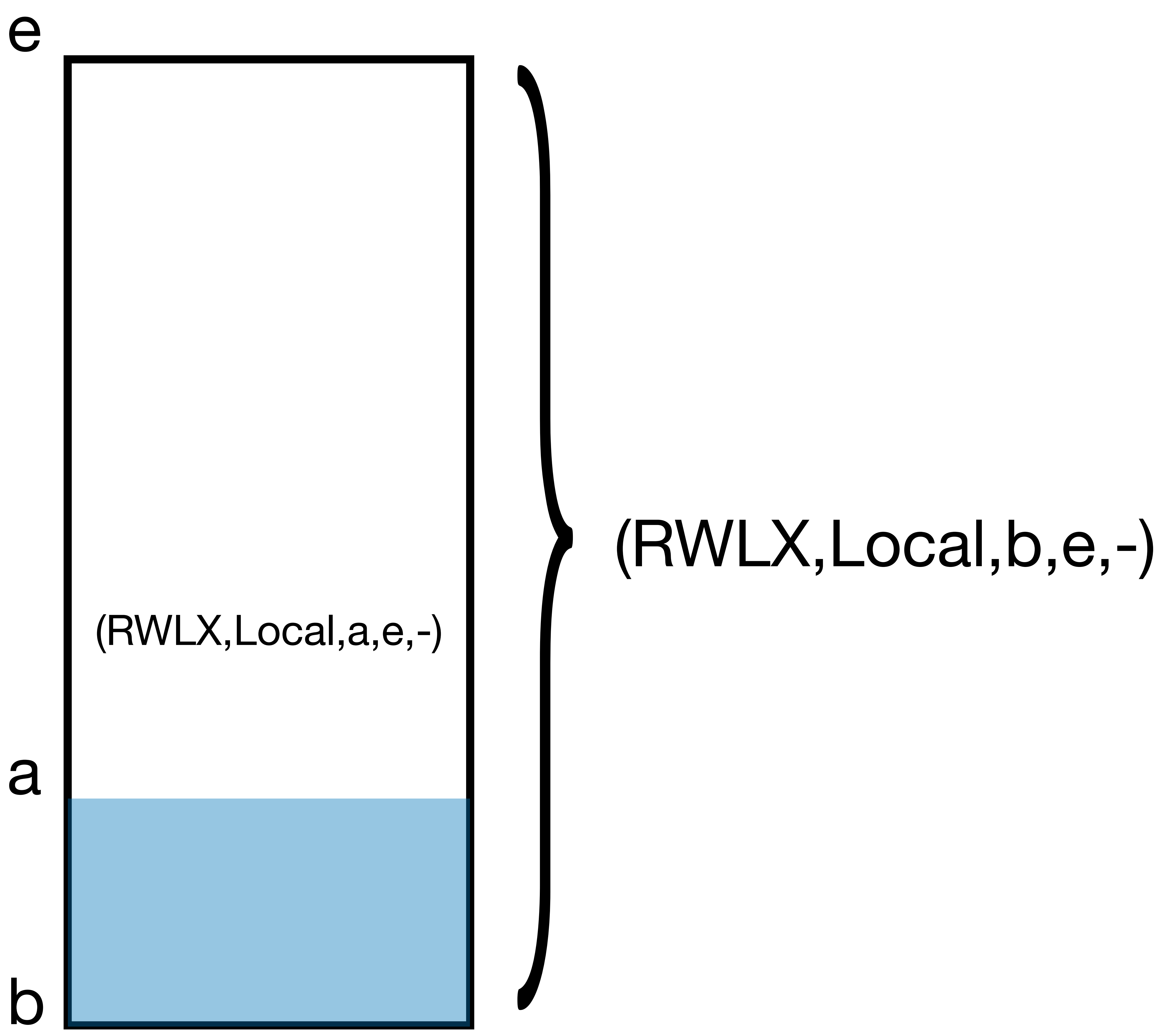
Motivating Revocation



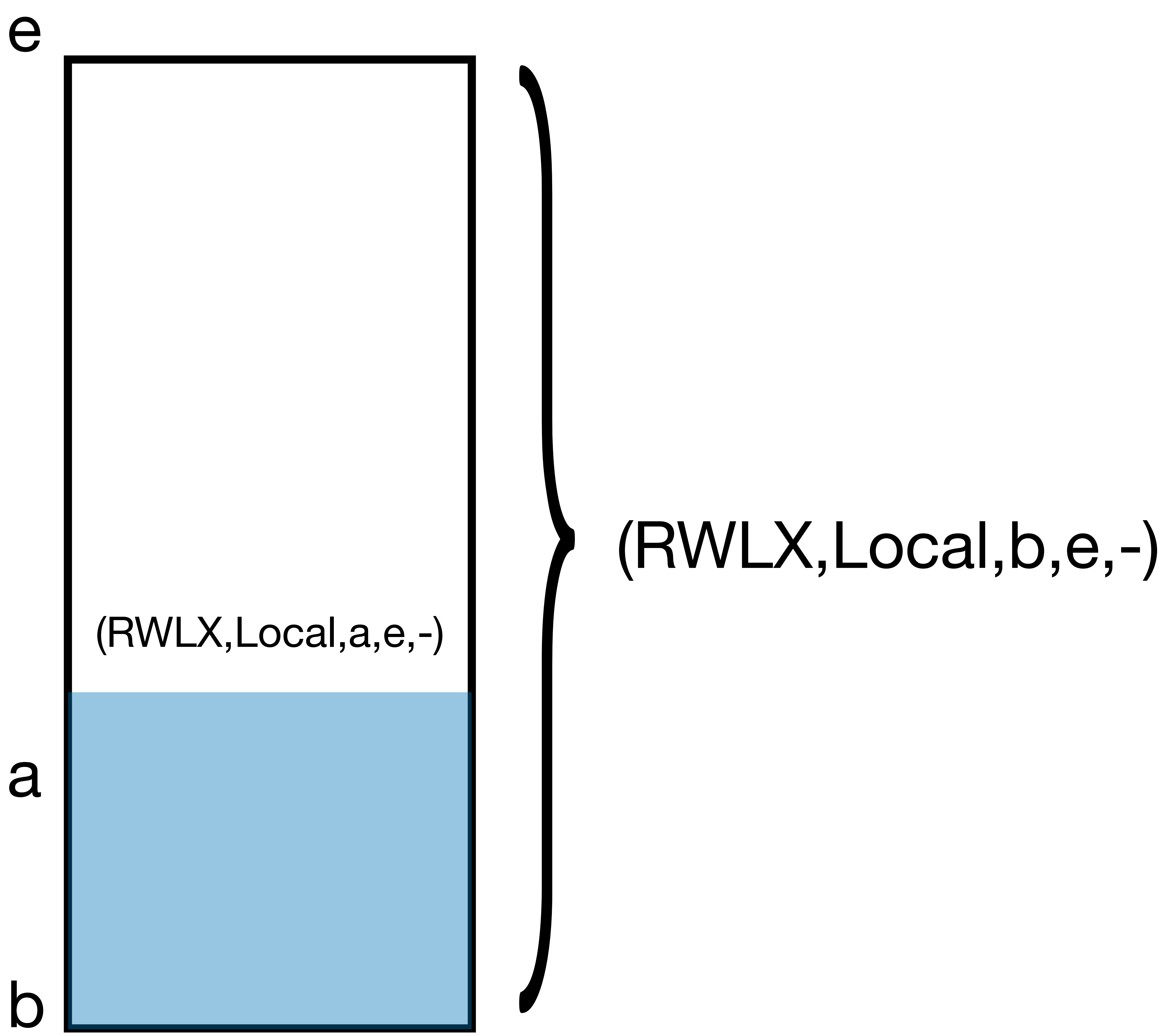
Motivating Revocation



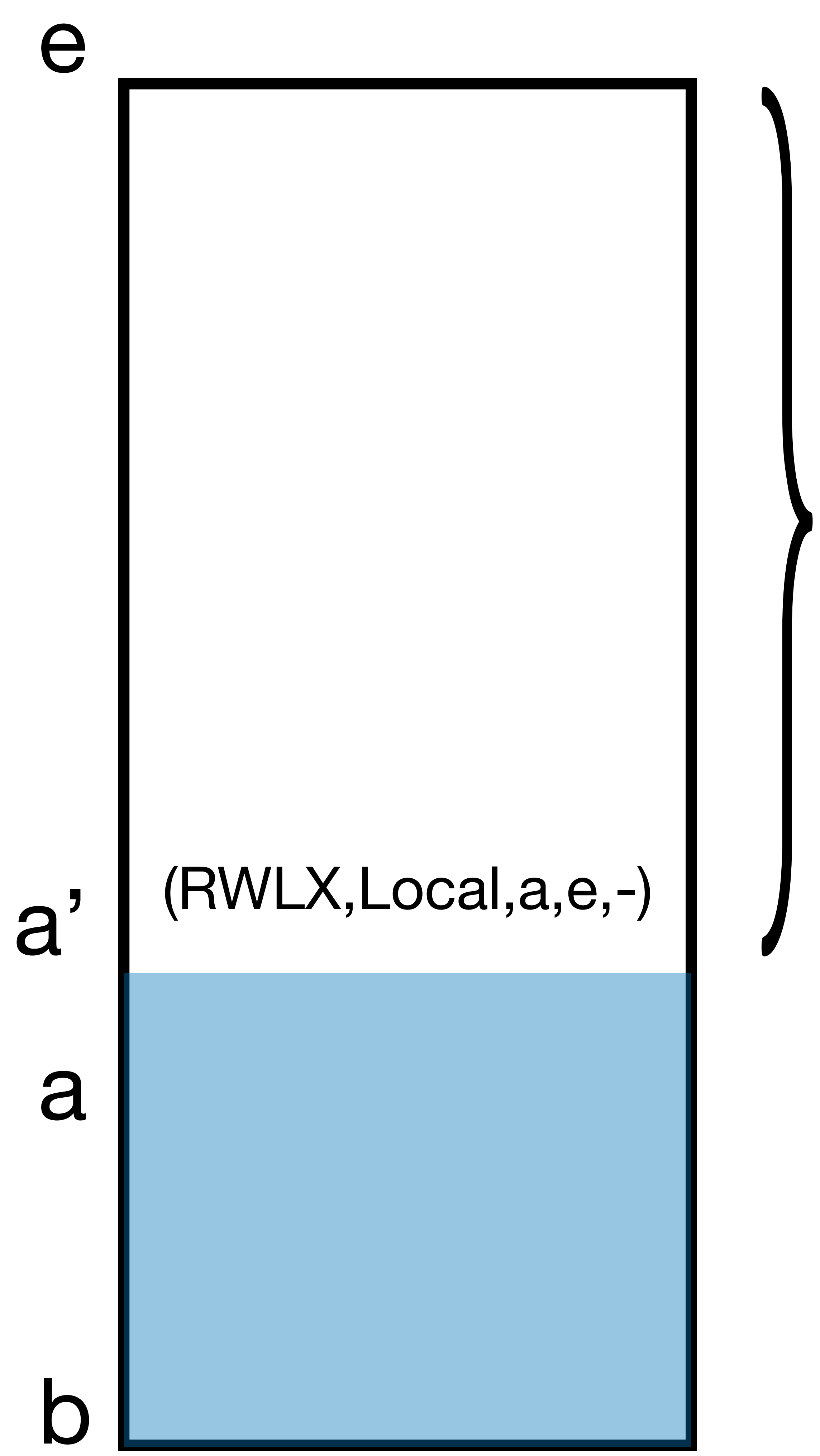
Motivating Revocation



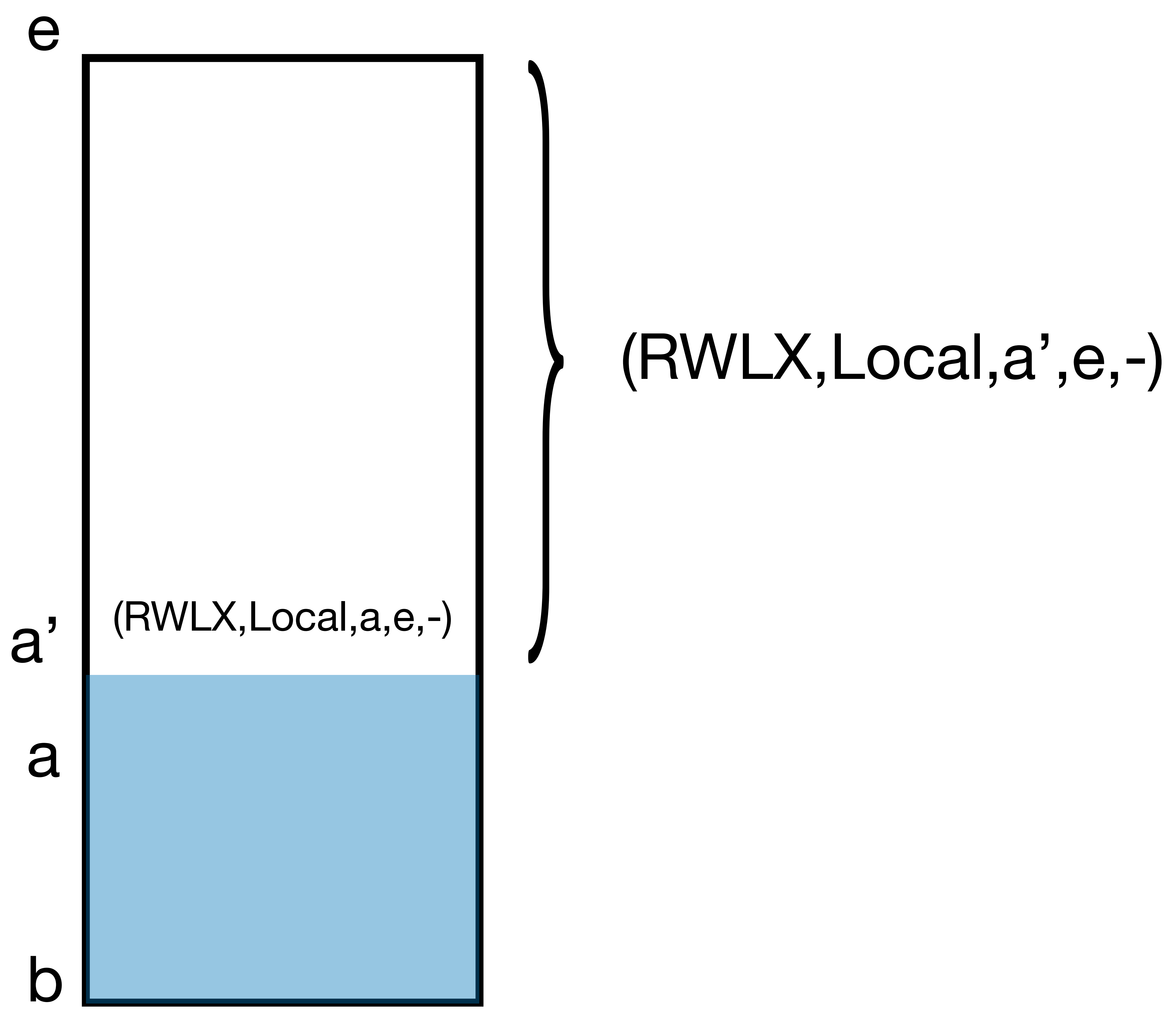
Motivating Revocation



Motivating Revocation

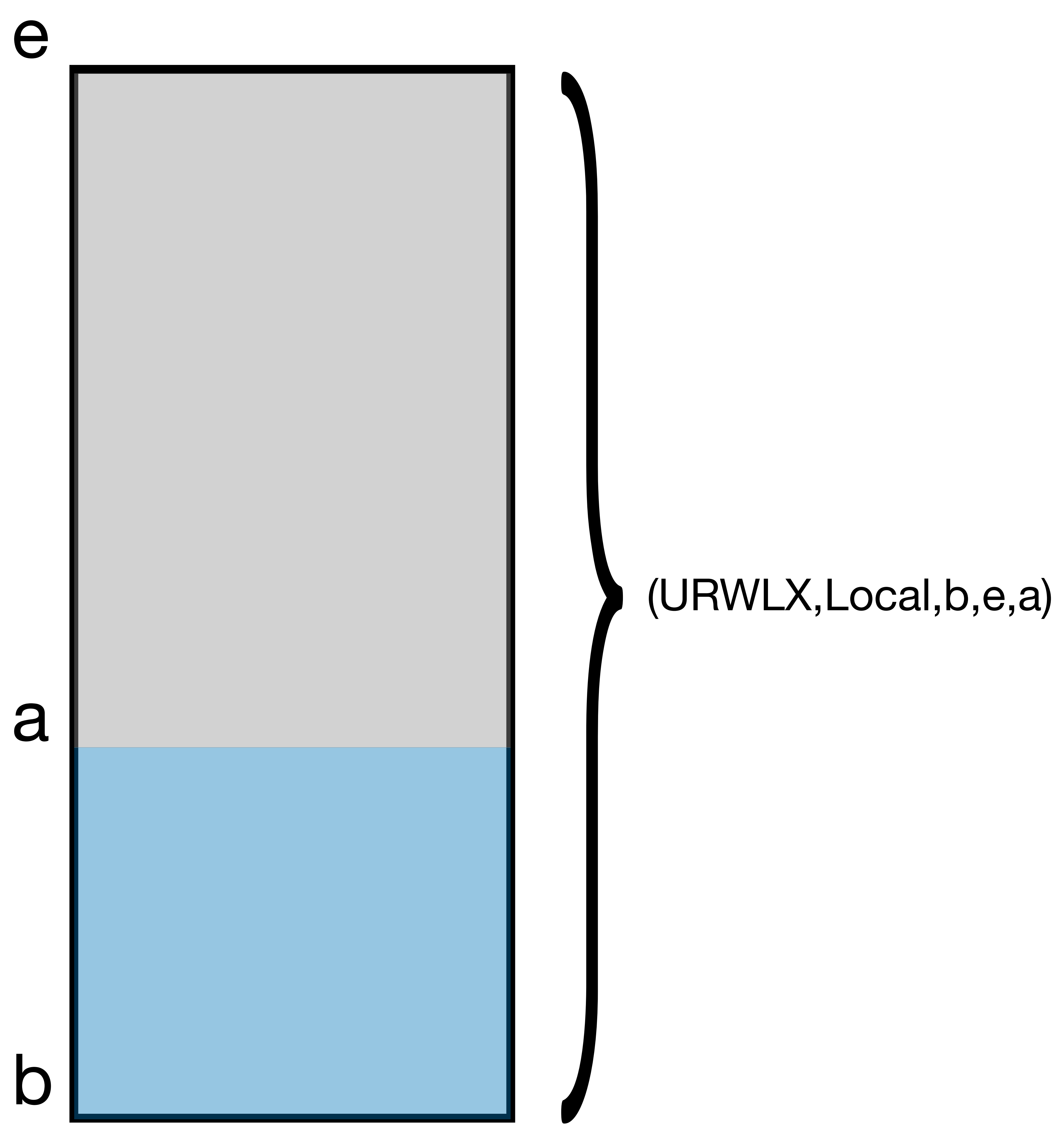


Motivating Revocation



Capability Revocation: Callee

Uninitialized capabilities

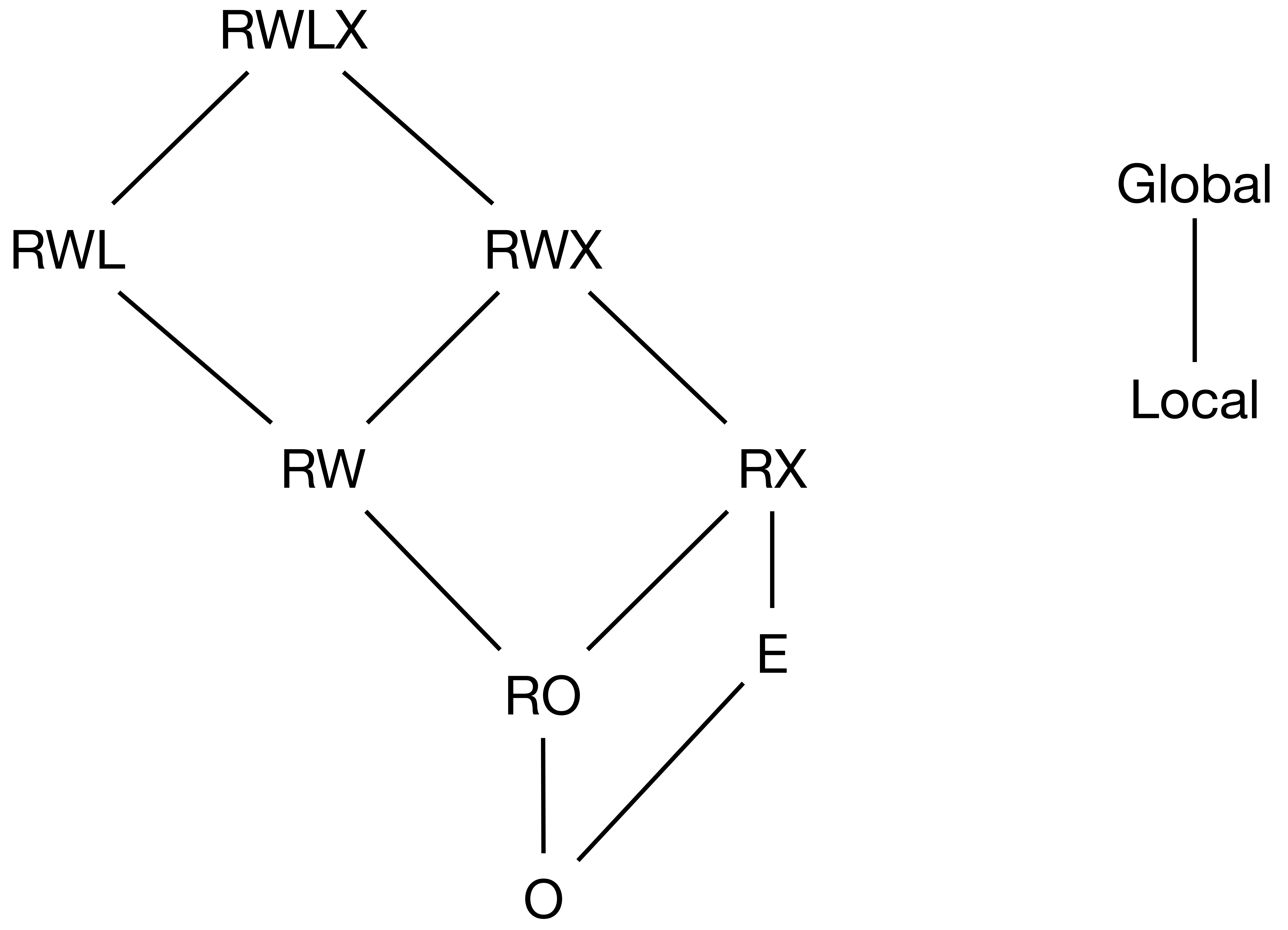


A capability with permission Uπ, range [b,e) and address a grants:

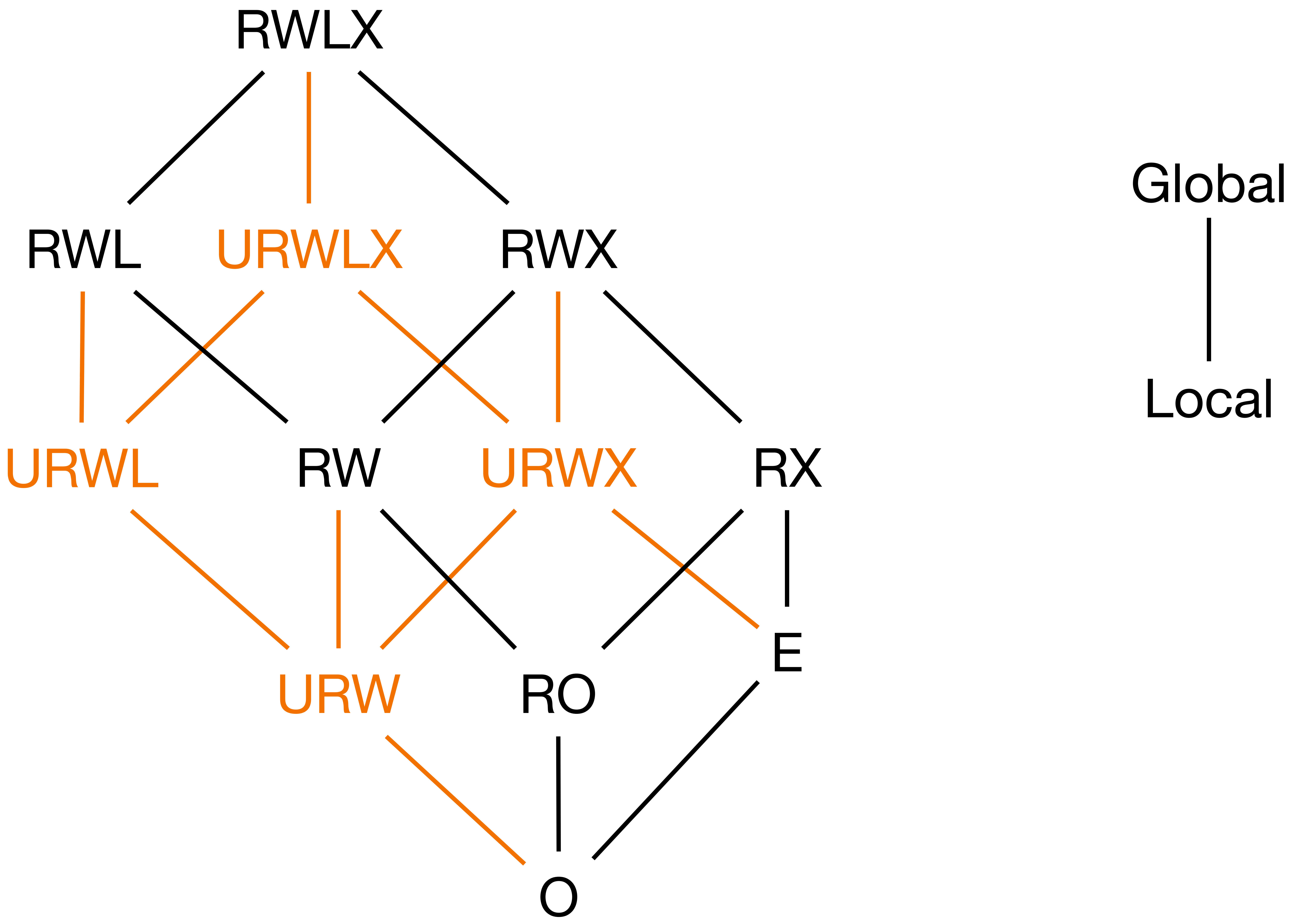
- authority π for the range [b,a)
- no authority over [a+1,e)
- write only authority over a

Its address a is incremented once written to

A Revisited Lattice of Permissions



A Revisited Lattice of Permissions



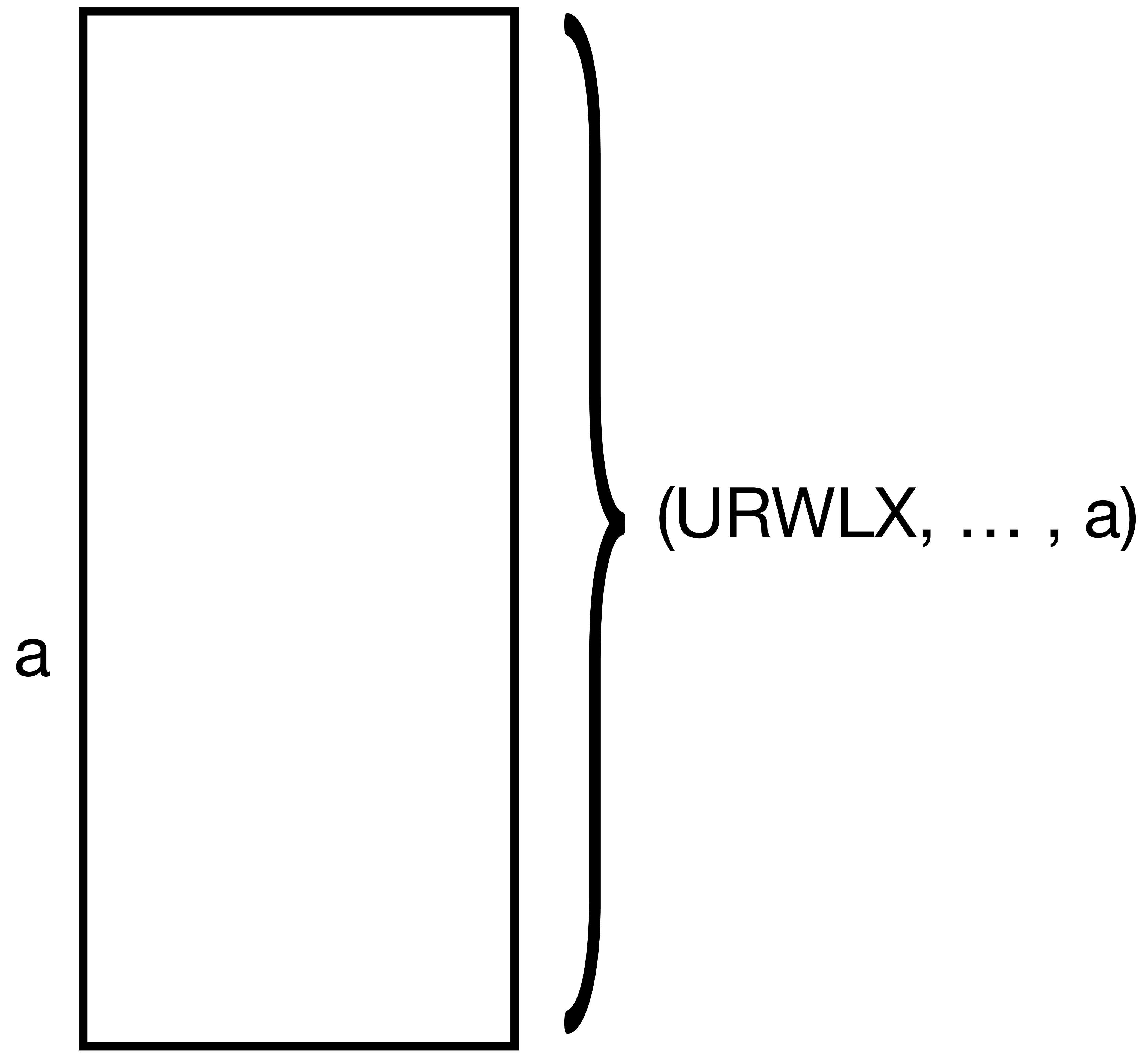
Capability Revocation: Caller

Disallow dangling stack pointers

- A dangling stack pointer is a capability that points to a “younger” stack frame (remember, in our stack higher means younger)
- Take advantage of the implicit lifetime information of on-stack capability addresses
- Higher address = younger stack
- Older stack capabilities cannot store younger stack capabilities

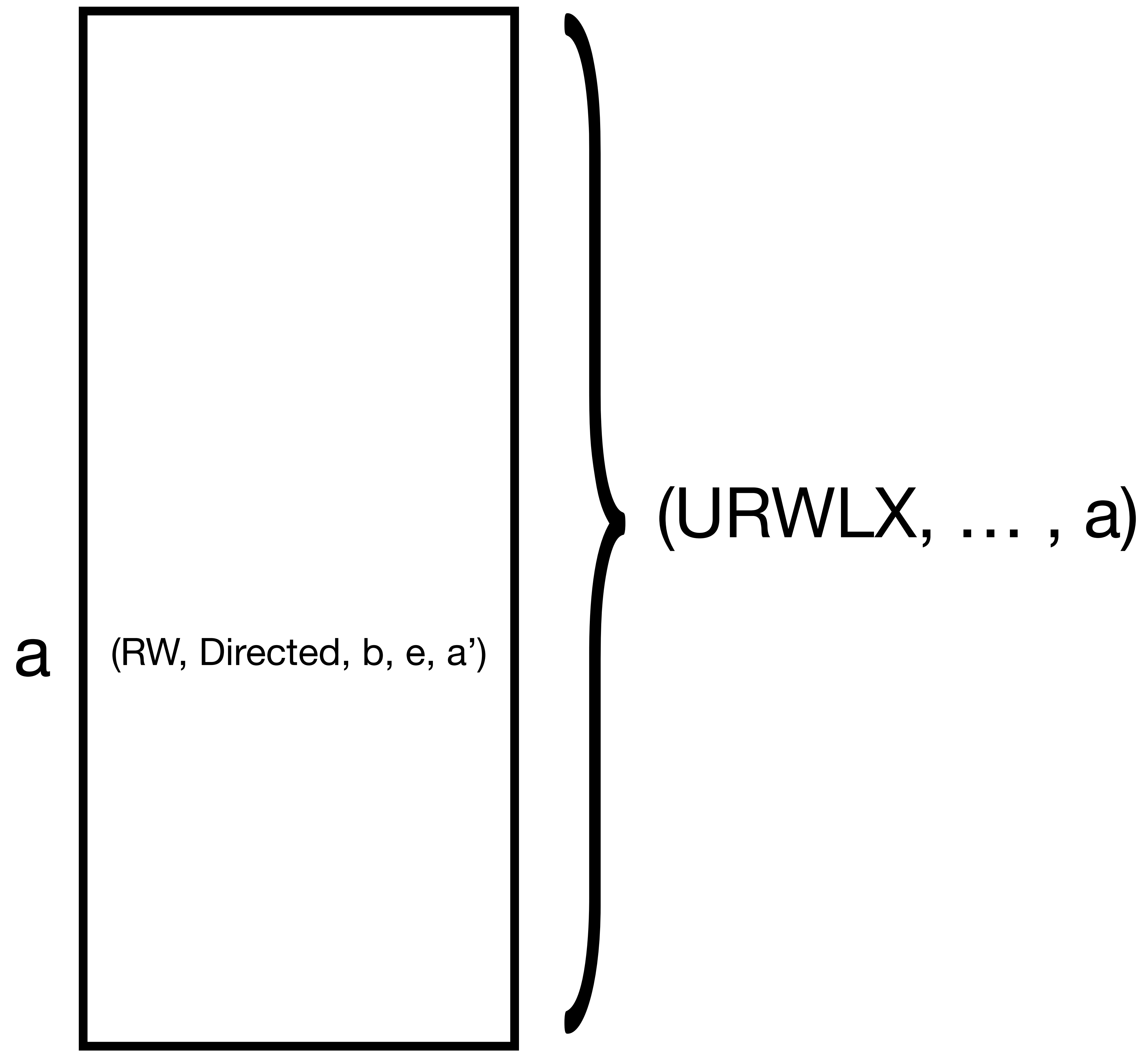
Directed Capabilities

Disallow dangling stack pointers



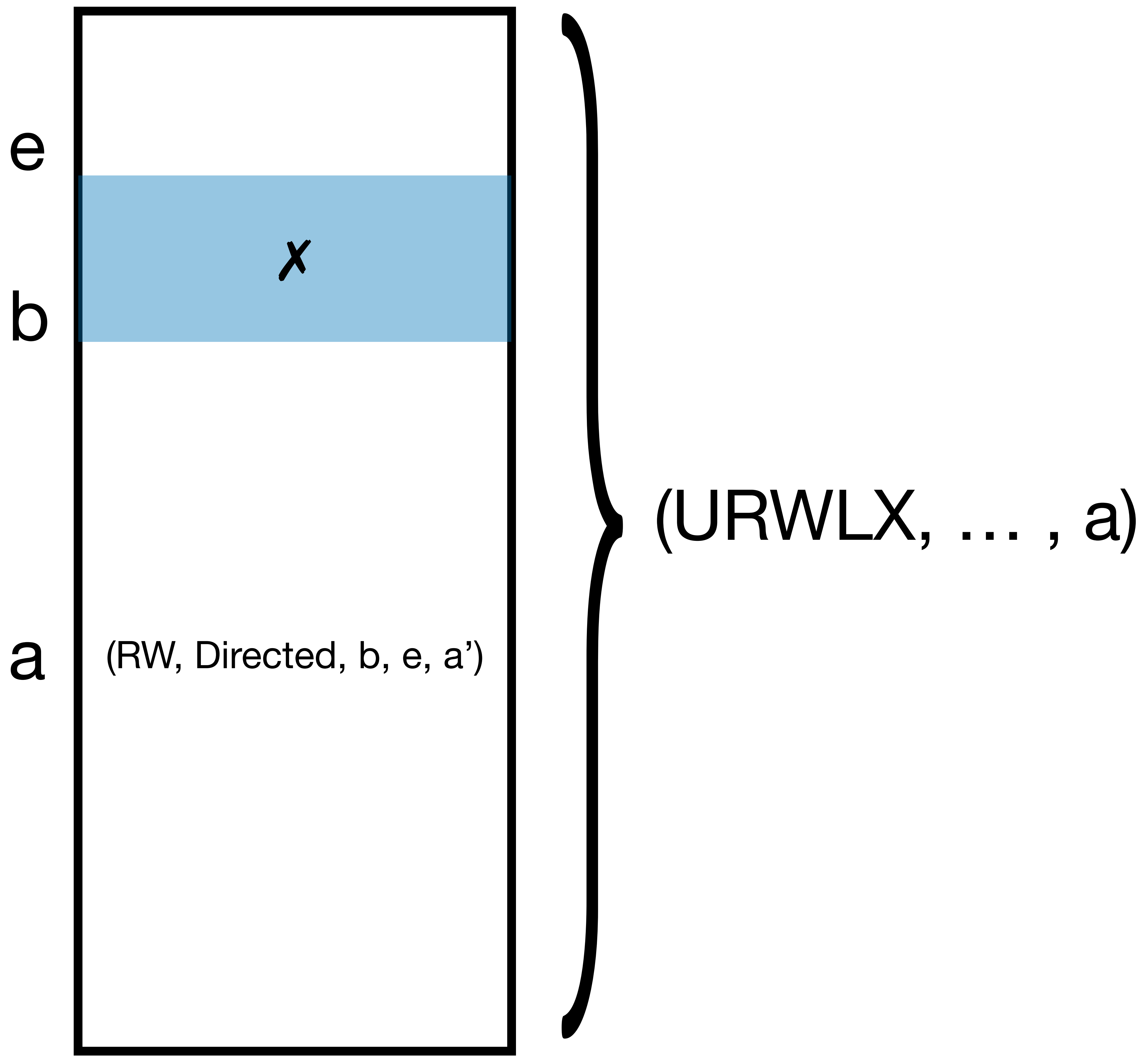
Directed Capabilities

Disallow dangling stack pointers



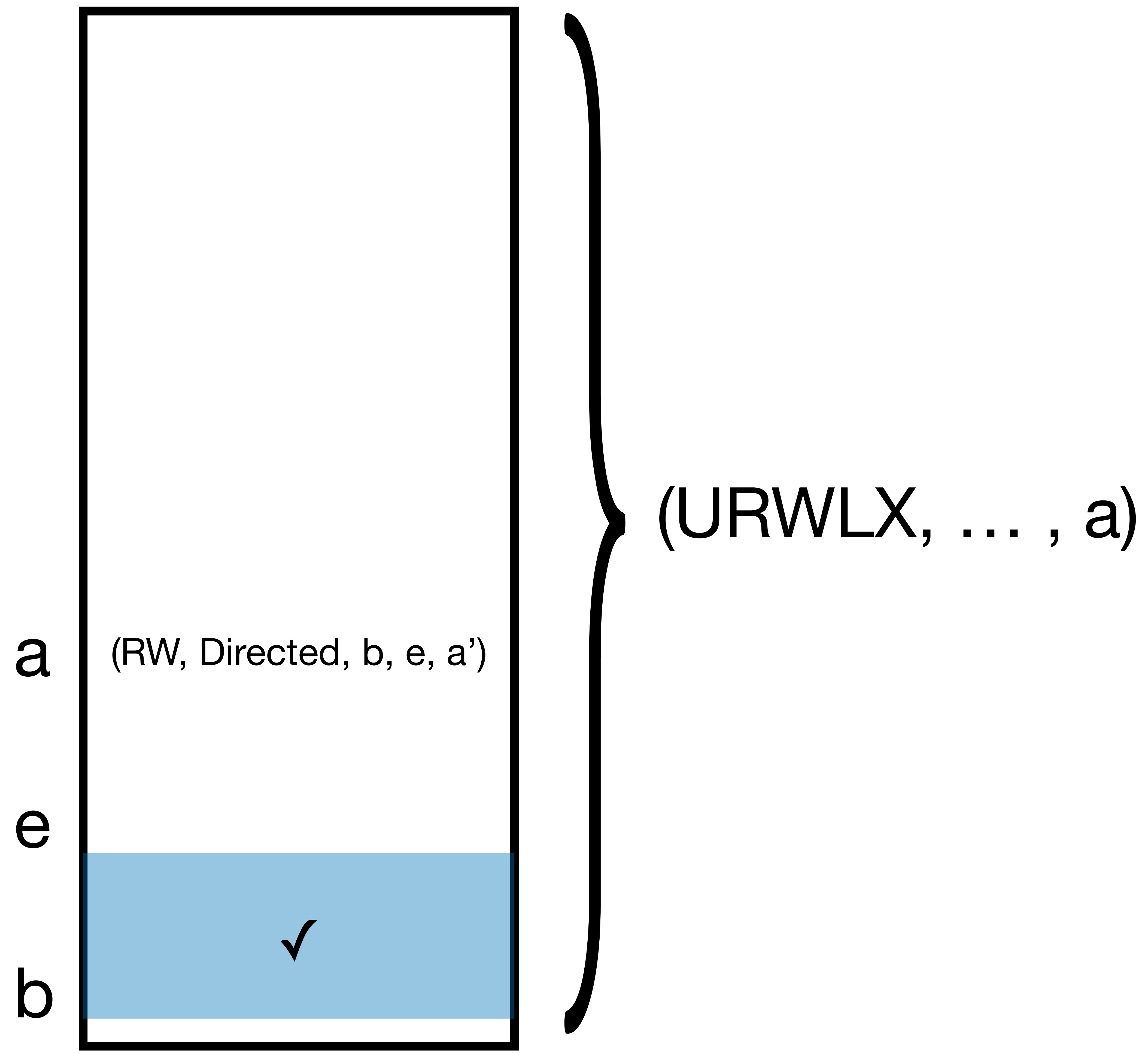
Directed Capabilities

Disallow dangling stack pointers



Directed Capabilities

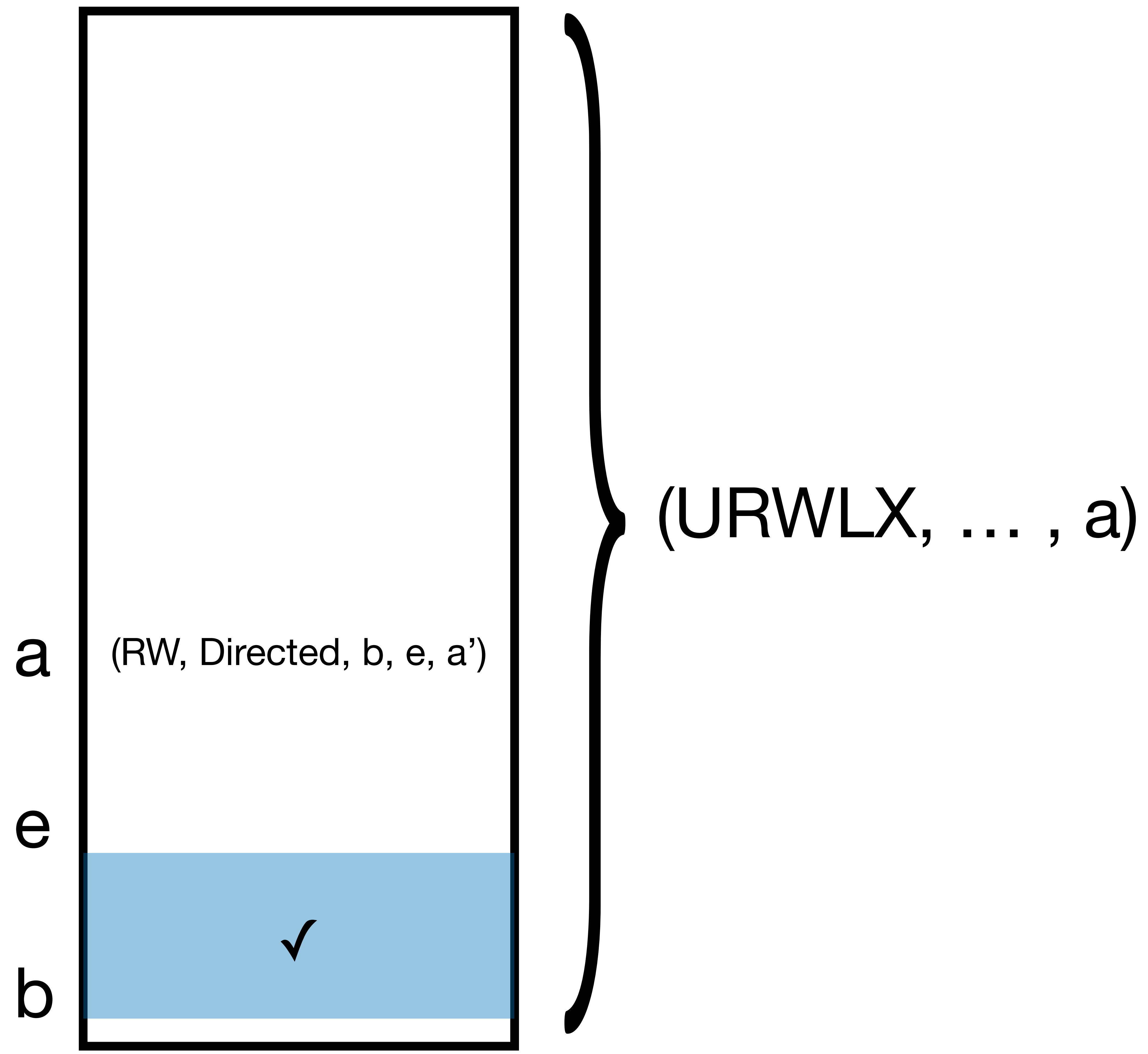
Disallow dangling stack pointers



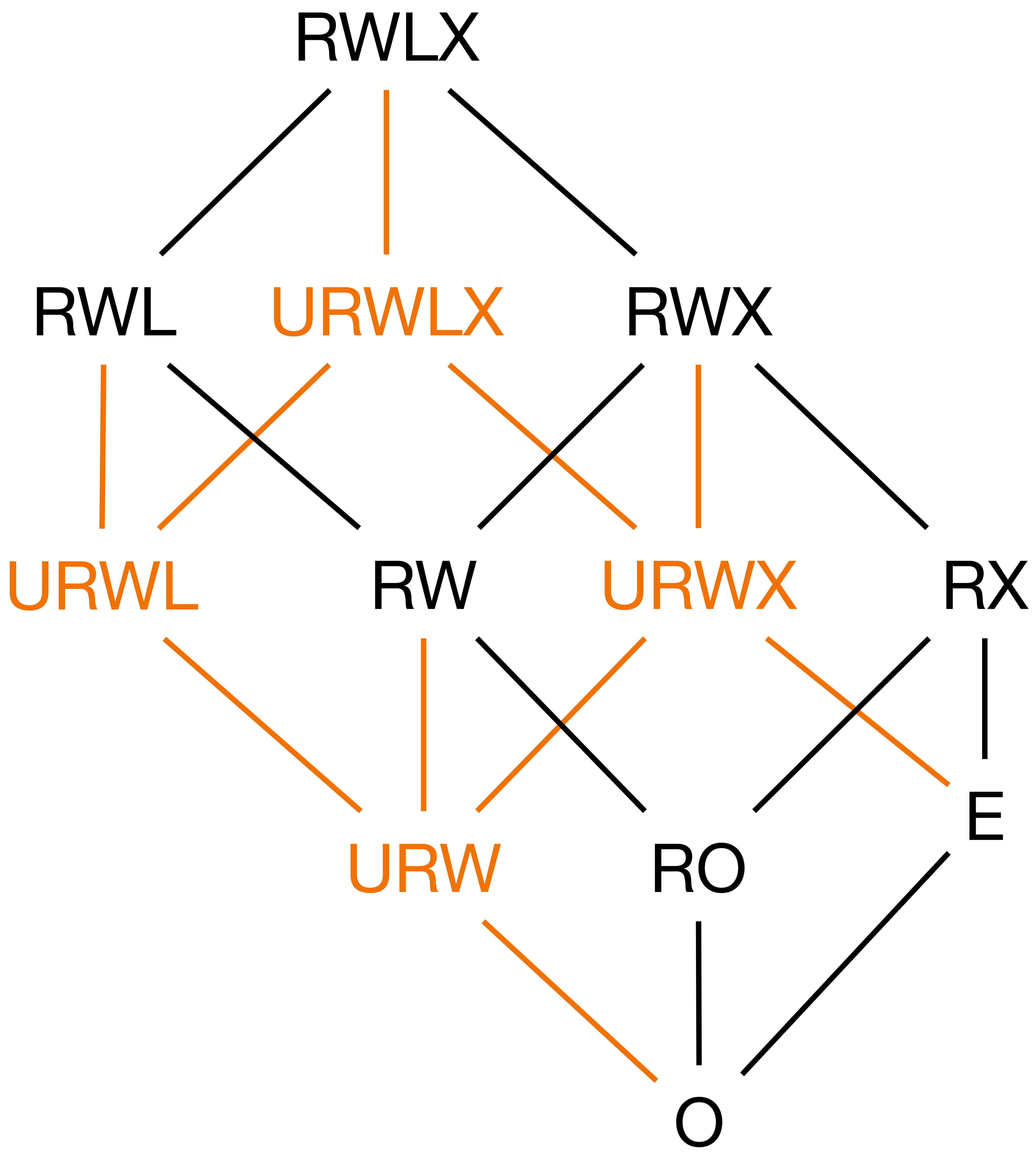
Directed Capabilities

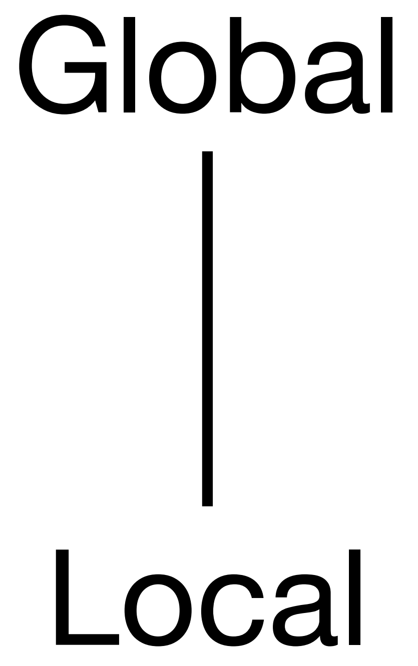
Disallow dangling stack pointers

- One additional locality bit
- One more dynamic arithmetic check

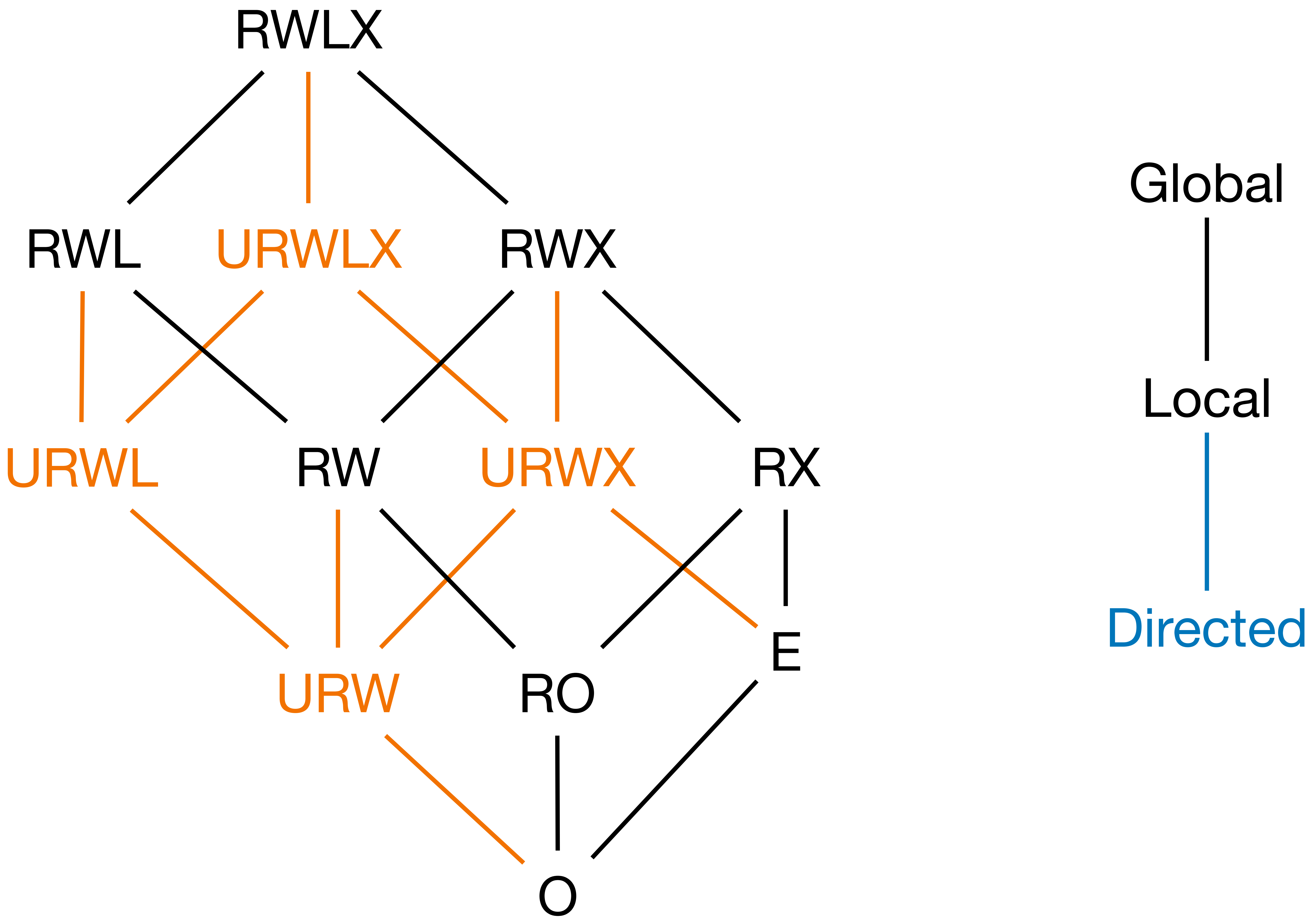


A Re-Revisited Lattice of Permissions



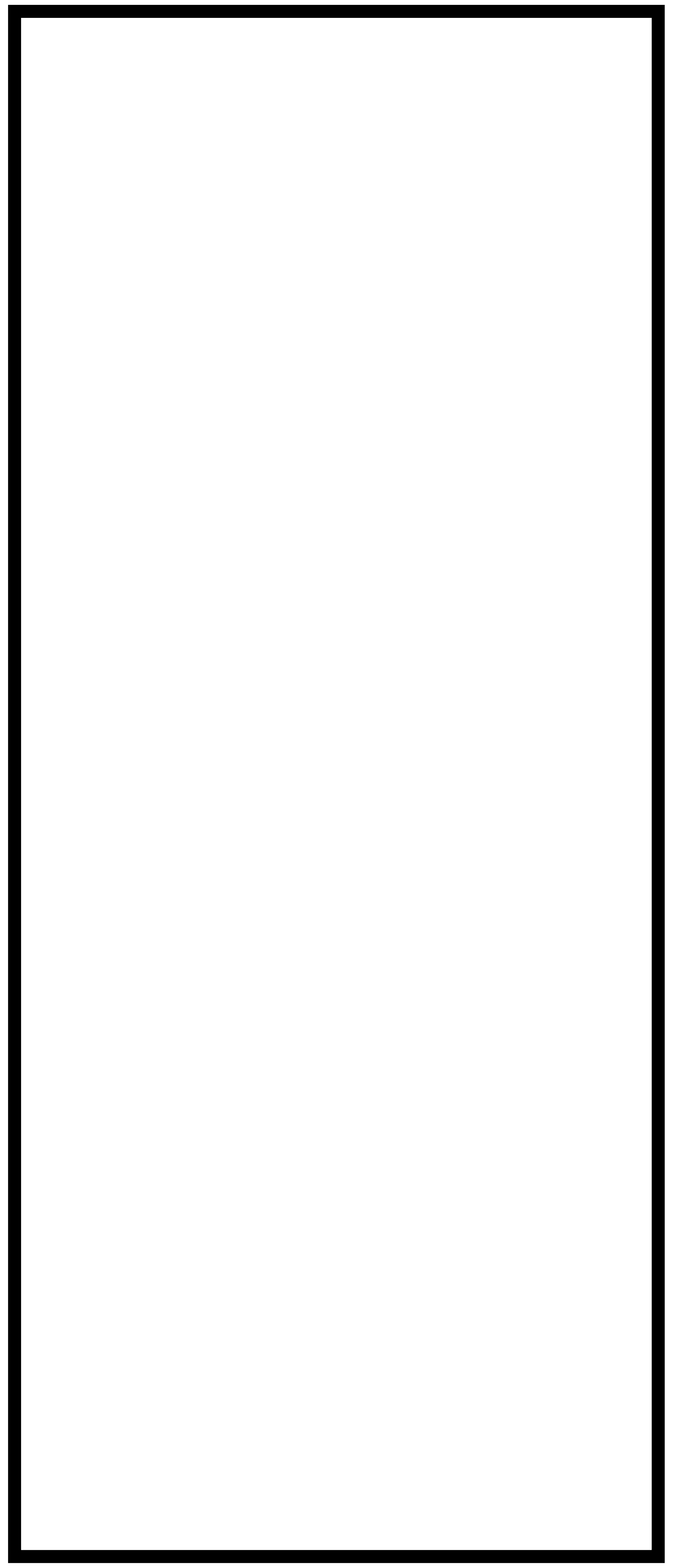


A Re-Revisited Lattice of Permissions



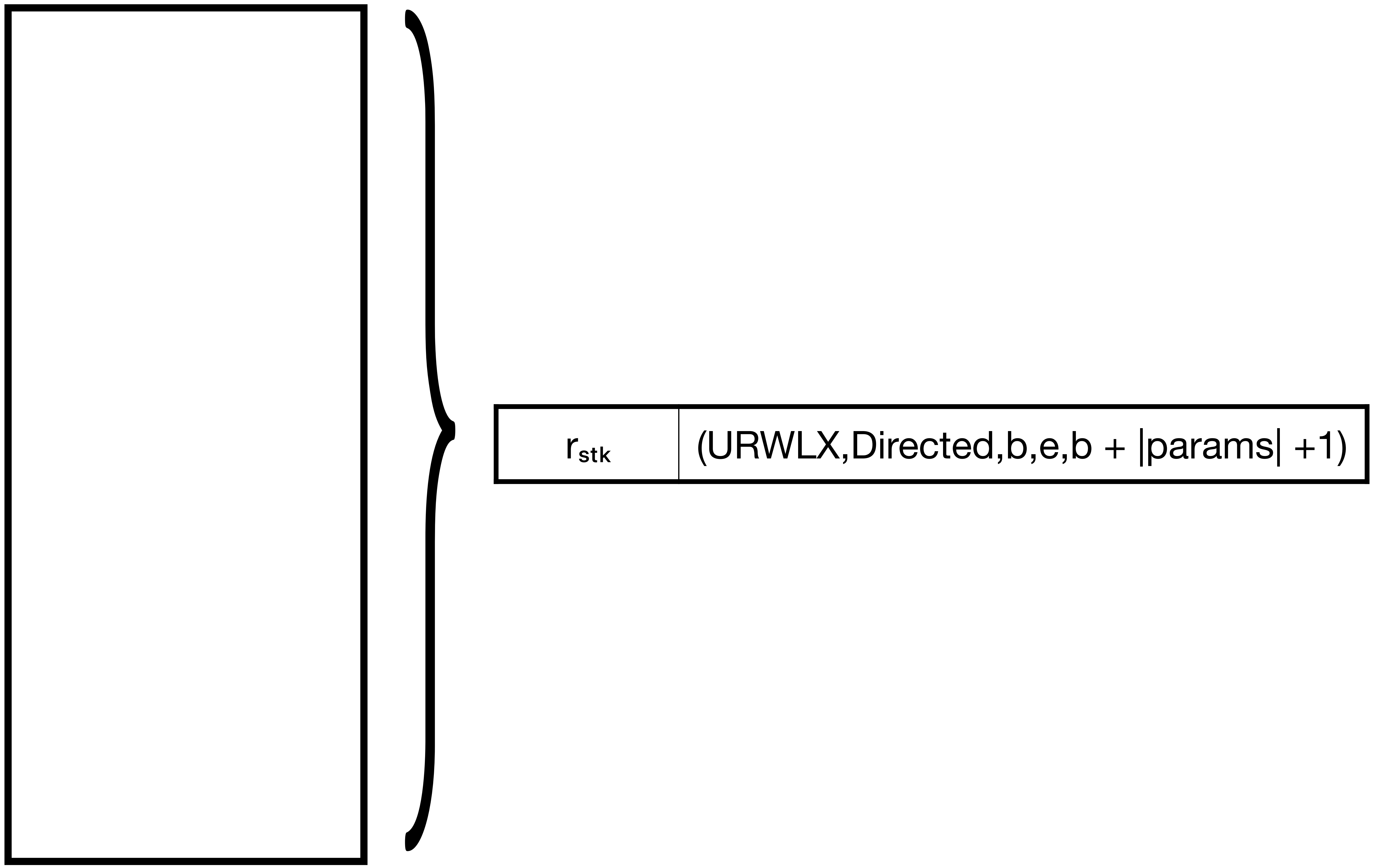
Secure Calling Convention

When called by an adversary



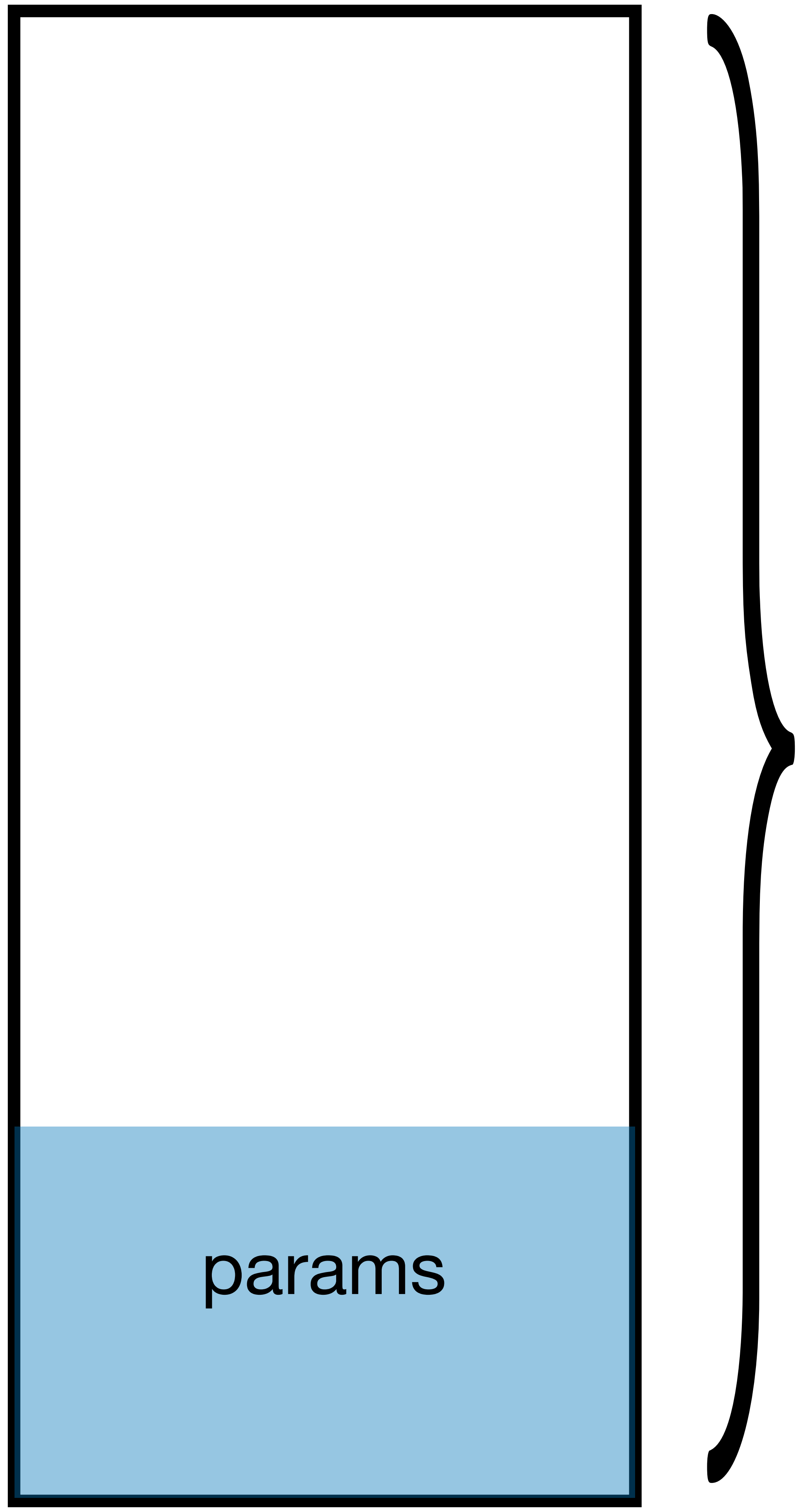
Secure Calling Convention

When called by an adversary



Secure Calling Convention

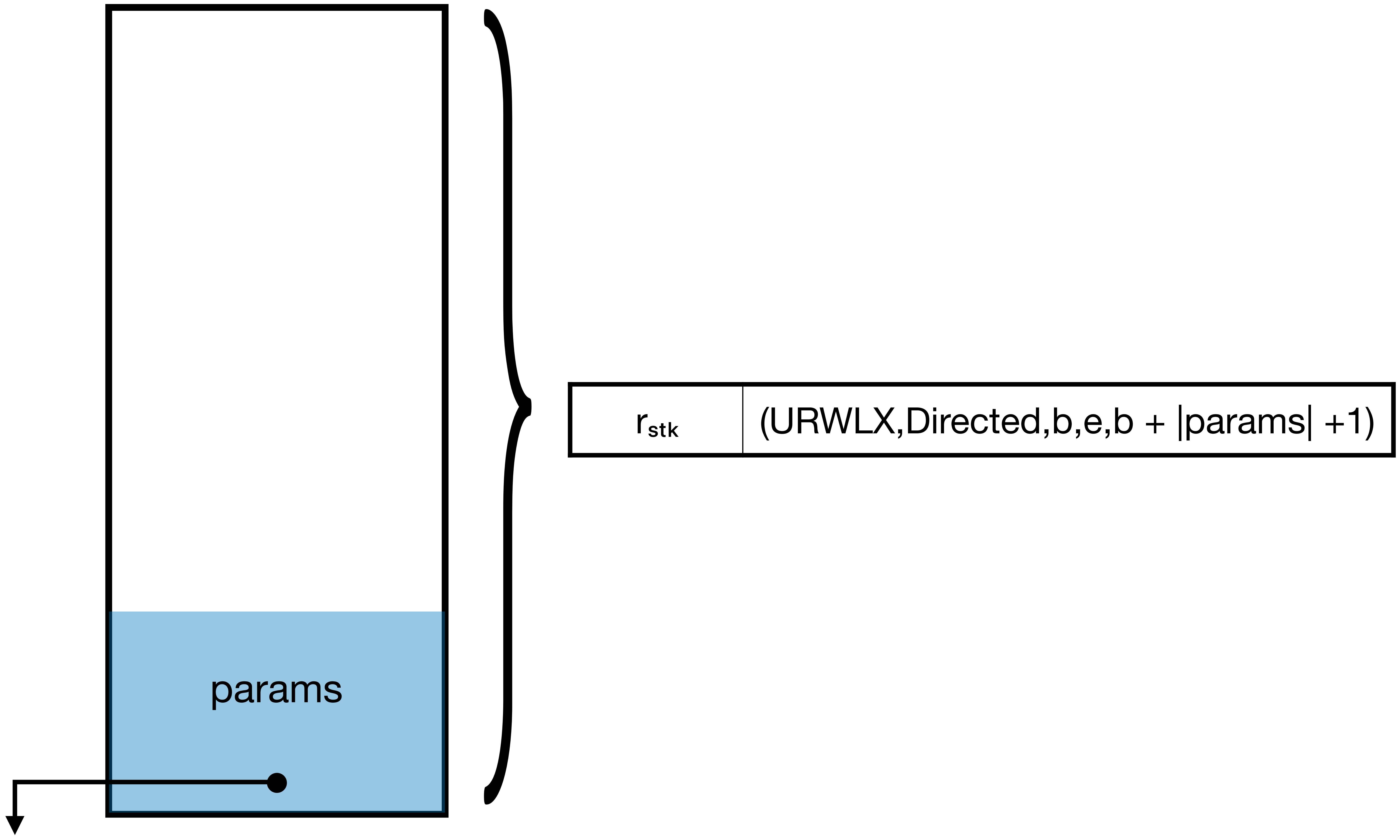
When called by an adversary



r_{stk}	(URWLX,Directed,b,e,b + |params| +1)

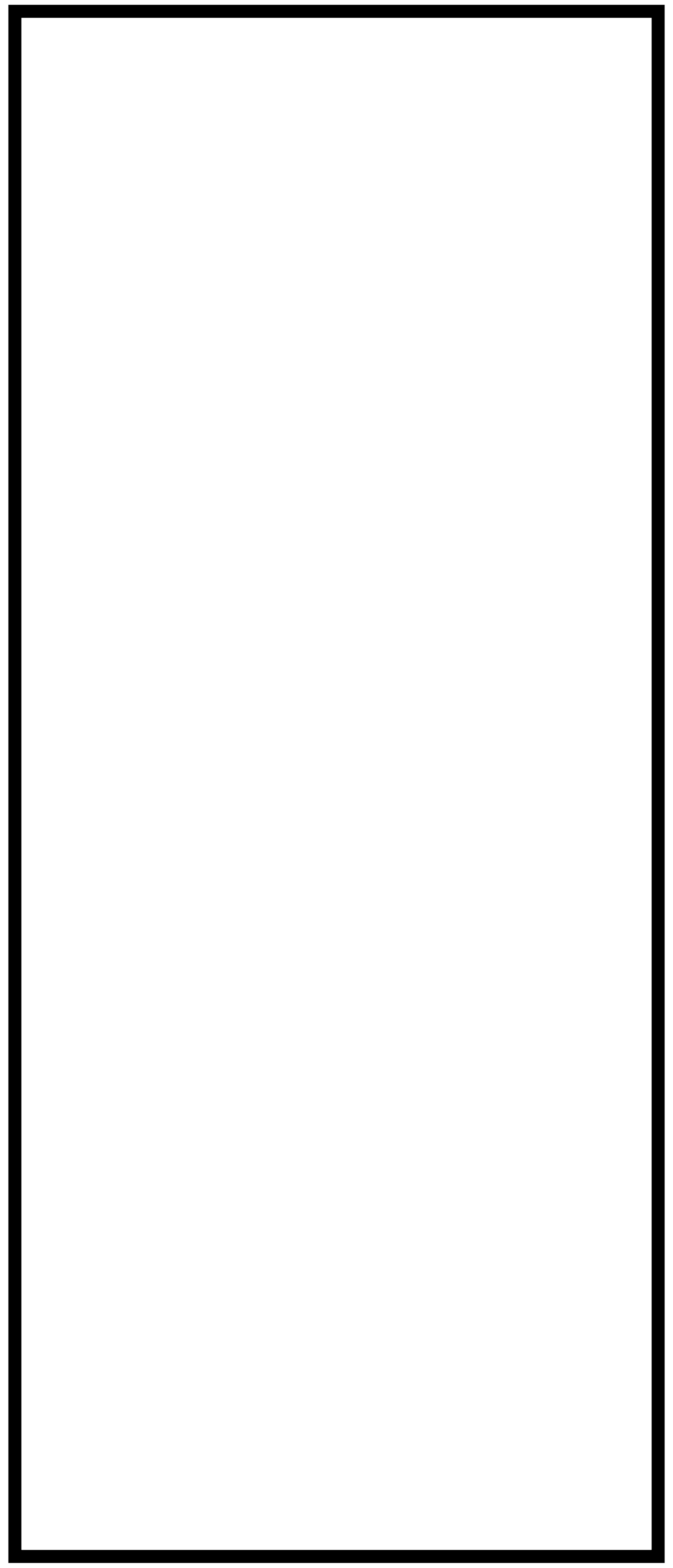
Secure Calling Convention

When called by an adversary



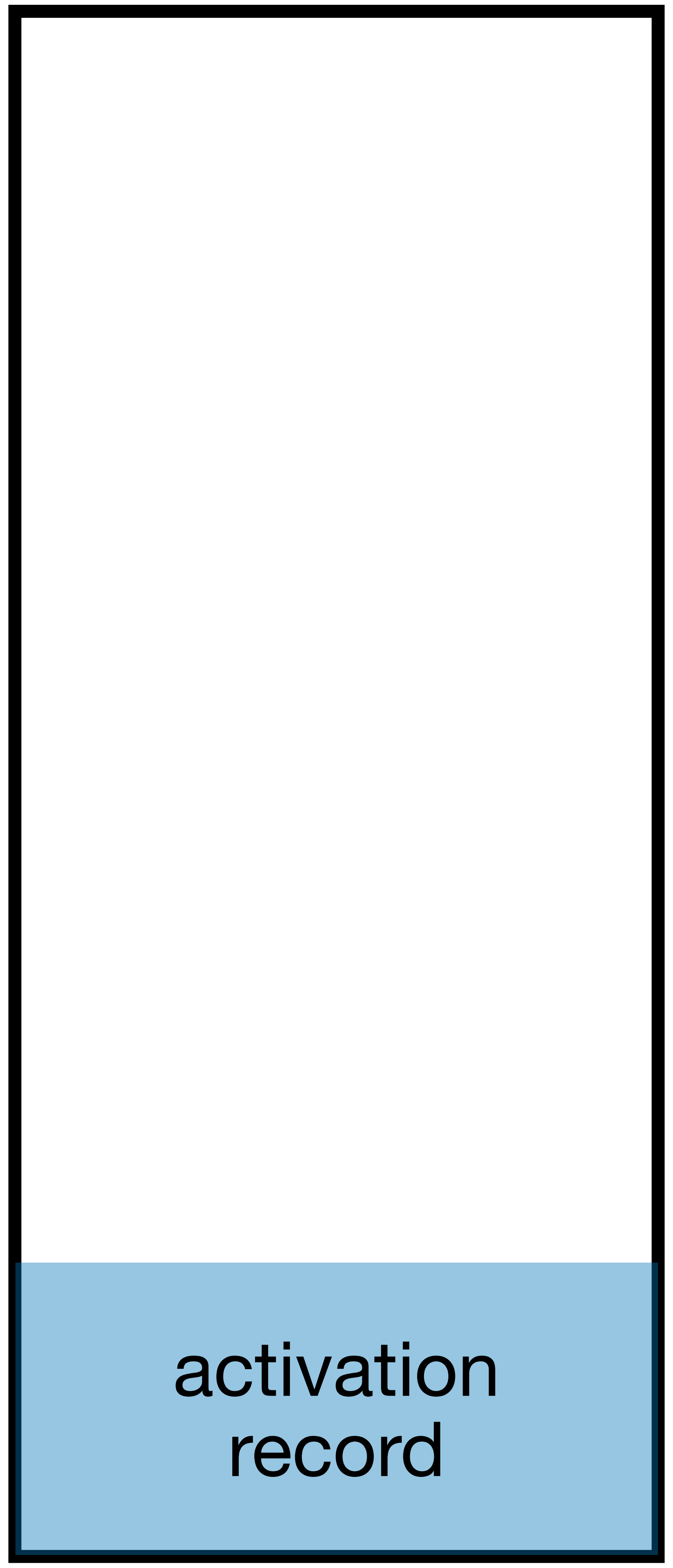
Secure Calling Convention

Before calling an adversary



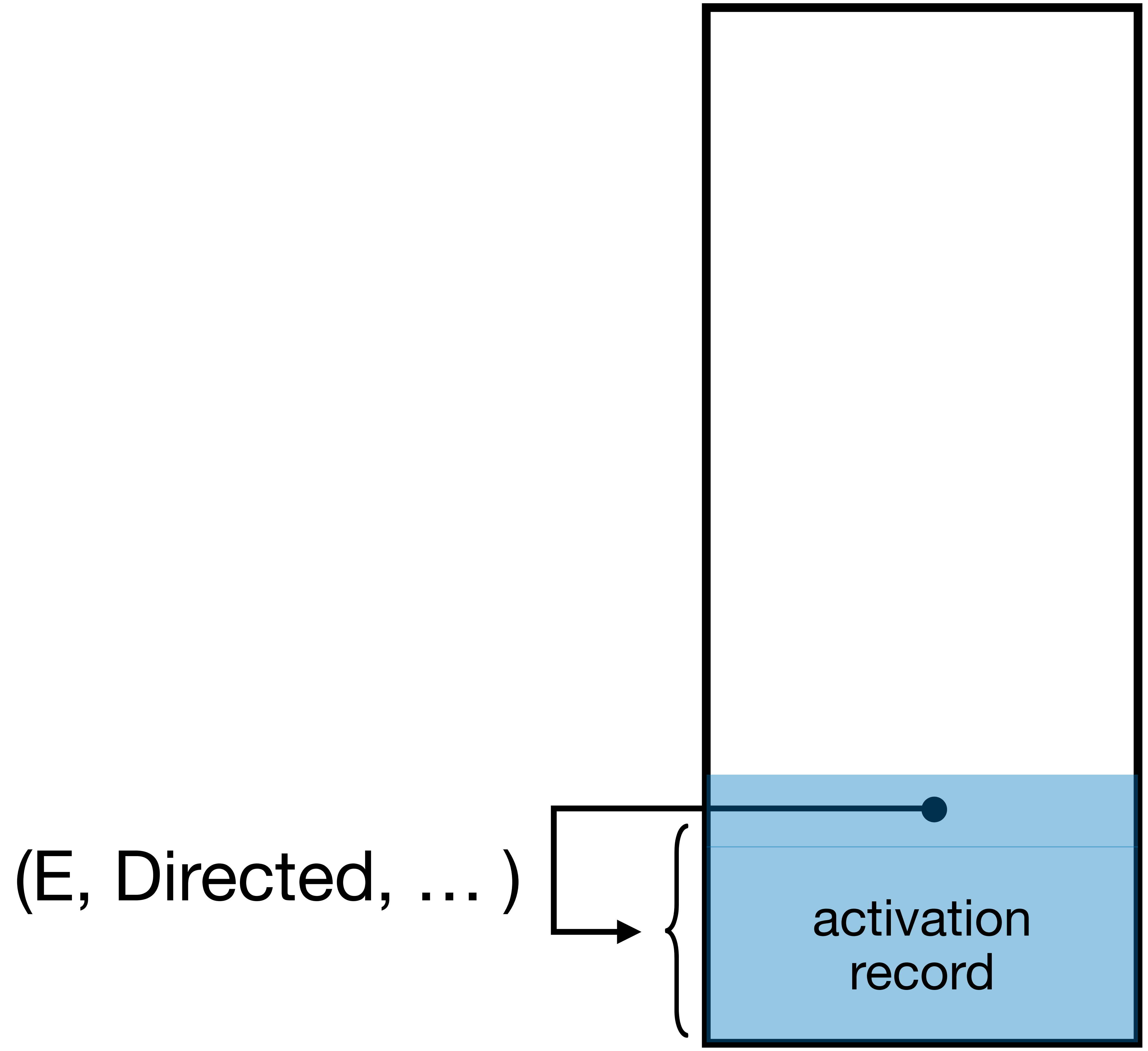
Secure Calling Convention

Before calling an adversary



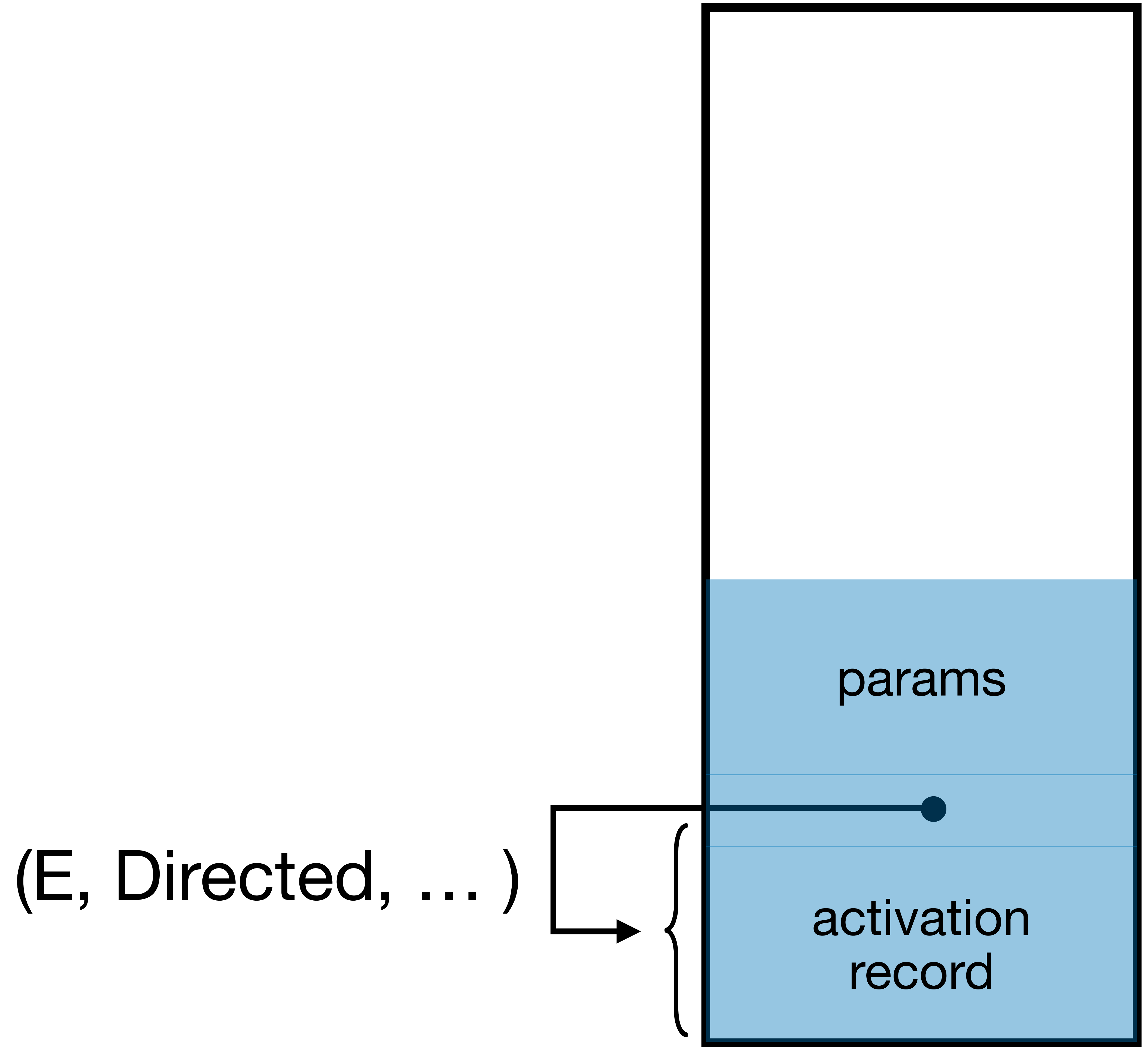
Secure Calling Convention

Before calling an adversary



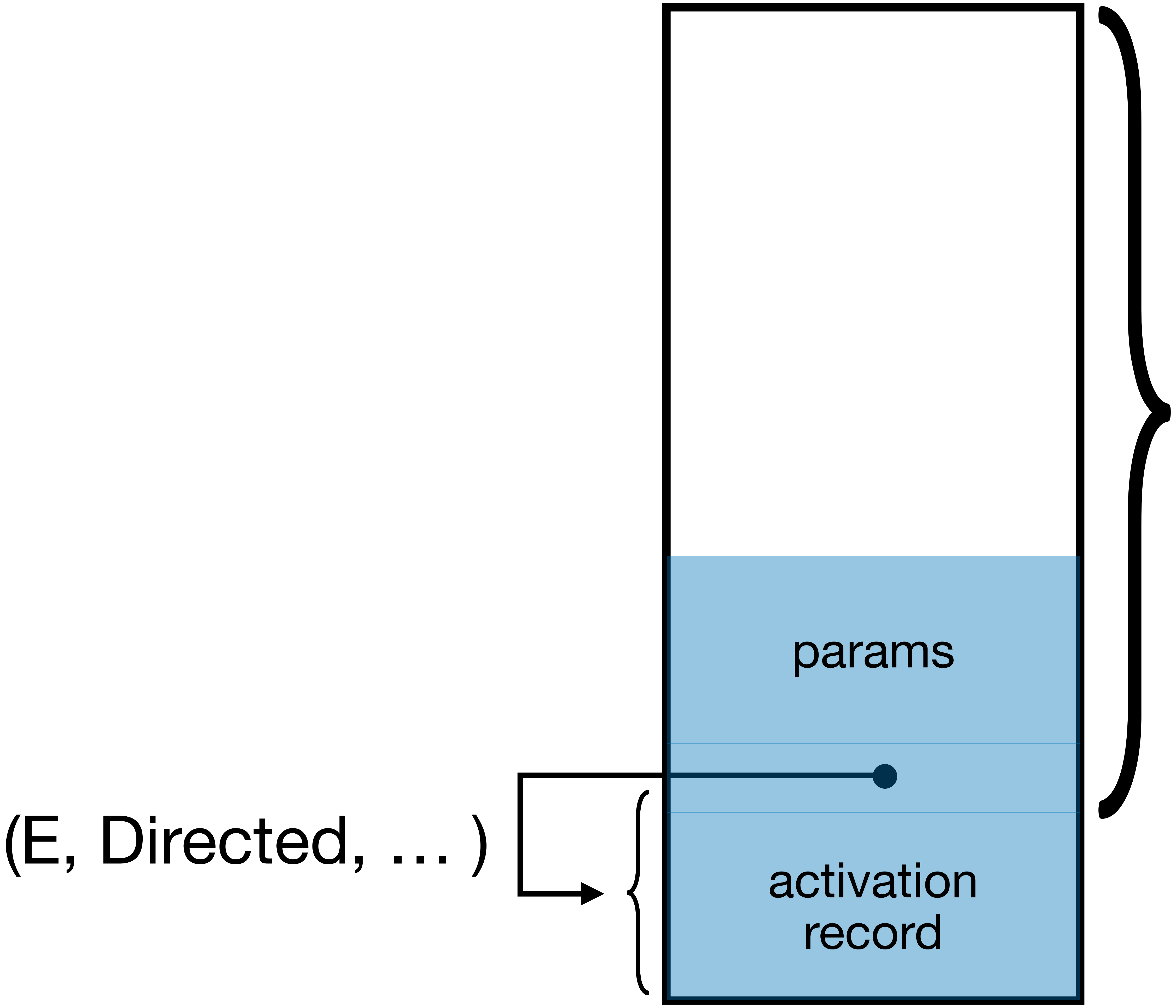
Secure Calling Convention

Before calling an adversary



Secure Calling Convention

Before calling an adversary



r_{stk}	(URWLX,Directed,b + |act|,e,b + |act| + |params| + 1)
…	0

Secure Calling Convention

Before returning to an adversary

Secure Calling Convention

Before returning to an adversary

Clear all general purpose registers

How can we trust that this works?

Summary of the Mechanized Verification

using the calling convention

- Unary logical relation
  - Used to prove the robust safety of examples that interact with unknown code (Awkward example, dangling stack pointer example, stack object example)
- Binary logical relation
  - Used to prove the contextual equivalence of examples that interact with unknown code
- Full-abstraction against an overlay semantics (proved in Coq)

Summary of the Mechanized Verification

using the calling convention

- Unary logical relation
  - Used to prove the robust safety of examples that interact with unknown code (Awkward example, dangling stack pointer example, stack object example)
- Binary logical relation
  - Used to prove the contextual equivalence of examples that interact with unknown code
- Full-abstraction against an overlay semantics (proved in Coq)

Dangling Stack Pointer Example

```
g1: malloc 1 r2
    store r2 2
    closure creation around
    r2 and f1
```

```
f1: prepstack r_stk
    loadU r0 r_stk -1
    push r_env
    load r_env r_env
    assert r_env 2
    rclear RegName\{PC,r0}
    jmp r0
```

Dangling Stack Pointer Example

```
g1: malloc 1 r2
    store r2 2
    closure creation around
    r2 and f1
```

```
f1: prepstack r_stk
    loadU r0 r_stk -1
    push r_env
    load r_env r_env
    assert r_env 2
    rclear RegName\{PC,r0}
    jmp r0
```

prepare the stack: check its size, check that the parameters can be read,…

Dangling Stack Pointer Example

```
g1: malloc 1 r2
    store r2 2
    closure creation around
    r2 and f1
```

```
f1: prepstack r_stk
    loadU r0 r_stk -1
    push r_env
    load r_env r_env
    assert r_env 2
    rclear RegName\{PC,r0}
    jmp r0
```

load the return pointer provided by caller

Dangling Stack Pointer Example

```
g1: malloc 1 r2
      store r2 2
      closure creation around
      r2 and f1
```

```
f1: prepstack r_stk
     loadU r0 r_stk -1
     push r_env
     load r_env r_env
     assert r_env 2
     rclear RegName\{PC,r0}
     jmp r0
```

purposefully try to leak the enclosed local state

Dangling Stack Pointer Example

```
g1: malloc 1 r2
    store r2 2
    closure creation around
    r2 and f1
```

```
f1: prepstack r_stk
    loadU r0 r_stk -1
    push r_env
    load r_env r_env
    assert r_env 2
    rclear RegName\{PC,r0}
    jmp r0
```

load the enclosed local state and assert it is still 2

Dangling Stack Pointer Example

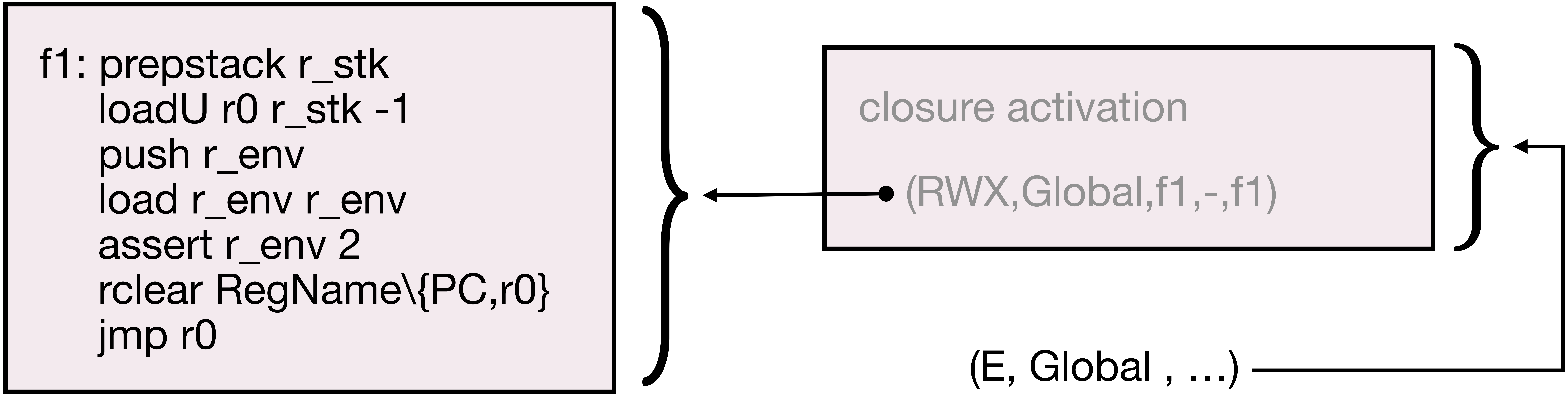
```
g1: malloc 1 r2
    store r2 2
    closure creation around
    r2 and f1
```

```
f1: prepstack r_stk
    loadU r0 r_stk -1
    push r_env
    load r_env r_env
    assert r_env 2
    rclear RegName\{PC,r0}
    jmp r0
```

apply the calling convention: clear registers and return

A Program Logic to Reason about Known Code

$$\left\{ \begin{array}{l} \mathsf{pc} \mapsto (\mathrm{RX}, \mathrm{GLOBAL}, b, e, b) \\ * \; [b, e) \mapsto f_1 \\ * \; r_{\mathrm{stk}} \mapsto (\mathrm{URWLX}, \mathrm{DIRECTED}, bstk, estk, astk) \\ * \; \cdots \end{array} \right\} \text{ Executing } \left\{ \begin{array}{l} \mathsf{pc} \mapsto - \\ * \; r_{\mathrm{stk}} \mapsto - \\ * \; \cdots \end{array} \right\}$$

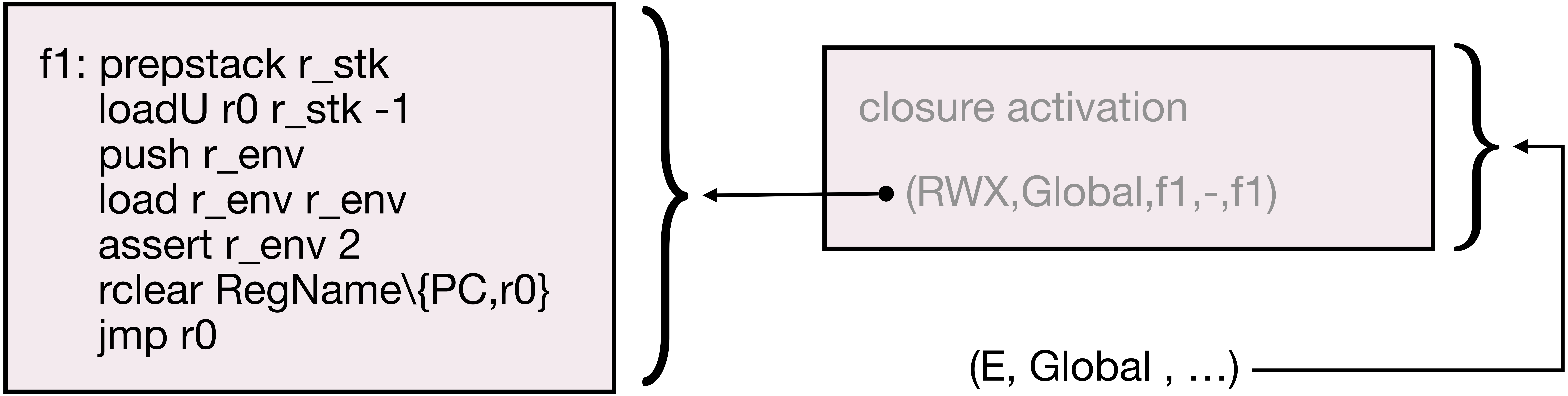


Defining “safe to share”

$\mathcal{V}(w)$ w is safe to share

$\mathcal{E}(w)$ w is safe to execute

$$\text{FTLR} : \mathcal{V}(w) \rightarrow \mathcal{E}(w)$$



Unary Logical Relation - a Simplified Attempt

Defining “safe to execute” and “safe to share”

Unary Logical Relation - a Simplified Attempt

Defining “safe to execute” and “safe to share”

$$\mathcal{E}(w) \triangleq \forall reg, \{\mathsf{pc} \mapsto w * [r_1, ..., r_{31}] \mapsto reg * \mathcal{R}(reg) * \cdots\} \text{ Executable } \{\cdots\}$$

Unary Logical Relation - a Simplified Attempt

Defining “safe to execute” and “safe to share”

$$\mathcal{E}(w) \triangleq \forall reg, \{\mathsf{pc} \mapsto w * [r_1, ..., r_{31}] \mapsto reg * \mathcal{R}(reg) * \cdots\} \text{ Executable } \{\cdots\}$$

$$\mathcal{R}(reg) \triangleq \mathcal{V}(reg[0]) * \cdots * \mathcal{V}(reg[30])$$

Unary Logical Relation - a Simplified Attempt

Defining “safe to execute” and “safe to share”

$$\mathcal{E}(w) \triangleq \forall reg, \{\mathsf{pc} \mapsto w * [r_1, ..., r_{31}] \mapsto reg * \mathcal{R}(reg) * \cdots\} \text{ Executable } \{\cdots\}$$

$$\mathcal{R}(reg) \triangleq \mathcal{V}(reg[0]) * \cdots * \mathcal{V}(reg[30])$$

$$\mathcal{V}(\text{E}, \text{GLOBAL}, \cdots) \triangleq \Box \triangleright \mathcal{E}(\text{RX}, \text{GLOBAL}, \cdots)$$

Unary Logical Relation - a Simplified Attempt

Defining “safe to execute” and “safe to share”

$$\mathcal{E}(w) \triangleq \forall reg, \{\mathsf{pc} \mapsto w * [r_1, ..., r_{31}] \mapsto reg * \mathcal{R}(reg) * \cdots\} \text{ Executable } \{\cdots\}$$

$$\mathcal{R}(reg) \triangleq \mathcal{V}(reg[0]) * \cdots * \mathcal{V}(reg[30])$$

$$\mathcal{V}(\mathrm{E}, \mathrm{GLOBAL}, \cdots) \triangleq \Box \triangleright \mathcal{E}(\mathrm{RX}, \mathrm{GLOBAL}, \cdots)$$

$$\mathcal{V}(z), \mathcal{V}(\mathrm{O}, -, -, -, -) \triangleq \top$$

Unary Logical Relation - a Simplified Attempt

Defining "safe to execute" and "safe to share"

$$\mathcal{E}(w) \triangleq \forall reg, \{\mathsf{pc} \mapsto w * [r_1, ..., r_{31}] \mapsto reg * \mathcal{R}(reg) * \cdots\} \text{ Executable } \{\cdots\}$$

$$\mathcal{R}(reg) \triangleq \mathcal{V}(reg[0]) * \cdots * \mathcal{V}(reg[30])$$

$$\mathcal{V}(\text{E}, \text{GLOBAL}, \cdots) \triangleq \Box \triangleright \mathcal{E}(\text{RX}, \text{GLOBAL}, \cdots)$$

$$\mathcal{V}(z), \mathcal{V}(\text{O}, -, -, -, -) \triangleq \top$$

$$\mathcal{V}(\text{RWX}, \text{GLOBAL}, b, e, -) \triangleq \mathop{\Large\ast}_{a \in [b,e)} \boxed{\exists w, a \mapsto w * \mathcal{V}(w)}^{\mathcal{N}.a}$$

Unary Logical Relation - a Simplified Attempt

Defining “safe to execute” and “safe to share”

$$\mathcal{E}(w) \triangleq \forall reg, \{\mathsf{pc} \mapsto w * [r_1, \ldots, r_{31}] \mapsto reg * \mathcal{R}(reg) * \cdots\} \text{ Executable } \{\cdots\}$$

$$\mathcal{R}(reg) \triangleq \mathcal{V}(reg[0]) * \cdots * \mathcal{V}(reg[30])$$

$$\mathcal{V}(\text{E}, \text{GLOBAL}, \cdots) \triangleq \Box \triangleright \mathcal{E}(\text{RX}, \text{GLOBAL}, \cdots)$$

$$\mathcal{V}(z), \mathcal{V}(\text{O}, -, -, -, -) \triangleq \top$$

$$\mathcal{V}(\text{RWX}, \text{GLOBAL}, b, e, -) \triangleq \mathop{\Huge *}_{a \in [b,e)} \boxed{\exists w, a \mapsto w * \mathcal{V}(w)}^{\mathcal{N}.a}$$

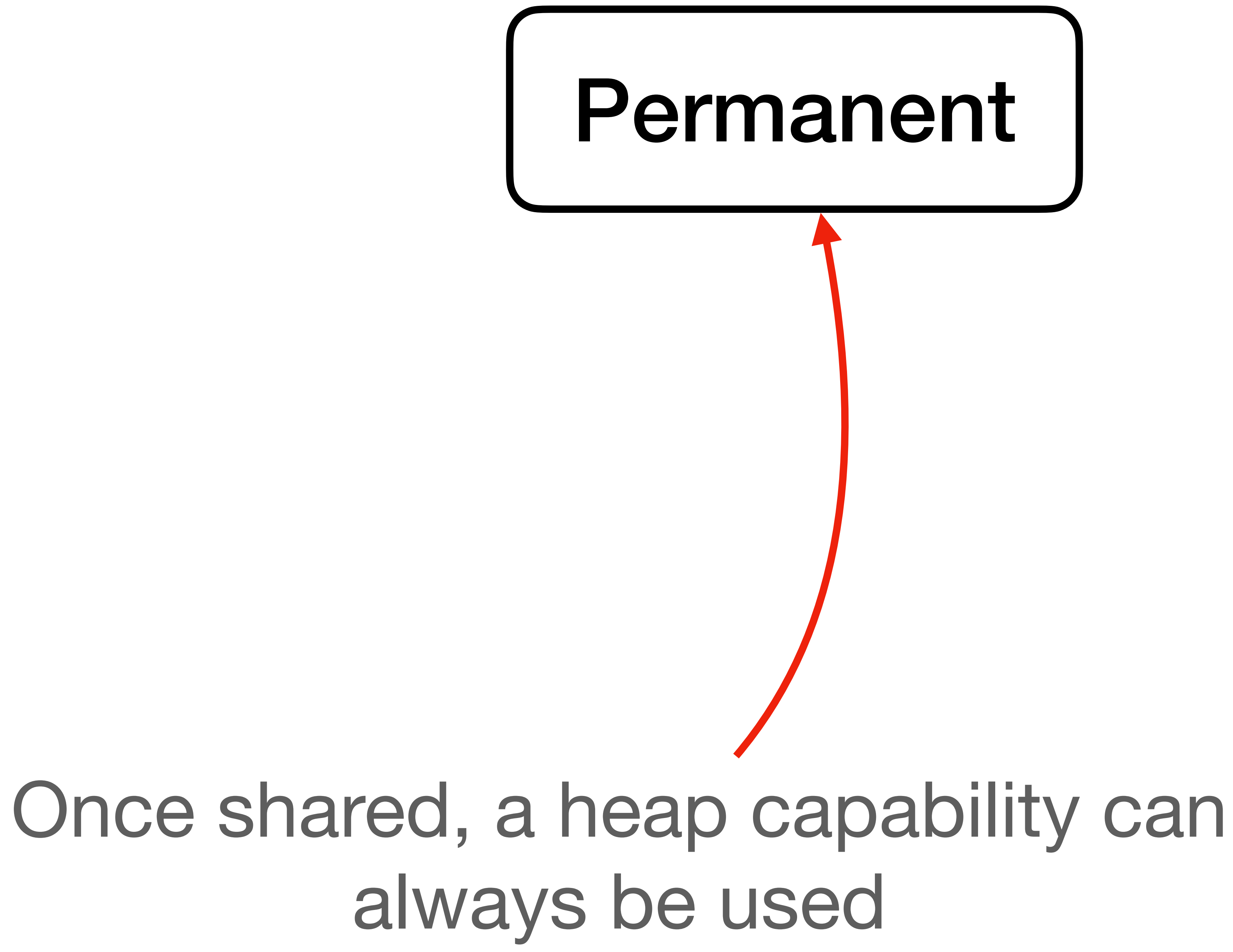
No formal distinction between global and directed capabilities

Modelling Lifetime Behaviour of Stack and Heap

What different states can the stack and heap be in?

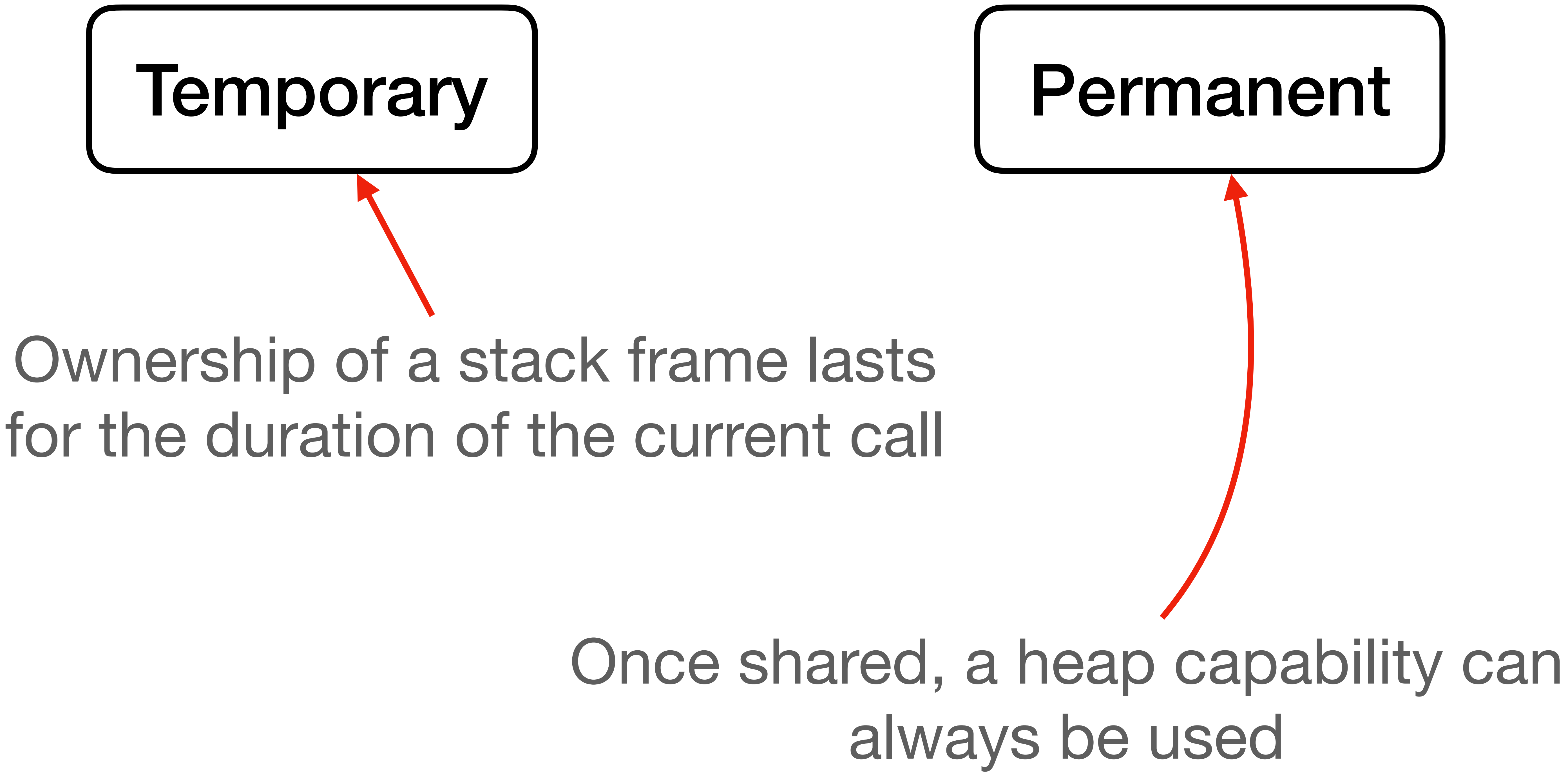
Modelling Lifetime Behaviour of Stack and Heap

What different states can the stack and heap be in?



Modelling Lifetime Behaviour of Stack and Heap

What different states can the stack and heap be in?



Modelling Lifetime Behaviour of Stack and Heap

What different states can the stack and heap be in?

A stack frame can be:

Temporary

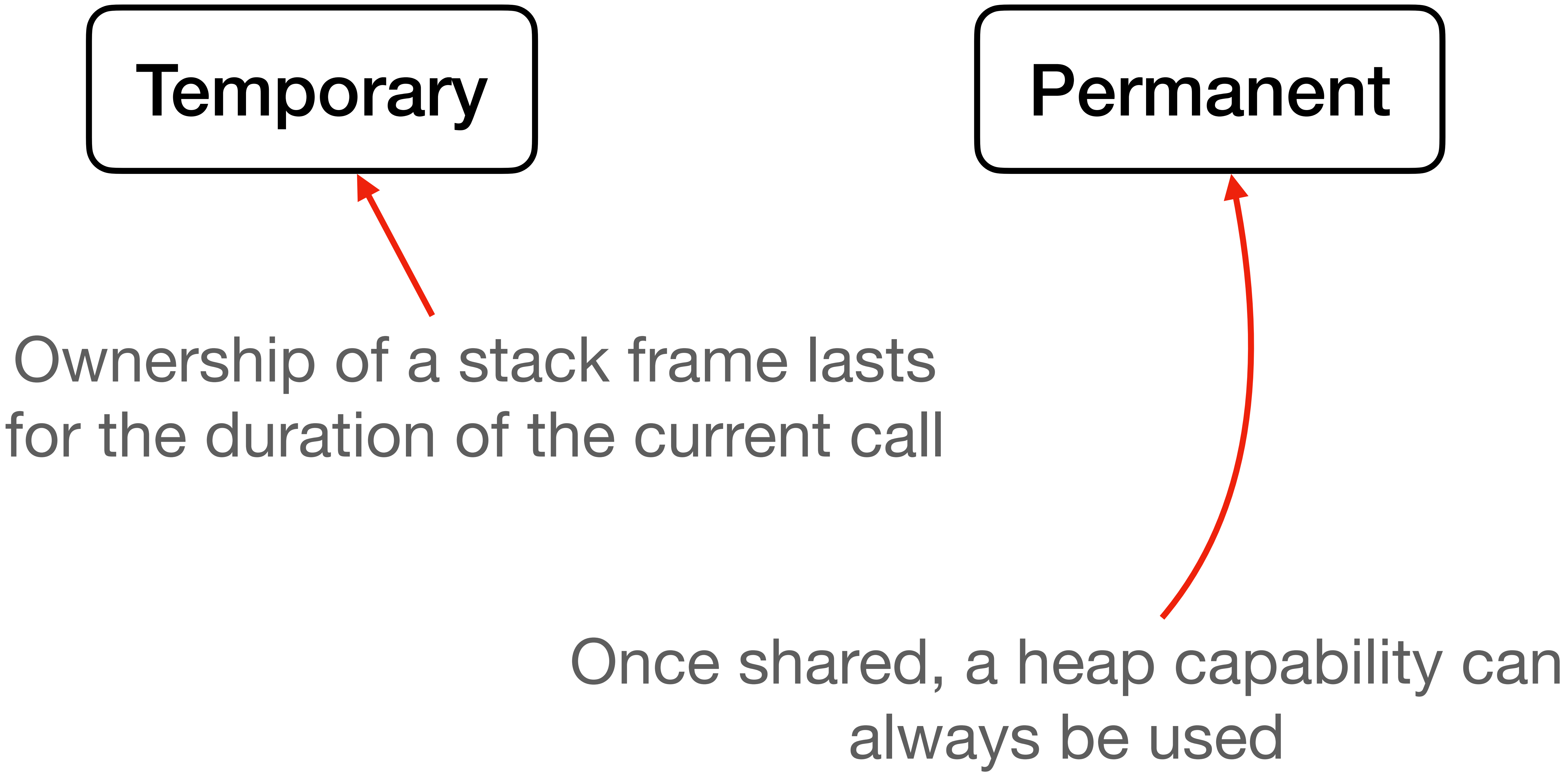
Permanent

Ownership of a stack frame lasts for the duration of the current call

Once shared, a heap capability can always be used

Modelling Lifetime Behaviour of Stack and Heap

What different states can the stack and heap be in?



A stack frame can be:

- Live

Modelling Lifetime Behaviour of Stack and Heap

What different states can the stack and heap be in?

Temporary

Ownership of a stack frame lasts for the duration of the current call

Permanent

Once shared, a heap capability can always be used

A stack frame can be:

- Live
- Dead/popped

Modelling Lifetime Behaviour of Stack and Heap

What different states can the stack and heap be in?

Temporary

Permanent

Ownership of a stack frame lasts for the duration of the current call

Once shared, a heap capability can always be used

A stack frame can be:

- Live
- Dead/popped
- Frozen

Modelling Lifetime Behaviour of Stack and Heap

What different states can the stack and heap be in?

Uninitialized(w) | **Temporary** | **Permanent**

Ownership of a stack frame lasts for the duration of the current call

Once shared, a heap capability can always be used

A stack frame can be:

- Live
- Dead/popped
- Frozen

Modelling Lifetime Behaviour of Stack and Heap

What different states can the stack and heap be in?

Frozen(m)

Uninitialized(w)

Temporary

Permanent

A stack frame can be:

- Live
- Dead/popped
- Frozen

Ownership of a stack frame lasts for the duration of the current call

Once shared, a heap capability can always be used

Modelling Lifetime Behaviour of Stack and Heap

What different states can the stack and heap be in?

Frozen(m)

Uninitialized(w)

Temporary

Permanent

Ownership of a stack frame lasts for the duration of the current call

Once shared, a heap capability can always be used

A stack frame can be:

- Live
- Dead/popped
- Frozen

Modelling Lifetime Behaviour of Stack and Heap

What different states can the stack and heap be in?

Frozen(m)

Uninitialized(w)

Temporary

Permanent

Ownership of a stack frame lasts for the duration of the current call

Once shared, a heap capability can always be used

A stack frame can be:

- Live
- Dead/popped
- Frozen

Modelling Lifetime Behaviour of Stack and Heap

What different states can the stack and heap be in?

Frozen(m)

Uninitialized(w)

Temporary

Permanent

Ownership of a stack frame lasts for the duration of the current call

Once shared, a heap capability can always be used

A stack frame can be:

- Live
- Dead/popped
- Frozen

Modelling Lifetime Behaviour of Stack and Heap

What different states can the stack and heap be in?

Frozen(m)

Uninitialized(w)

Temporary

Permanent

Ownership of a stack frame lasts for the duration of the current call

Once shared, a heap capability can always be used

A stack frame can be:

- Live
- Dead/popped
- Frozen

Modelling Lifetime Behaviour of Stack and Heap

What different states can the stack and heap be in?

Frozen(m)

Uninitialized(w)

Temporary

Permanent

Ownership of a stack frame lasts for the duration of the current call

Once shared, a heap capability can always be used

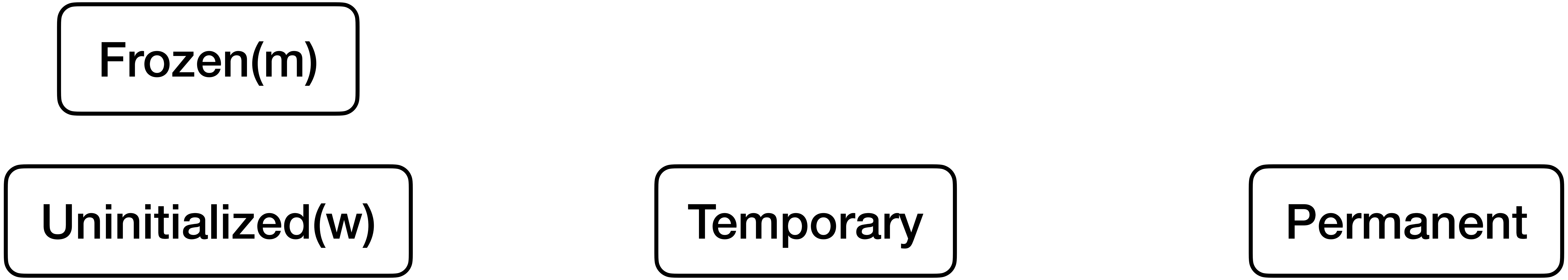
A stack frame can be:

- Live
- Dead/popped
- Frozen

standard states

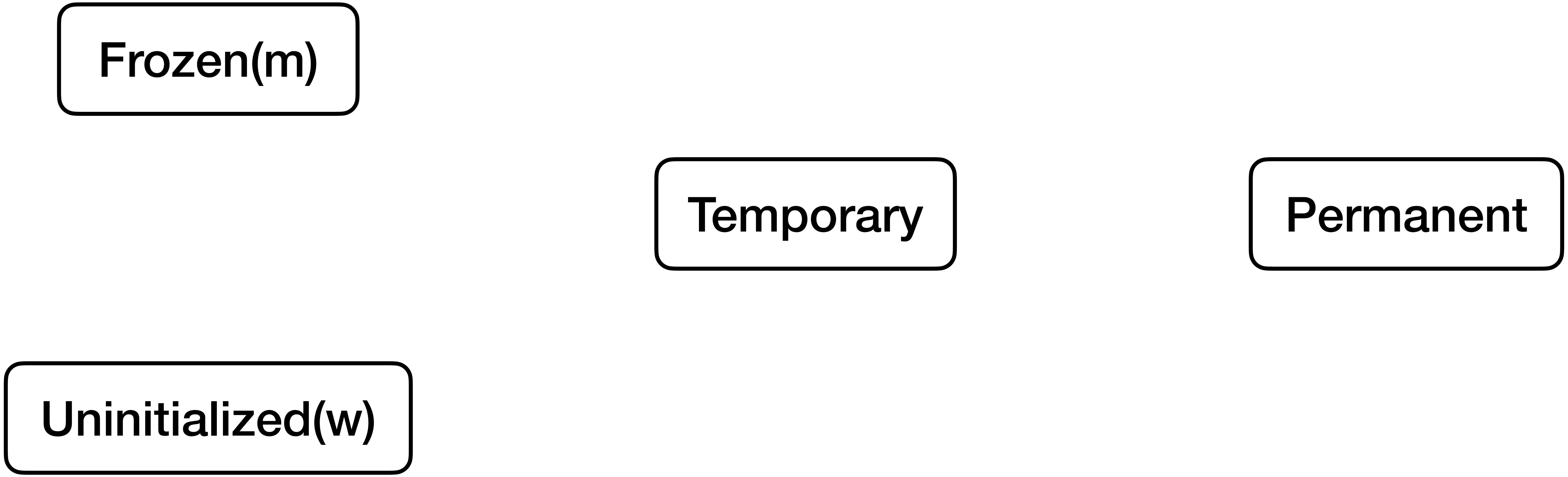
Modelling Lifetime Behaviour of Stack and Heap

Which transitions are safe to observe by the caller, and by the callee?



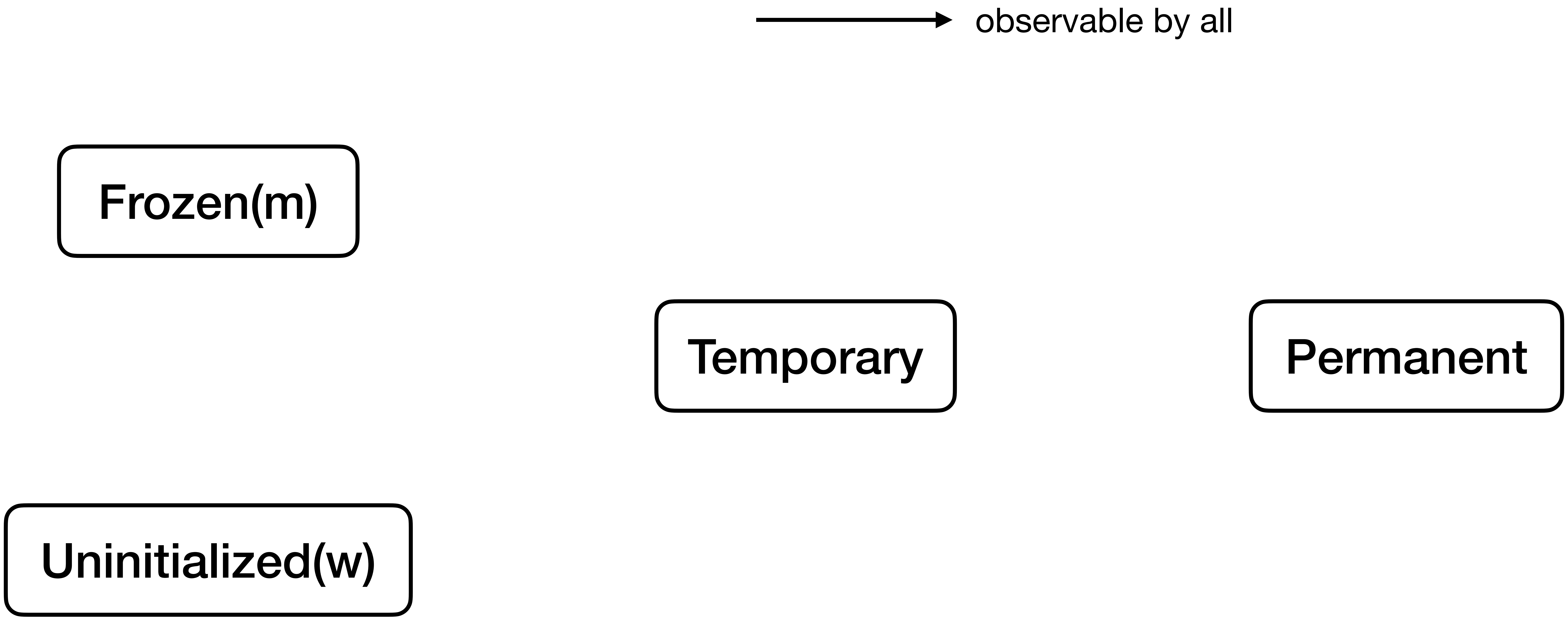
Modelling Lifetime Behaviour of Stack and Heap

Which transitions are safe to observe by the caller, and by the callee?



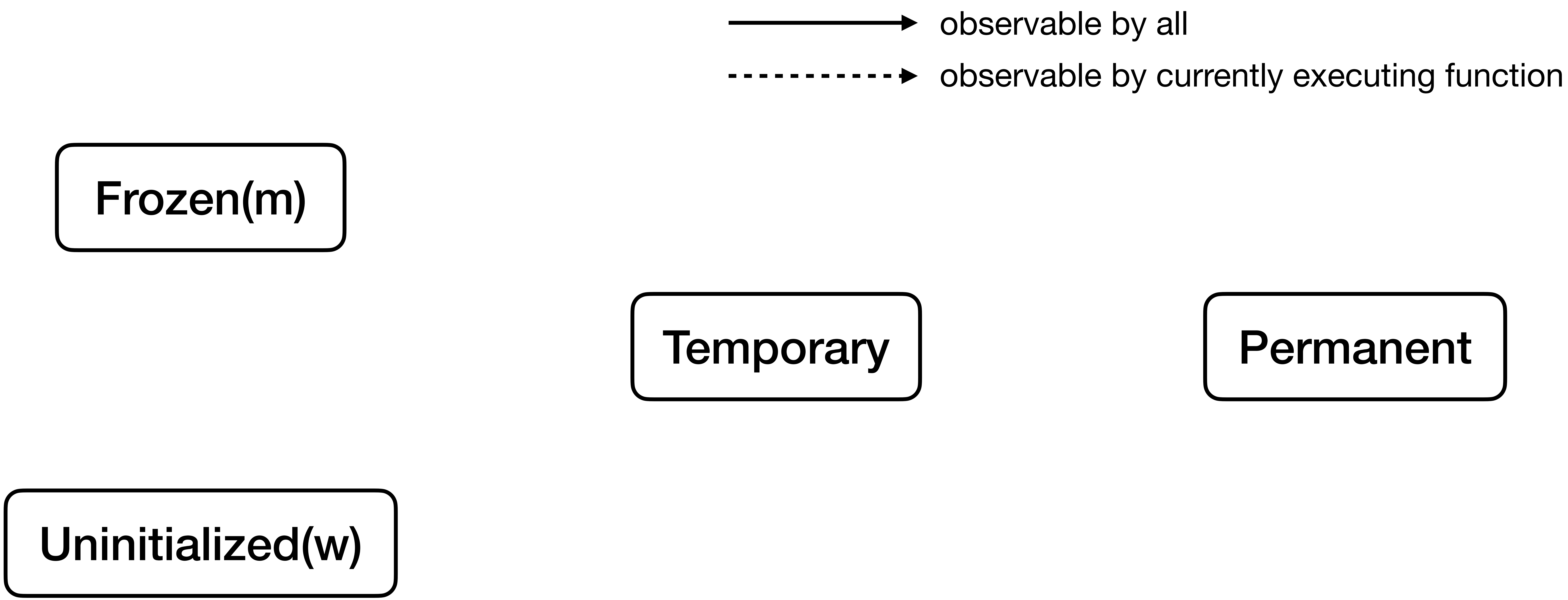
Modelling Lifetime Behaviour of Stack and Heap

Which transitions are safe to observe by the caller, and by the callee?



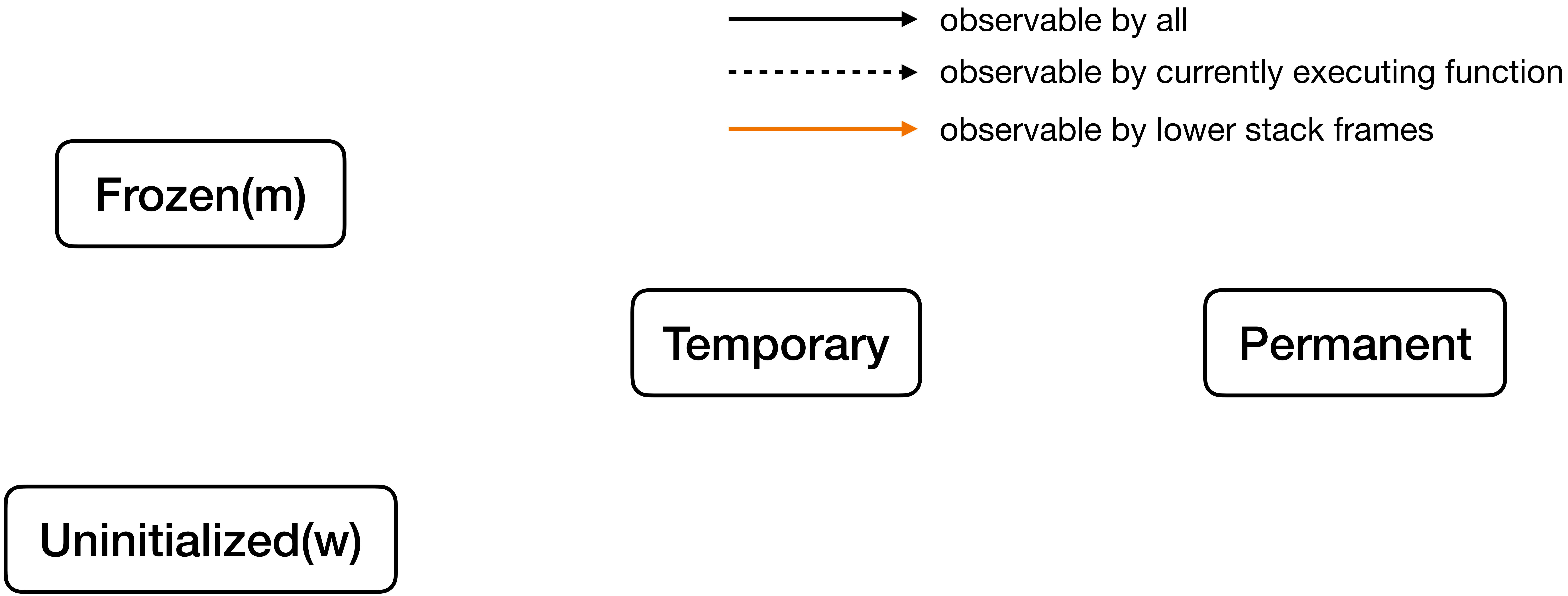
Modelling Lifetime Behaviour of Stack and Heap

Which transitions are safe to observe by the caller, and by the callee?



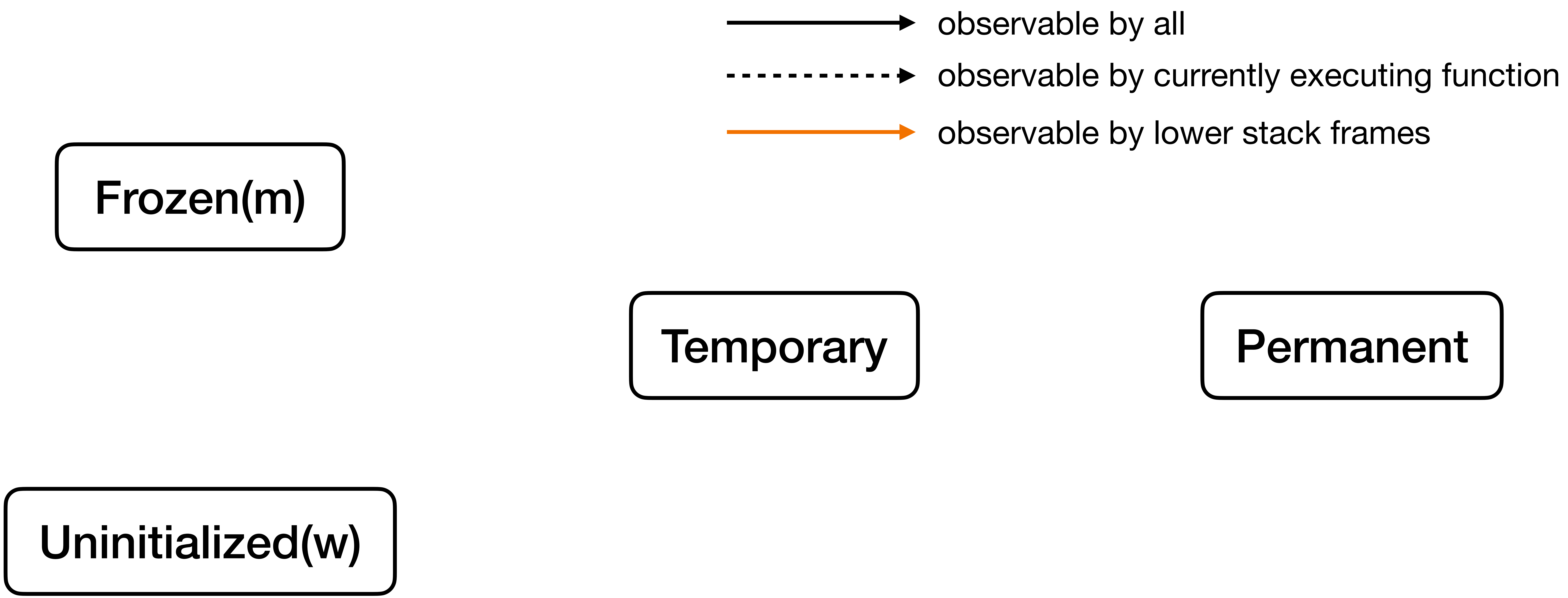
Modelling Lifetime Behaviour of Stack and Heap

Which transitions are safe to observe by the caller, and by the callee?



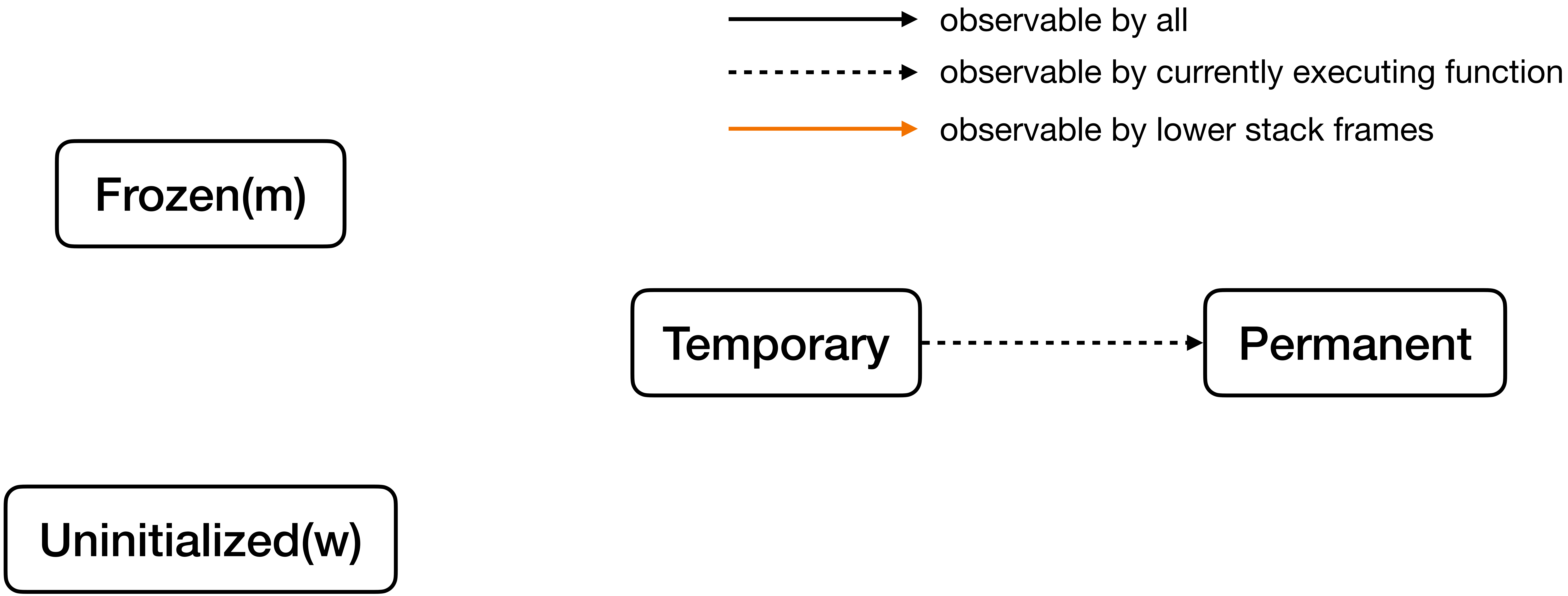
Modelling Lifetime Behaviour of Stack and Heap

Which transitions are safe to observe by the caller, and by the callee?



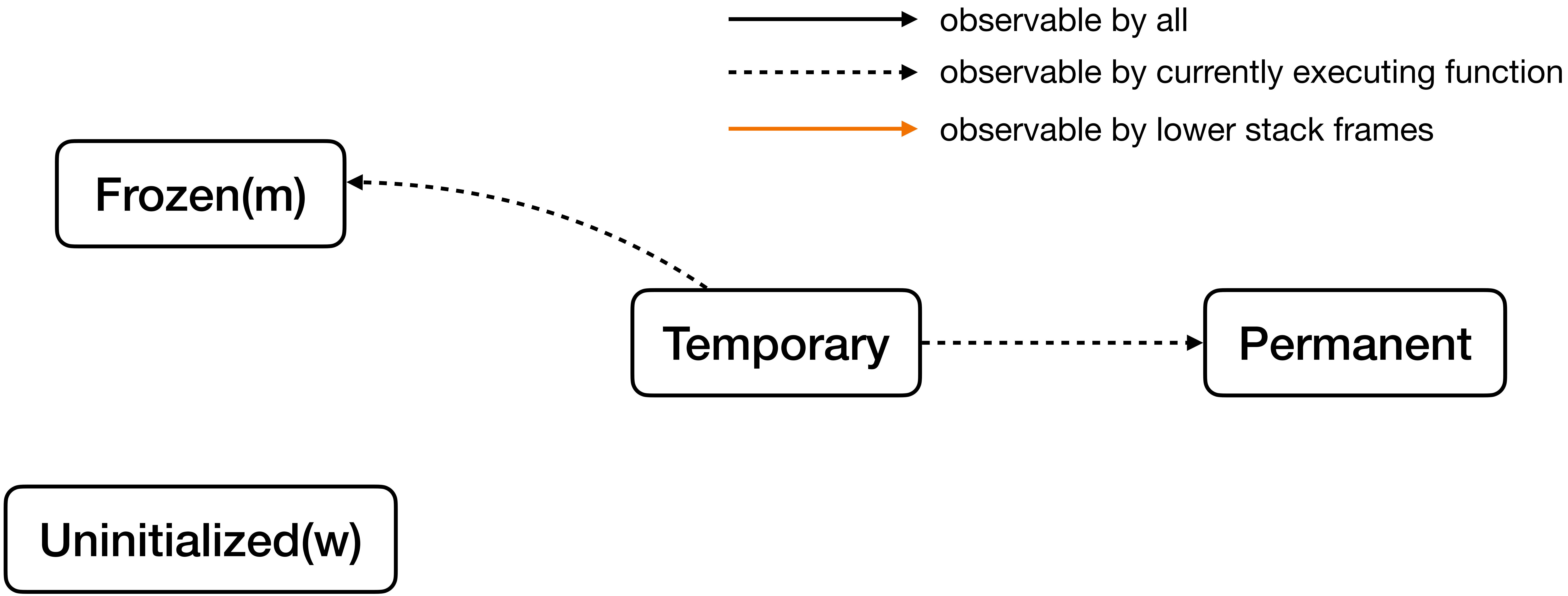
Modelling Lifetime Behaviour of Stack and Heap

Which transitions are safe to observe by the caller, and by the callee?



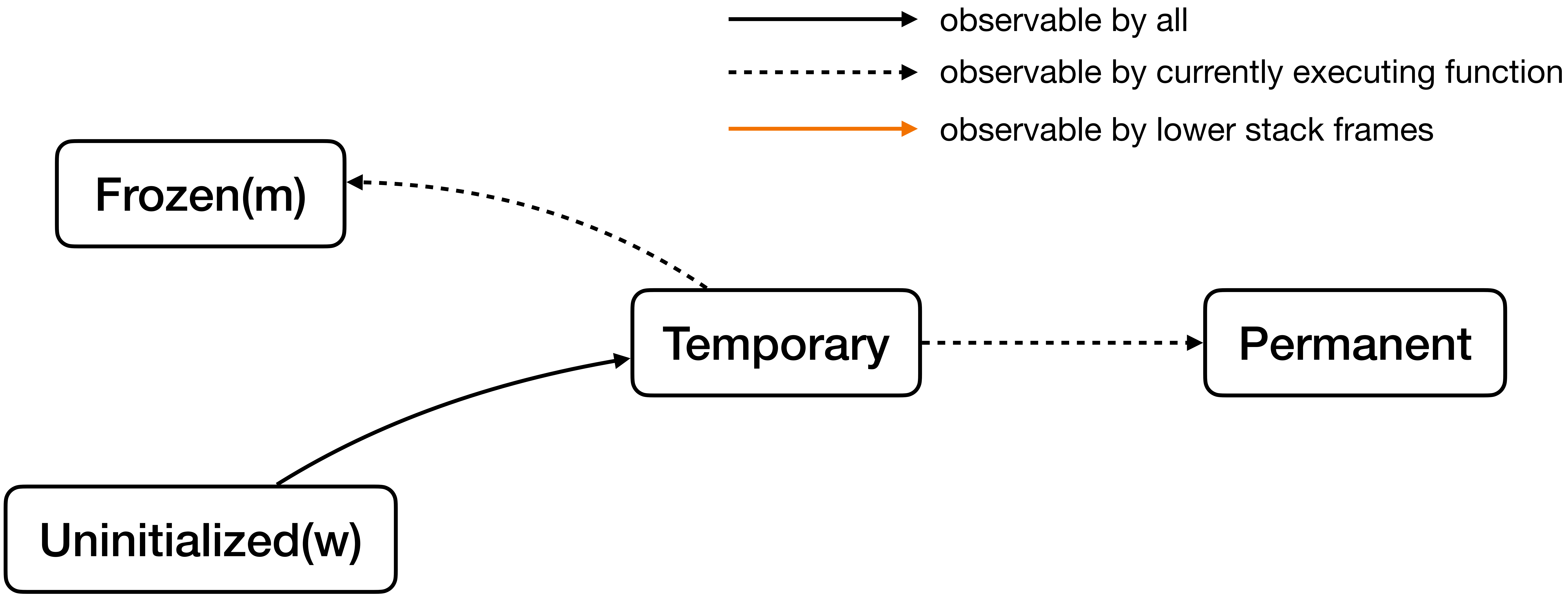
Modelling Lifetime Behaviour of Stack and Heap

Which transitions are safe to observe by the caller, and by the callee?



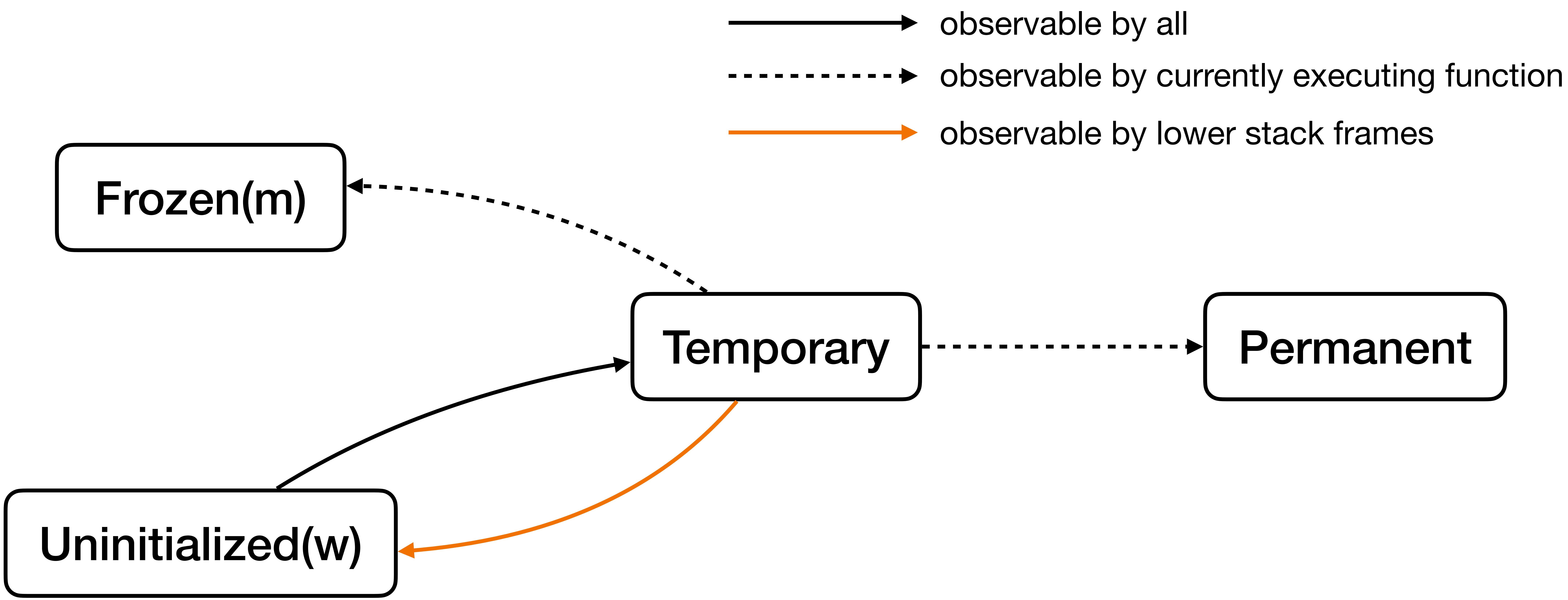
Modelling Lifetime Behaviour of Stack and Heap

Which transitions are safe to observe by the caller, and by the callee?



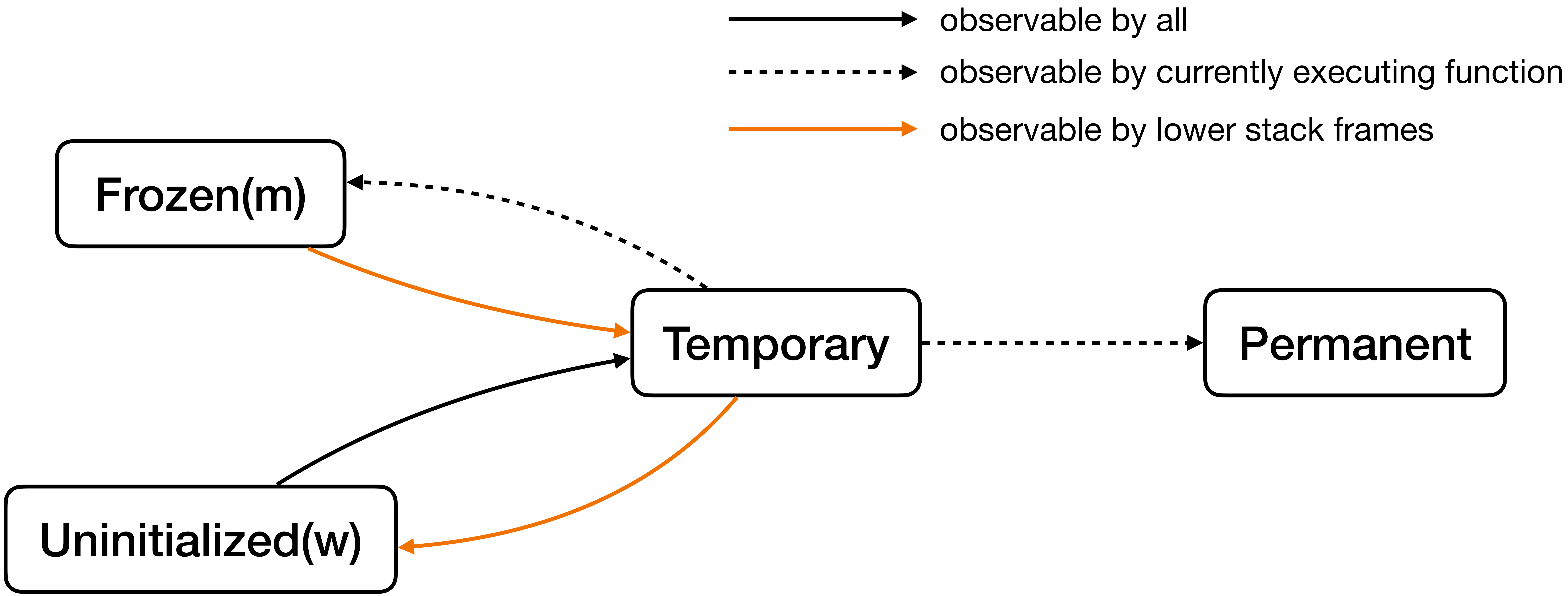
Modelling Lifetime Behaviour of Stack and Heap

Which transitions are safe to observe by the caller, and by the callee?



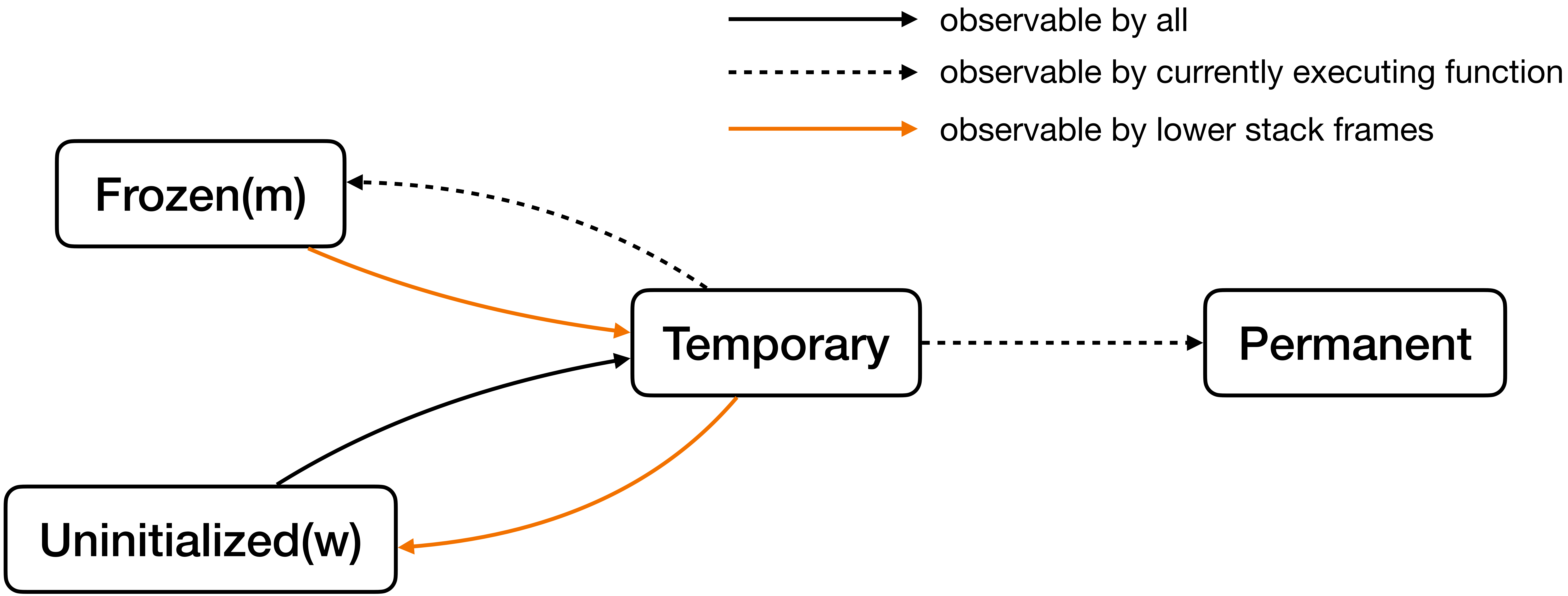
Modelling Lifetime Behaviour of Stack and Heap

Which transitions are safe to observe by the caller, and by the callee?



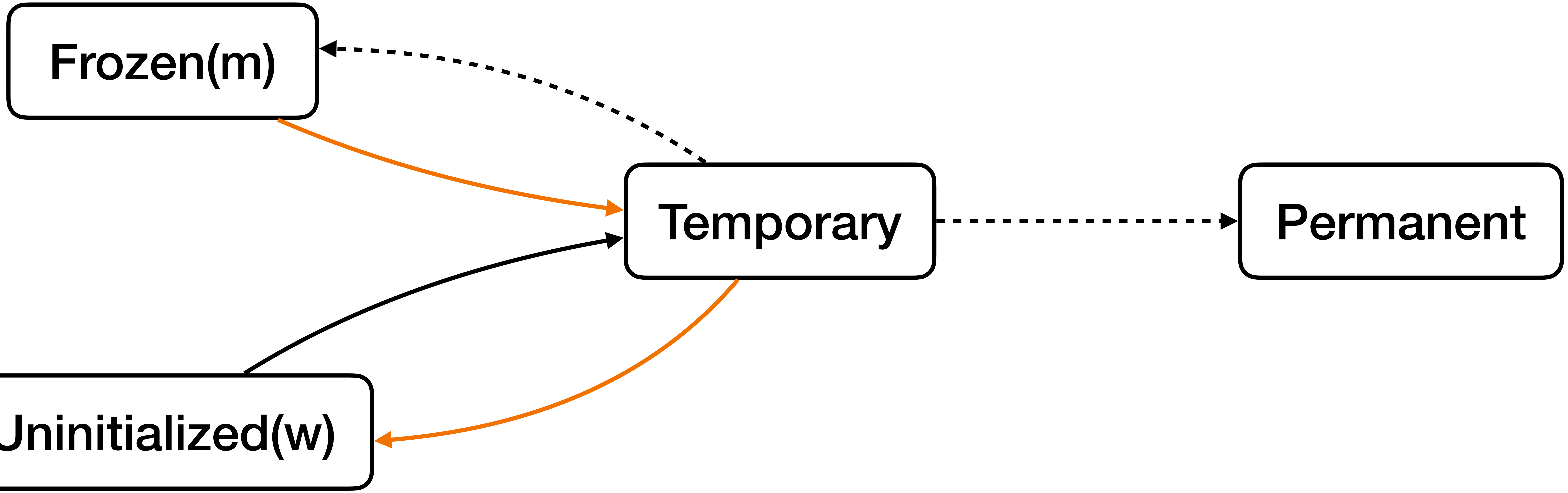
Modelling Lifetime Behaviour of Stack and Heap

Which transitions are safe to observe by the caller, and by the callee?



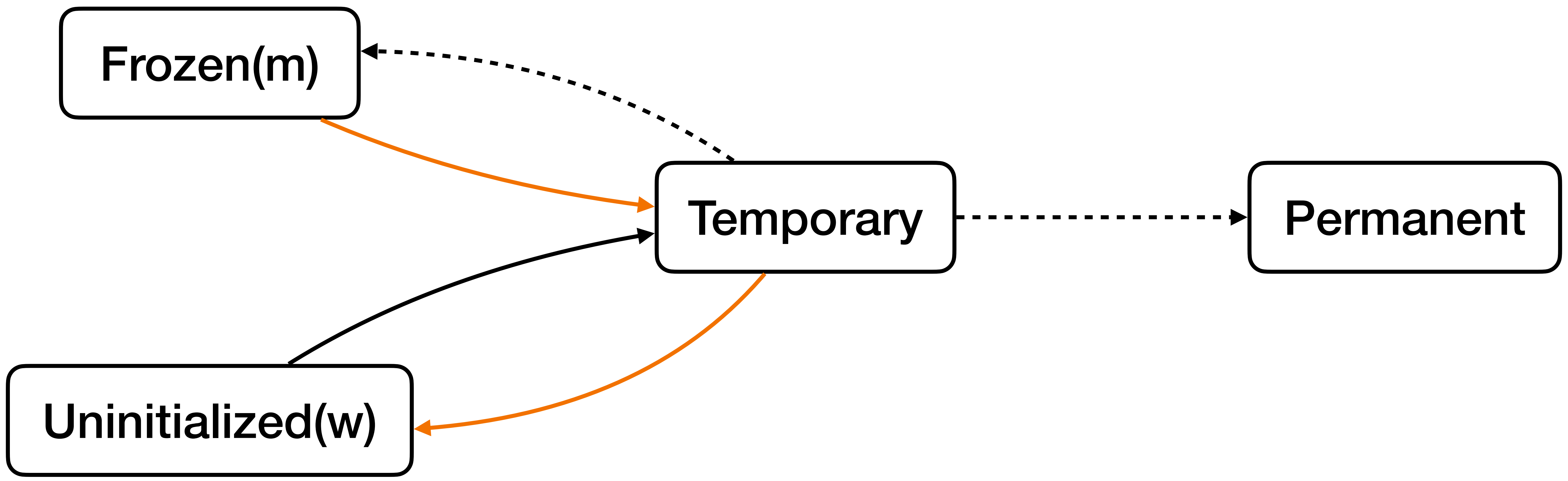
standard state transition system

Modelling Lifetime Behaviour of Stack and Heap



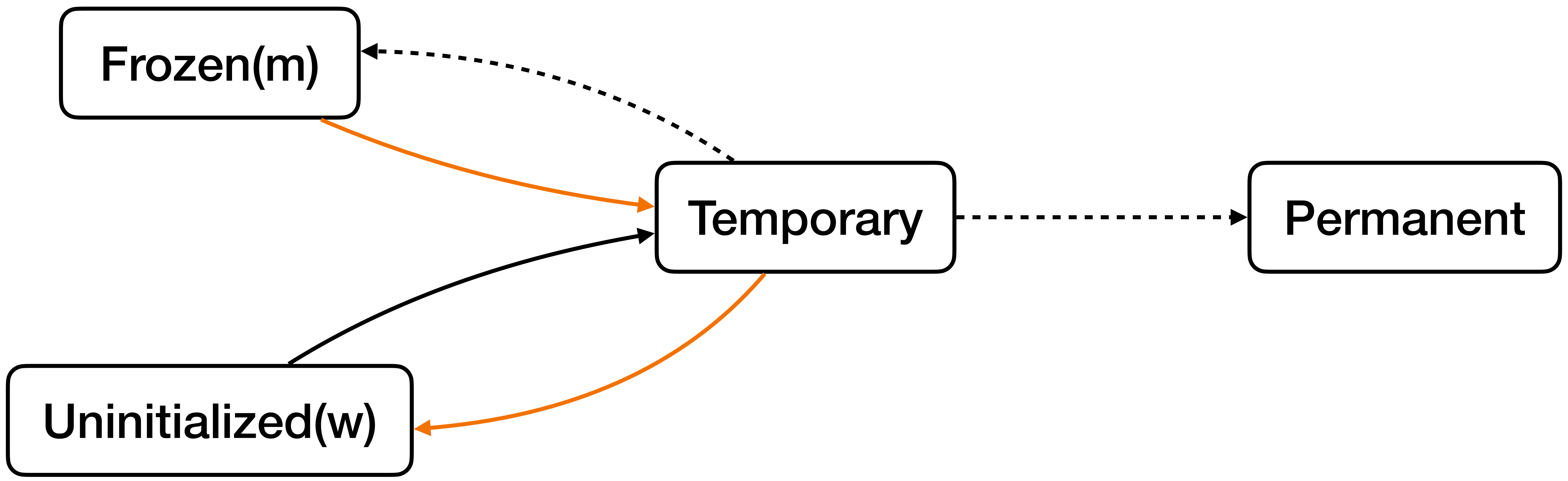
standard state transition system

Modelling Lifetime Behaviour of Stack and Heap



standard state transition system

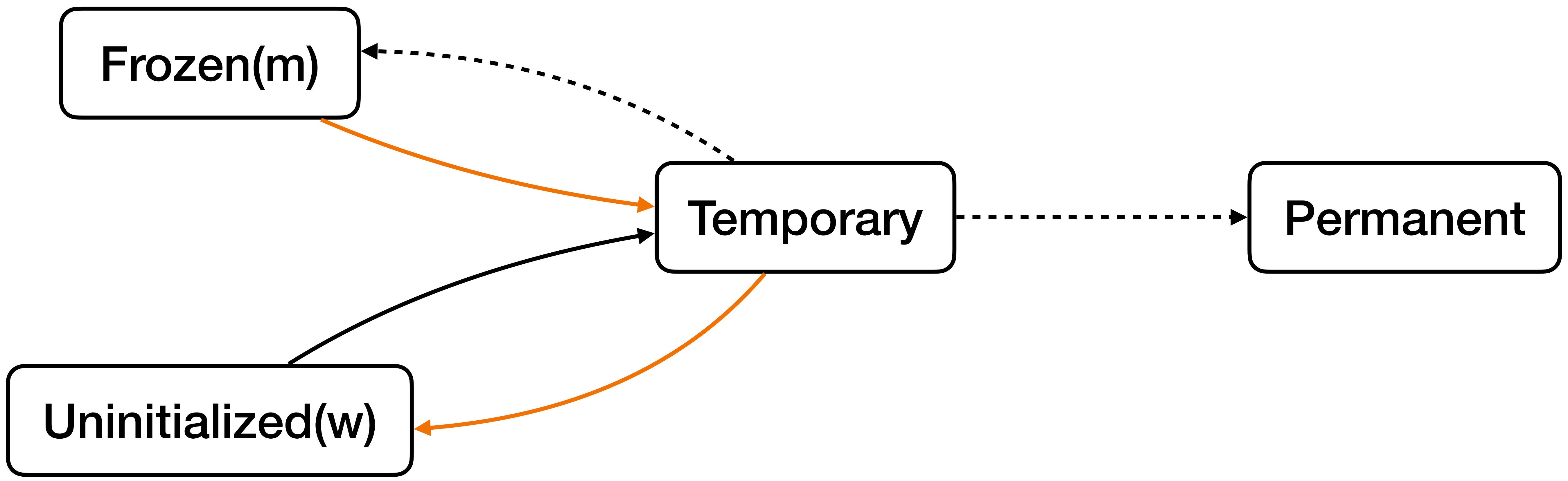
Modelling Lifetime Behaviour of Stack and Heap



standard state transition system

- public future world relation $\sqsupseteq^{pub}$

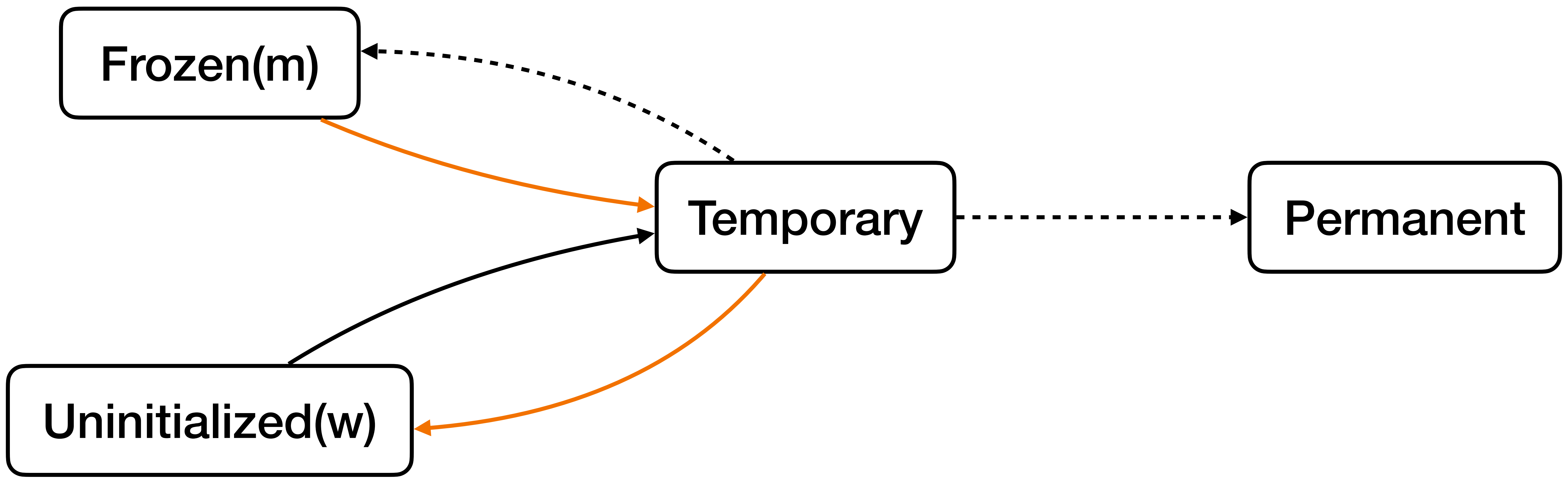
Modelling Lifetime Behaviour of Stack and Heap



standard state transition system

- public future world relation $\sqsupseteq^{pub}$
- private future world relation $\sqsupseteq^{priv}$

Modelling Lifetime Behaviour of Stack and Heap



standard state transition system

- public future world relation $\sqsupseteq^{pub}$
- private future world relation $\sqsupseteq^{priv}$
- *relative* future world relation $\sqsupseteq^{a}$

A Kripke World

$$\mathcal{V} : \mathsf{WORLD} \to \mathsf{Word} \to iProp$$

$$\mathcal{E} : \mathsf{WORLD} \to \mathsf{Word} \to iProp$$

Where a WORLD is a map from addresses to standard states

A Kripke World

$$\mathcal{V} : \text{WORLD} \to \text{Word} \to iProp$$

$$\mathcal{E} : \text{WORLD} \to \text{Word} \to iProp$$

Where a WORLD is a map from addresses to standard states

We will need to satisfy the following monotonicity requirements:

A Kripke World

$$\mathcal{V} : \text{WORLD} \rightarrow \text{Word} \rightarrow iProp$$

$$\mathcal{E} : \text{WORLD} \rightarrow \text{Word} \rightarrow iProp$$

Where a WORLD is a map from addresses to standard states

We will need to satisfy the following monotonicity requirements:

- For **uninitialized** capabilities: $W' \sqsupseteq^a W \rightarrow \mathcal{V}(W)(p, g, b, e, a) \mathrel{-\!\!*} \mathcal{V}(W')(p, g, b, e, a)$

A Kripke World

$$\mathcal{V} : \text{WORLD} \rightarrow \text{Word} \rightarrow iProp$$

$$\mathcal{E} : \text{WORLD} \rightarrow \text{Word} \rightarrow iProp$$

Where a WORLD is a map from addresses to standard states

We will need to satisfy the following monotonicity requirements:

- For **uninitialized** capabilities: $W' \sqsupseteq^a W \rightarrow \mathcal{V}(W)(p,g,b,e,a) \longrightarrow\!\!\ast\; \mathcal{V}(W')(p,g,b,e,a)$
- For **non uninitialized** capabilities: $W' \sqsupseteq^e W \rightarrow \mathcal{V}(W)(p,g,b,e,a) \longrightarrow\!\!\ast\; \mathcal{V}(W')(p,g,b,e,a)$

A Kripke World

$$\mathcal{V} : \text{WORLD} \rightarrow \text{Word} \rightarrow iProp$$

$$\mathcal{E} : \text{WORLD} \rightarrow \text{Word} \rightarrow iProp$$

Where a WORLD is a map from addresses to standard states

We will need to satisfy the following monotonicity requirements:

- For **uninitialized** capabilities: $W' \sqsupseteq^{a} W \rightarrow \mathcal{V}(W)(p, g, b, e, a) \twoheadrightarrow \mathcal{V}(W')(p, g, b, e, a)$
- For **non uninitialized** capabilities: $W' \sqsupseteq^{e} W \rightarrow \mathcal{V}(W)(p, g, b, e, a) \twoheadrightarrow \mathcal{V}(W')(p, g, b, e, a)$
- For **Global** capabilities: $W' \sqsupseteq^{priv} W \rightarrow \mathcal{V}(W)(p, \text{GLOBAL}, b, e, a) \twoheadrightarrow \mathcal{V}(W')(p, \text{GLOBAL}, b, e, a)$

Back to the Unary Logical Relation

$$\mathcal{V}(W)(\textsc{e}, \textsc{directed}, b, e, a) \triangleq \Box \forall W' \sqsupseteq^{e} W, \triangleright \mathcal{E}(W')(\textsc{rx}, \textsc{directed}, b, e, a)$$

$$\mathcal{V}(W)(\textsc{e}, \textsc{global}, b, e, a) \triangleq \Box \forall W' \sqsupseteq^{priv} W, \triangleright \mathcal{E}(W')(\textsc{rx}, \textsc{global}, b, e, a)$$

Back to the Unary Logical Relation

$$\mathcal{V}(W)(\textsc{e}, \textsc{directed}, b, e, a) \triangleq \square \forall W' \sqsupseteq^{e} W, \triangleright \mathcal{E}(W')(\textsc{rx}, \textsc{directed}, b, e, a)$$

$$\mathcal{V}(W)(\textsc{e}, \textsc{global}, b, e, a) \triangleq \square \forall W' \sqsupseteq^{priv} W, \triangleright \mathcal{E}(W')(\textsc{rx}, \textsc{global}, b, e, a)$$

$$\mathcal{V}(W)(\textsc{rwlx}, \textsc{directed}, b, e, -) \triangleq \mathop{\Huge *}_{a \in [b,e)} rel(a, \mathcal{V}) * W(a) = \text{Temporary}$$

Back to the Unary Logical Relation

$$\mathcal{V}(W)(\text{E}, \text{DIRECTED}, b, e, a) \triangleq \square \forall W' \sqsupseteq^{e} W, \triangleright \mathcal{E}(W')(\text{RX}, \text{DIRECTED}, b, e, a)$$

$$\mathcal{V}(W)(\text{E}, \text{GLOBAL}, b, e, a) \triangleq \square \forall W' \sqsupseteq^{priv} W, \triangleright \mathcal{E}(W')(\text{RX}, \text{GLOBAL}, b, e, a)$$

$$\mathcal{V}(W)(\text{RWLX}, \text{DIRECTED}, b, e, -) \triangleq \mathop{\Huge *}_{a \in [b,e)} rel(a, \mathcal{V}) * W(a) = \text{Temporary}$$

$$\mathcal{E}(W)(w) \triangleq \forall reg,$$
$$\{\cdots * stsCollection(W) * sharedResources(W)\}$$
Executable
$$\{\cdots * \exists W' \sqsupseteq^{priv} W, stsCollection(W') * sharedResources(W')\}$$

Standard Resources

Standard Resources

$$\mathsf{MonoReq}(W, \phi, v, \sqsupseteq) \quad \triangleq \quad \Box \forall W', W' \sqsupseteq W \rightarrow \phi(W, v) \twoheadrightarrow \phi(W', v)$$

Standard Resources

$$\mathsf{MonoReq}(W, \phi, v, \sqsupseteq) \quad \triangleq \quad \Box \forall W', W' \sqsupseteq W \rightarrow \phi(W, v) \mathrel{-\!\!*} \phi(W', v)$$

$$\mathsf{permR}(a, W, \phi) \quad \triangleq \quad \exists v, a \mapsto v * \triangleright \phi(W, v) * \mathsf{MonoReq}(W, \phi, v, \sqsupseteq^{priv})$$

Standard Resources

$$\mathsf{MonoReq}(W, \phi, v, \sqsupseteq) \quad \triangleq \quad \Box \forall W', W' \sqsupseteq W \to \phi(W, v) \mathrel{-\!\!*} \phi(W', v)$$

$$\mathsf{permR}(a, W, \phi) \quad \triangleq \quad \exists v, a \mapsto v * \triangleright \phi(W, v) * \mathsf{MonoReq}(W, \phi, v, \sqsupseteq^{priv})$$

$$\mathsf{tempR}(a, W, \phi) \quad \triangleq \quad \exists v, a \mapsto v * \triangleright \phi(W, v) * \mathsf{MonoReq}(W, \phi, v, \sqsupseteq^{a})$$

Standard Resources

$$\mathsf{MonoReq}(W, \phi, v, \sqsupseteq) \triangleq \Box \forall W', W' \sqsupseteq W \rightarrow \phi(W, v) \twoheadrightarrow \phi(W', v)$$

$$\mathsf{permR}(a, W, \phi) \triangleq \exists v, a \mapsto v * \triangleright \phi(W, v) * \mathsf{MonoReq}(W, \phi, v, \sqsupseteq^{priv})$$

$$\mathsf{tempR}(a, W, \phi) \triangleq \exists v, a \mapsto v * \triangleright \phi(W, v) * \mathsf{MonoReq}(W, \phi, v, \sqsupseteq^{a})$$

$$\mathsf{uninitR}(a, v) \triangleq a \mapsto v$$

Standard Resources

$$\mathsf{MonoReq}(W, \phi, v, \sqsupseteq) \triangleq \Box \forall W', W' \sqsupseteq W \to \phi(W, v) \mathrel{-\!\!*} \phi(W', v)$$

$$\mathsf{permR}(a, W, \phi) \triangleq \exists v, a \mapsto v * \triangleright \phi(W, v) * \mathsf{MonoReq}(W, \phi, v, \sqsupseteq^{priv})$$

$$\mathsf{tempR}(a, W, \phi) \triangleq \exists v, a \mapsto v * \triangleright \phi(W, v) * \mathsf{MonoReq}(W, \phi, v, \sqsupseteq^{a})$$

$$\mathsf{uninitR}(a, v) \triangleq a \mapsto v$$

$$\mathsf{frozenR}(a, m) \triangleq a \mapsto m(a) * \forall a' \in dom(m), W(a') = \mathsf{Frozen}(m)$$

Standard Resources

$$\mathsf{MonoReq}(W, \phi, v, \sqsupseteq) \triangleq \Box \forall W', W' \sqsupseteq W \to \phi(W, v) \twoheadrightarrow \phi(W', v)$$

$$\mathsf{permR}(a, W, \phi) \triangleq \exists v, a \mapsto v * \triangleright \phi(W, v) * \mathsf{MonoReq}(W, \phi, v, \sqsupseteq^{priv})$$

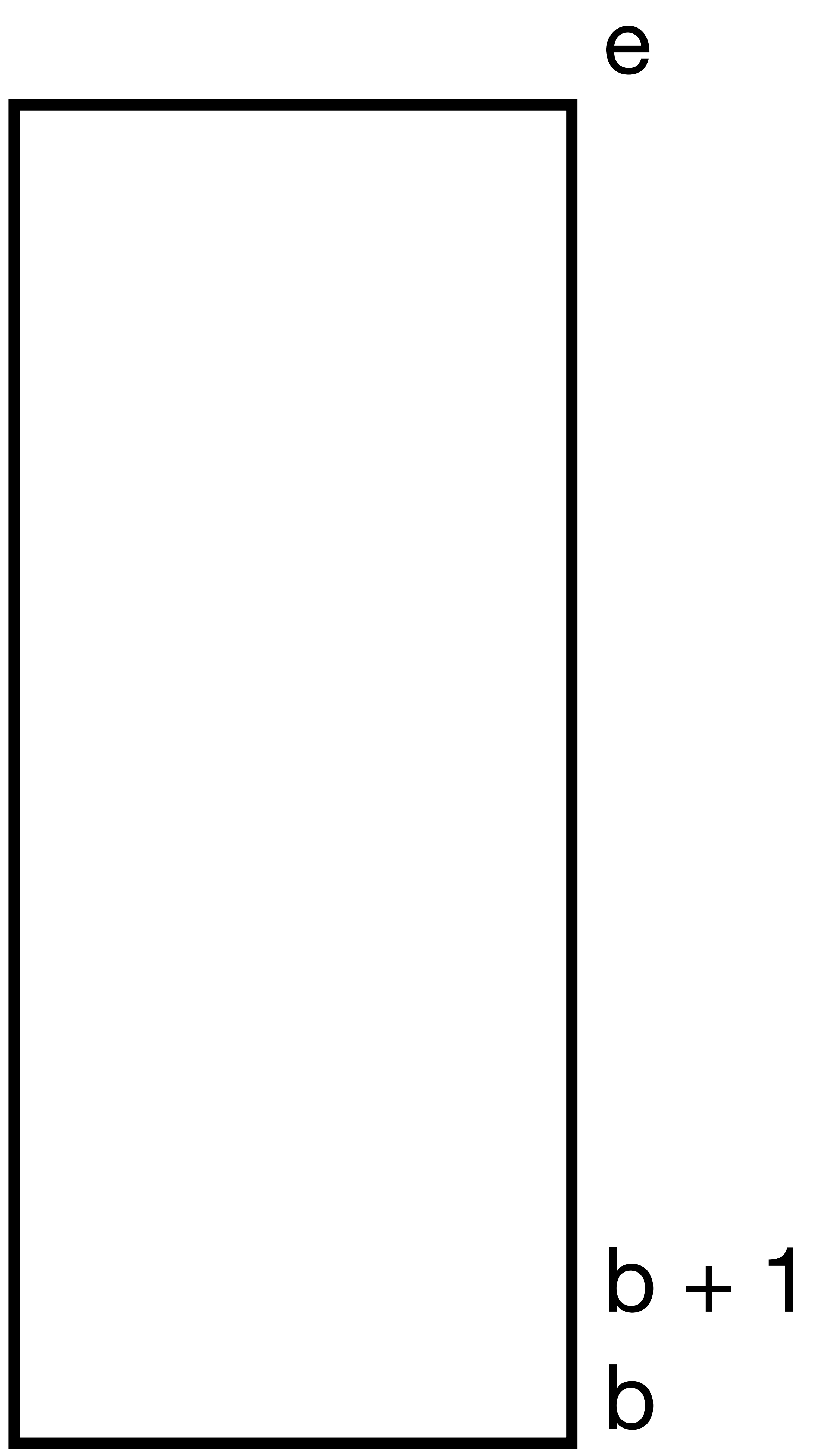
$$\mathsf{tempR}(a, W, \phi) \triangleq \exists v, a \mapsto v * \triangleright \phi(W, v) * \mathsf{MonoReq}(W, \phi, v, \sqsupseteq^{a})$$

$$\mathsf{uninitR}(a, v) \triangleq a \mapsto v$$

$$\mathsf{frozenR}(a, m) \triangleq a \mapsto m(a) * \forall a' \in dom(m), W(a') = \mathsf{Frozen}(m)$$

Returning to our Example

```
f1: prepstack r_stk
    loadU r0 r_stk -1
    push r_env
    load r_env r_env
    assert r_env 2
    rclear RegName\{PC,r0}
    jmp r0
```

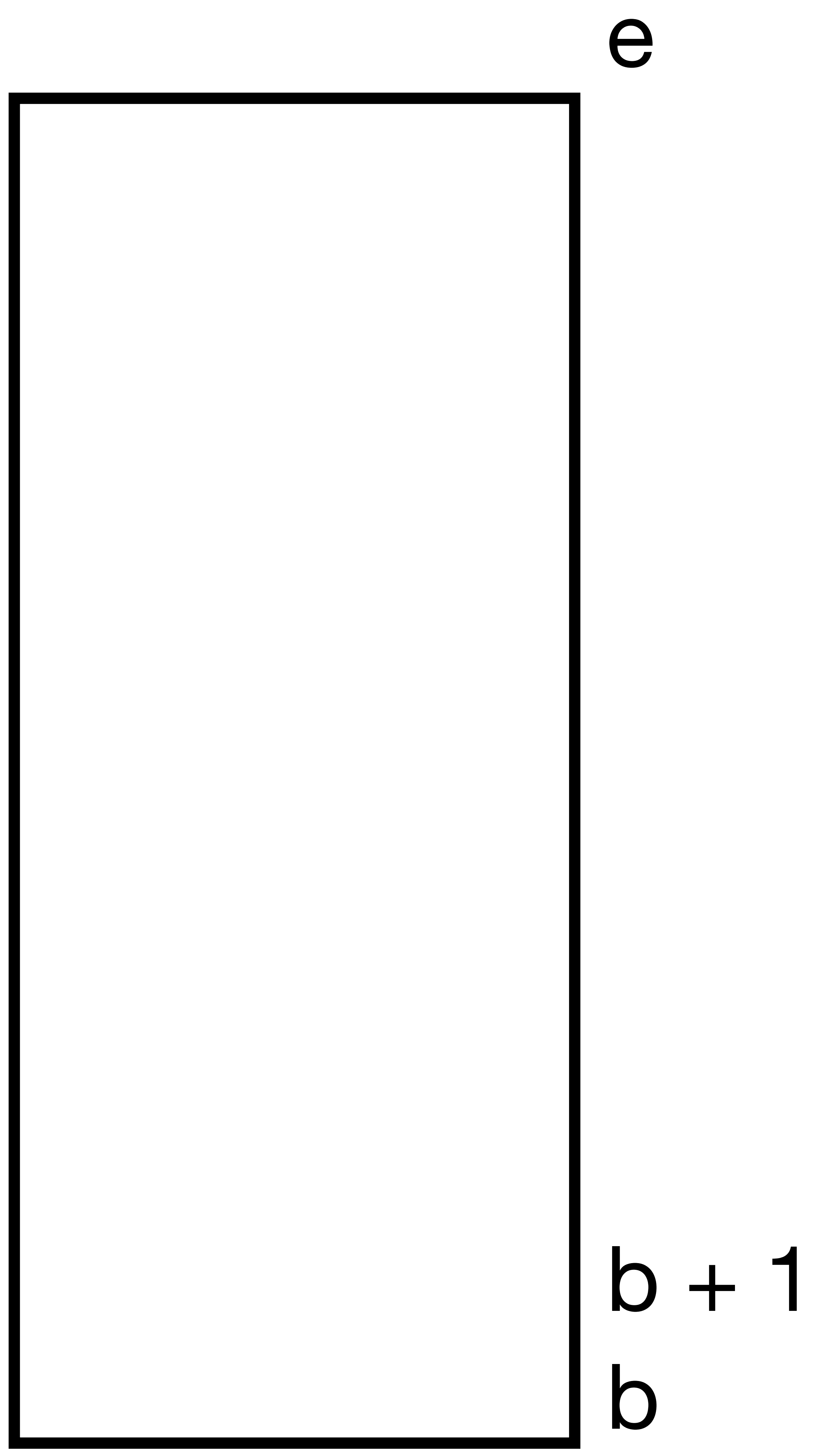


Returning to our Example

```
f1: prepstack r_stk
    loadU r0 r_stk -1
    push r_env
    load r_env r_env
    assert r_env 2
    rclear RegName\{PC,r0}
    jmp r0
```

We begin with:

$$stsCollection(W)$$
$$sharedResources(W)$$



Returning to our Example

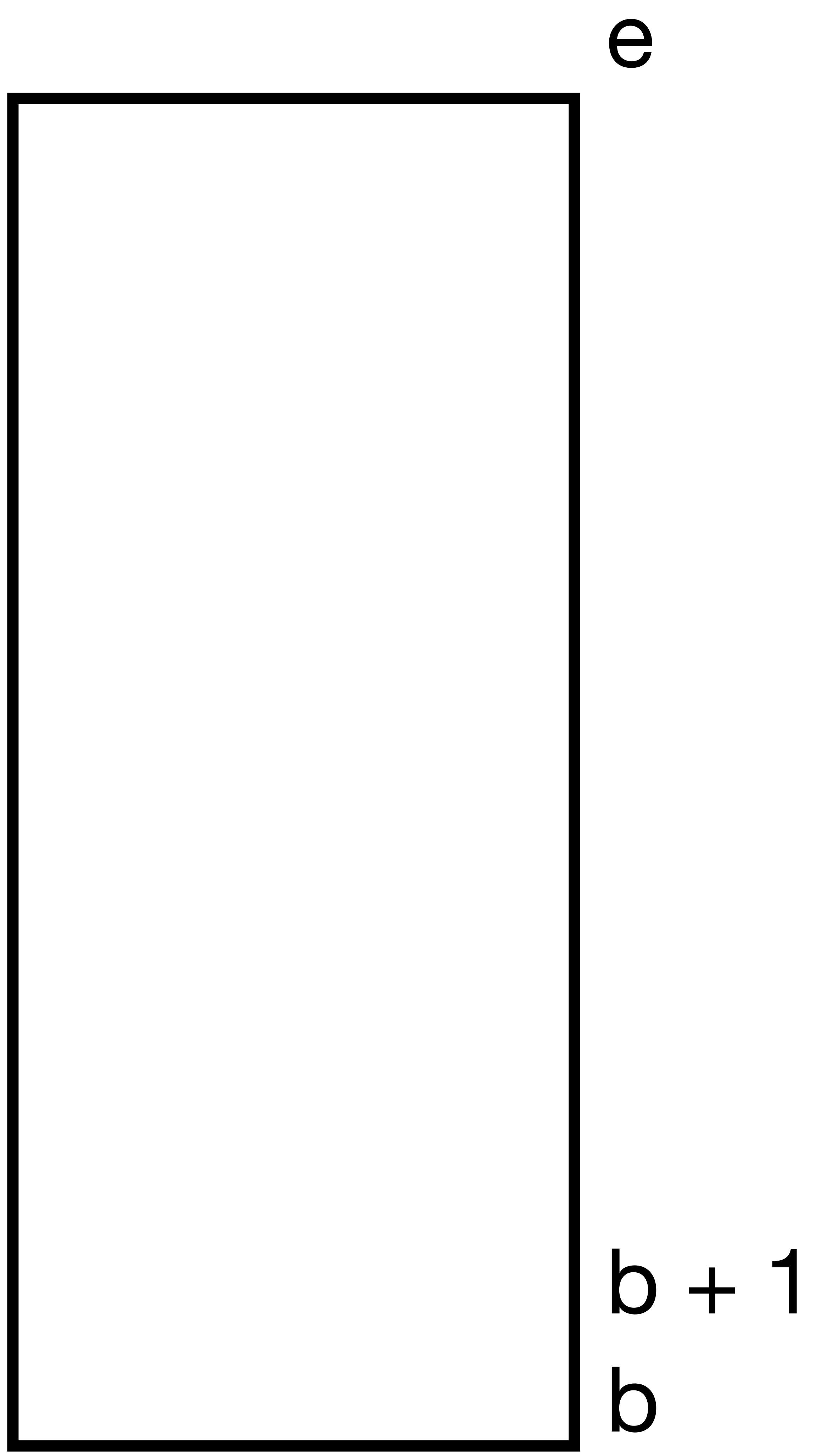
```
f1: prepstack r_stk
    loadU r0 r_stk -1
    push r_env
    load r_env r_env
    assert r_env 2
    rclear RegName\{PC,r0}
    jmp r0
```

Context:

$$\mathcal{V}(W)(\text{URWLX}, \text{DIRECTED}, b, e, b+1)$$

We begin with:

$$stsCollection(W)$$
$$sharedResources(W)$$



Returning to our Example

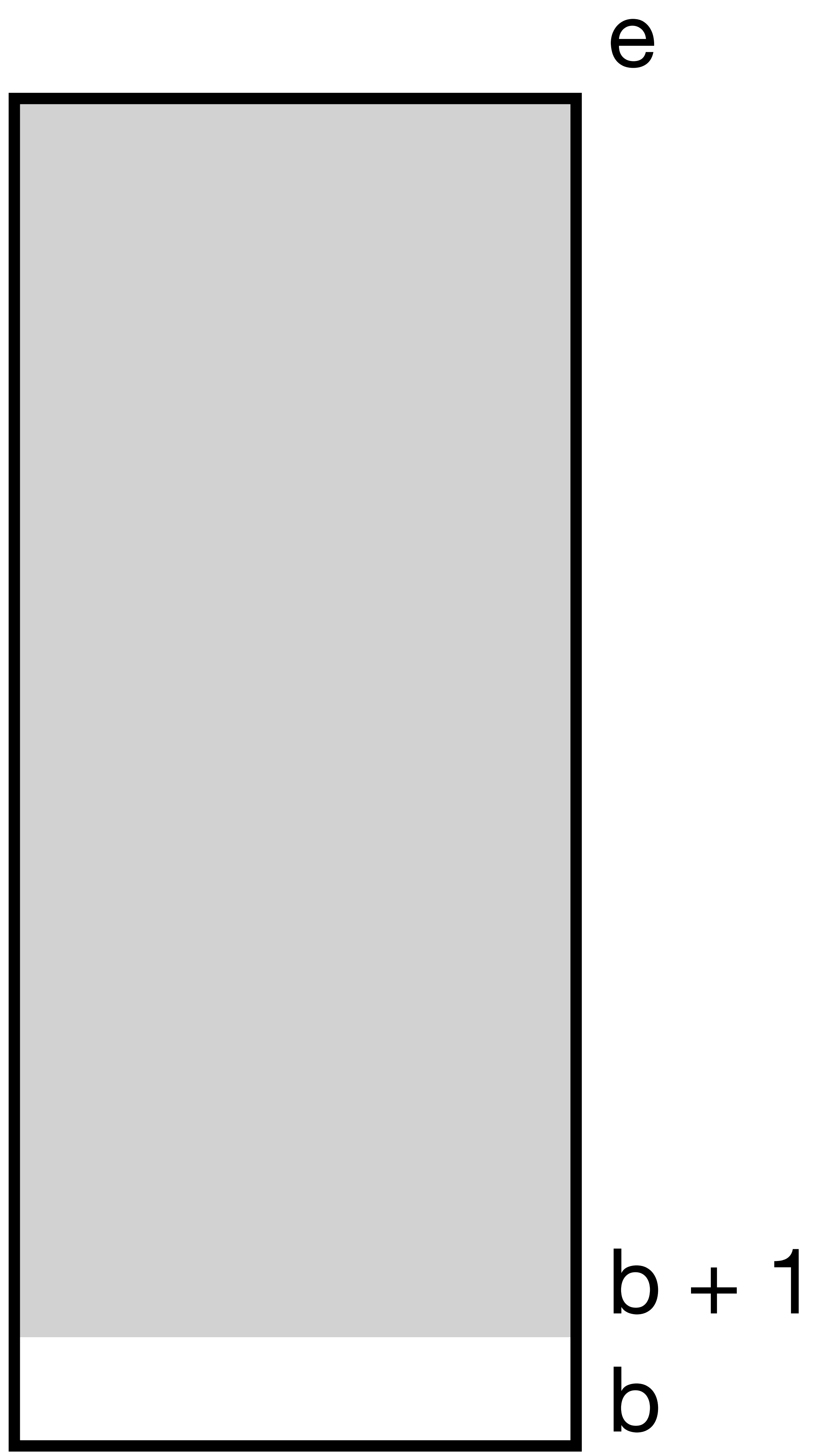
```
f1: prepstack r_stk
    loadU r0 r_stk -1
    push r_env
    load r_env r_env
    assert r_env 2
    rclear RegName\{PC,r0}
    jmp r0
```

Context:

$$\mathcal{V}(W)(\textsc{urwlx}, \textsc{directed}, b, e, b+1)$$

We begin with:

$$stsCollection(W)$$
$$sharedResources(W)$$



Returning to our Example

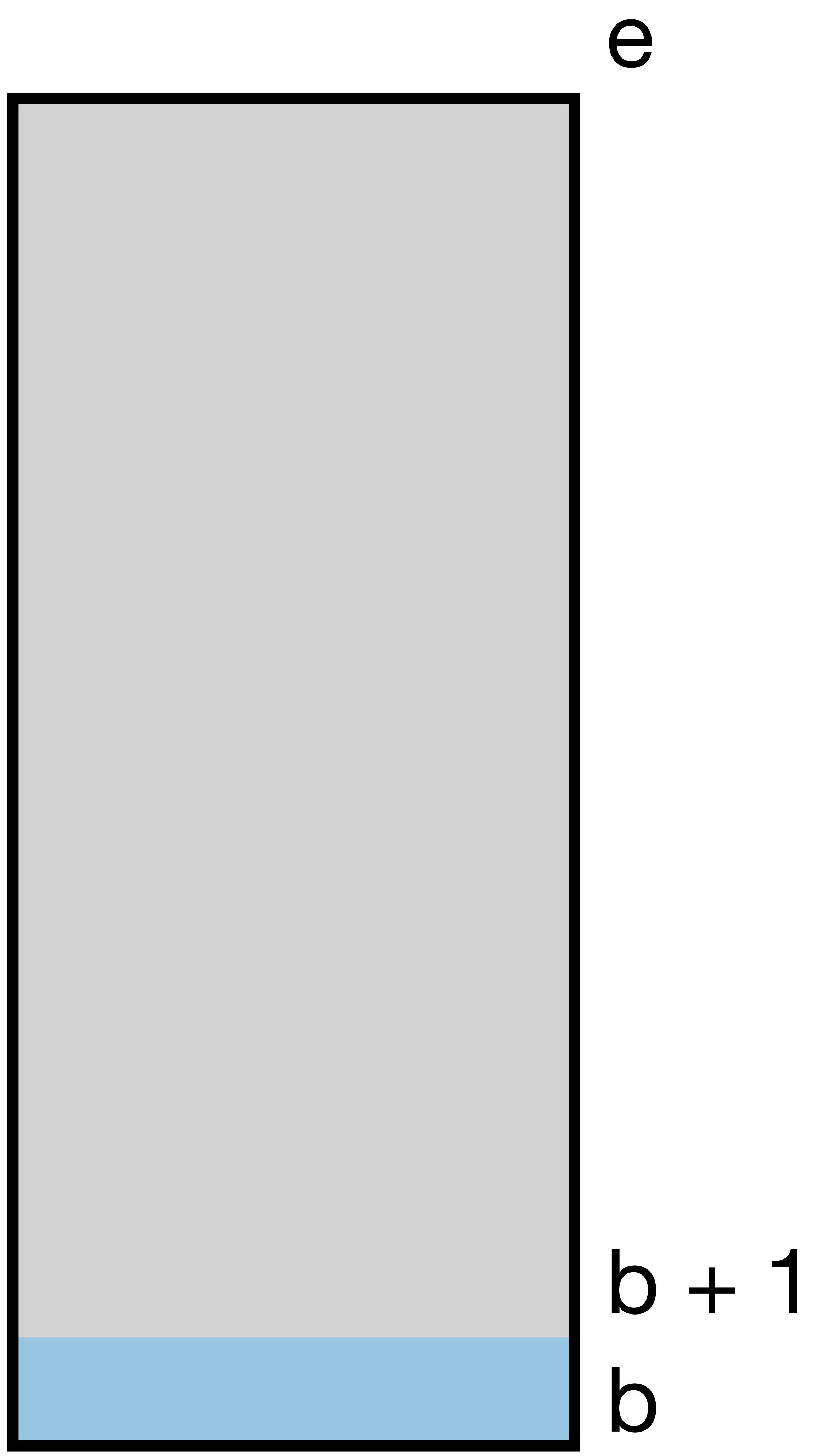
```
f1: prepstack r_stk
    loadU r0 r_stk -1
    push r_env
    load r_env r_env
    assert r_env 2
    rclear RegName\{PC,r0}
    jmp r0
```

Context:

$\mathcal{V}(W)(\text{URWLX}, \text{DIRECTED}, b, e, b+1)$

We begin with:

$$stsCollection(W)$$
$$sharedResources(W)$$



Returning to our Example

```
f1: prepstack r_stk
    loadU r0 r_stk -1
    push r_env
    load r_env r_env
    assert r_env 2
    rclear RegName\{PC,r0}
    jmp r0
```

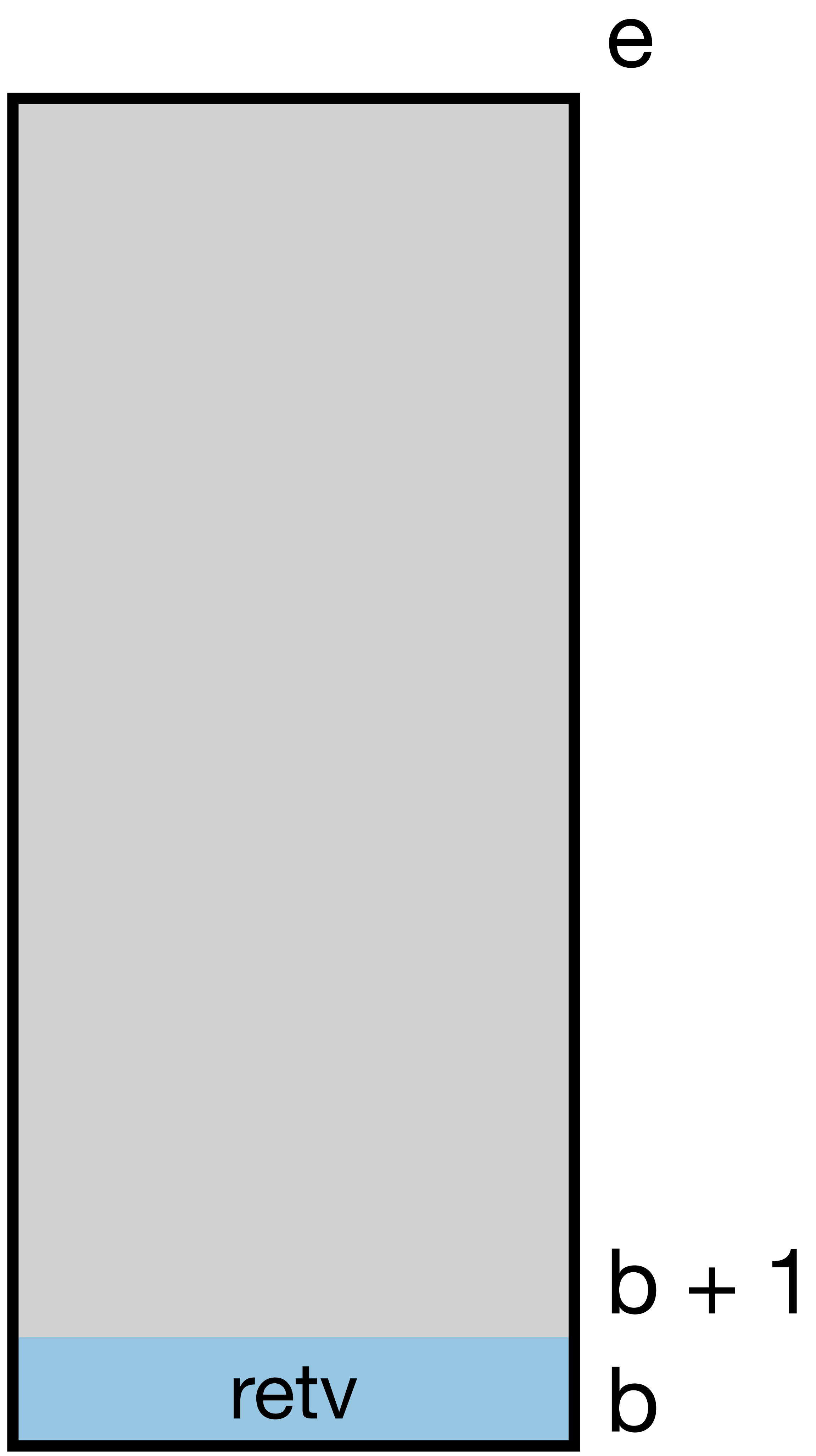
Context:

$\mathcal{V}(W)(\text{URWLX}, \text{DIRECTED}, b, e, b+1)$

$\mathcal{V}(W)(retv) * \text{MonoReq}(\text{W}, \mathcal{V}, retv, \sqsupseteq^{b})$

We begin with:

$stsCollection(W)$
$sharedResources(W)$



Returning to our Example

```
f1: prepstack r_stk
    loadU r0 r_stk -1
    push r_env
    load r_env r_env
    assert r_env 2
    rclear RegName\{PC,r0}
    jmp r0
```

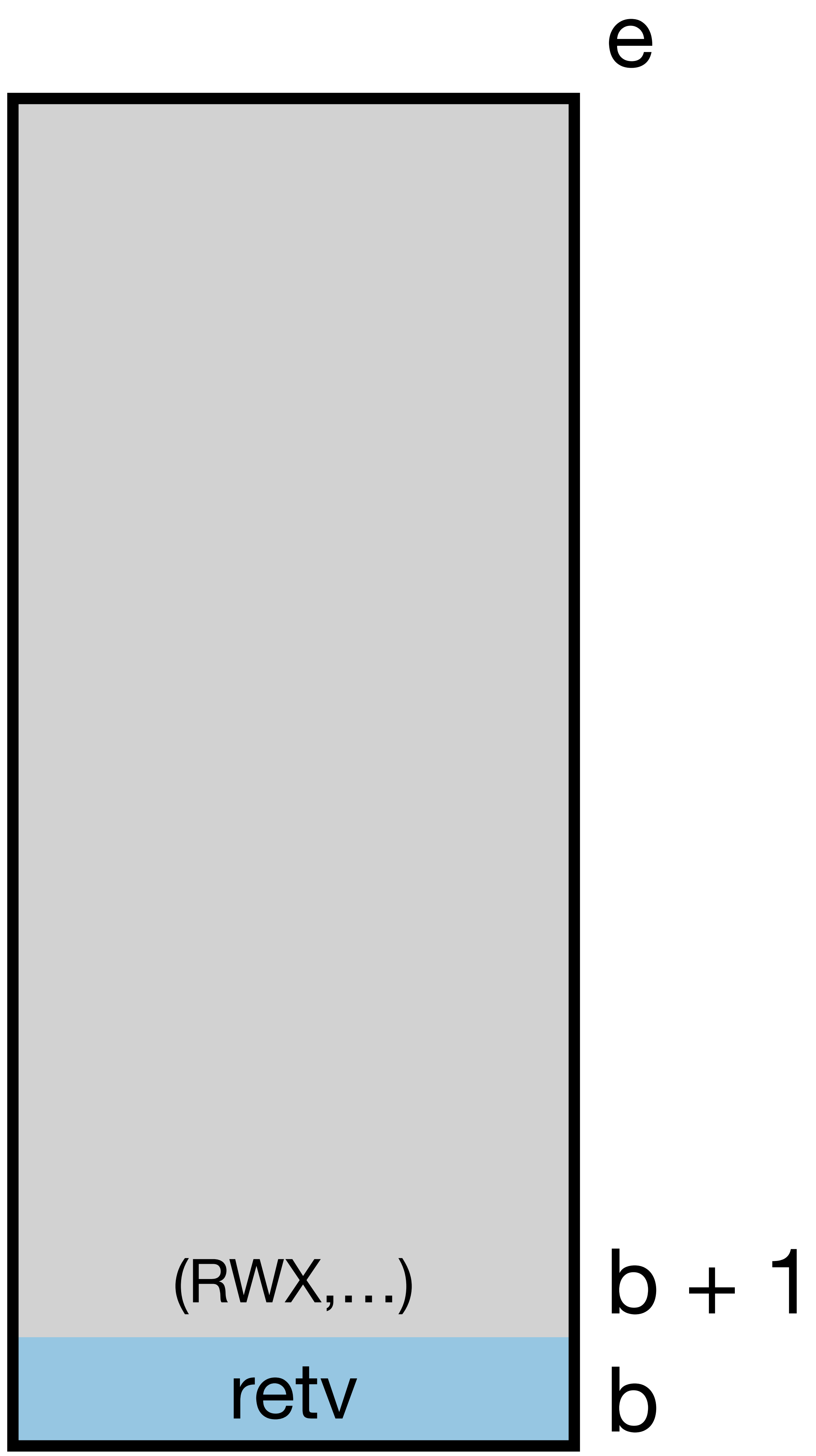
Context:

$$\mathcal{V}(W)(\text{URWLX}, \text{DIRECTED}, b, e, b+1)$$

$$\mathcal{V}(W)(retv) * \text{MonoReq}(\text{W}, \mathcal{V}, retv, \sqsupseteq^{b})$$

We begin with:

$$stsCollection(W)$$
$$sharedResources(W)$$



Returning to our Example

```
f1: prepstack r_stk
    loadU r0 r_stk -1
    push r_env
    load r_env r_env
    assert r_env 2
    rclear RegName\{PC,r0}
    jmp r0
```

Context:

$$\mathcal{V}(W)(\text{URWLX}, \text{DIRECTED}, b, e, b+1)$$

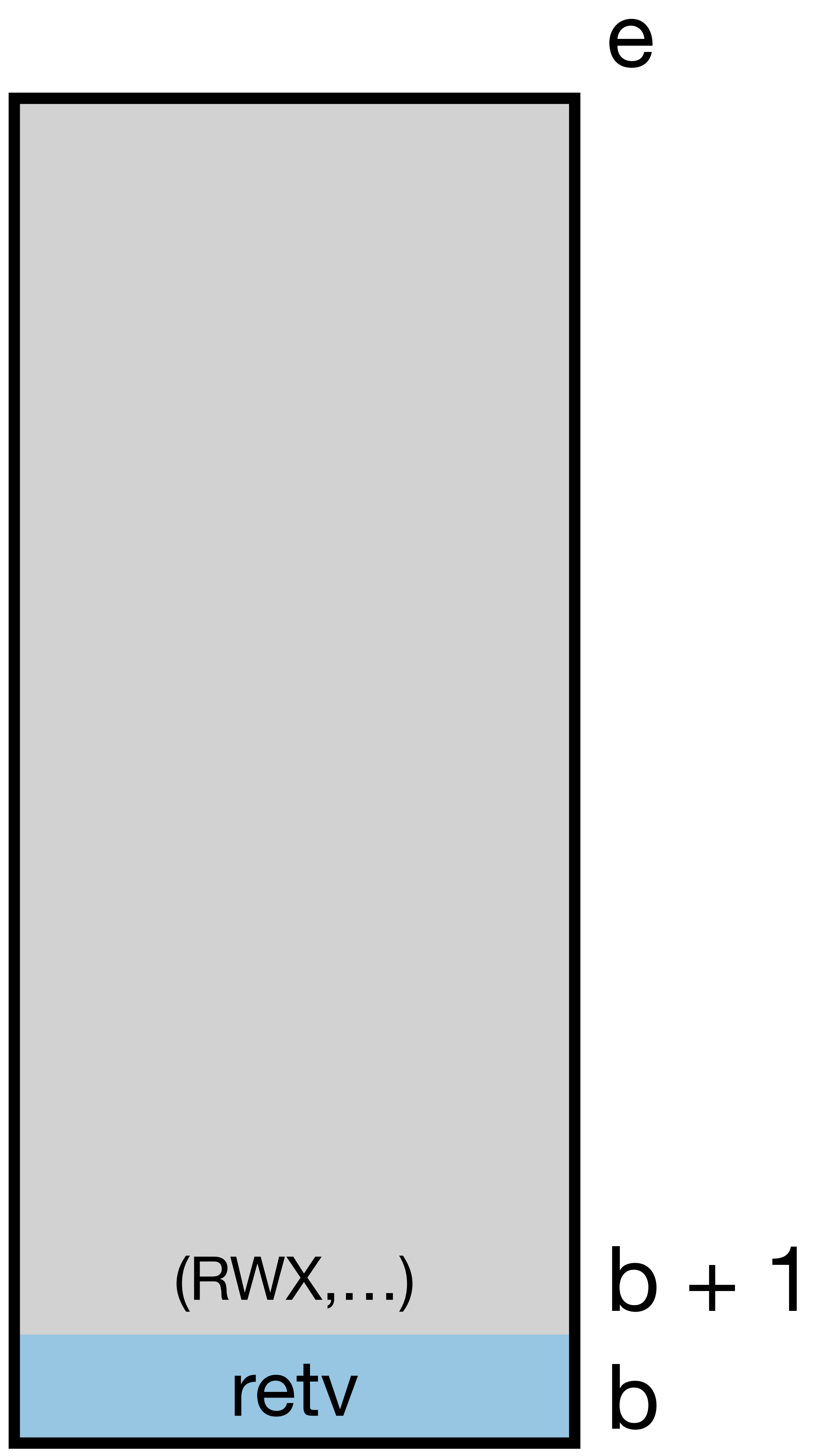
$$\mathcal{V}(W)(retv) * \text{MonoReq}(\text{W}, \mathcal{V}, retv, \sqsupseteq^{b})$$

We begin with:

$$stsCollection(W)$$
$$sharedResources(W)$$

We end with:

$$stsCollection([b+1 := \text{Uninitialized}(\text{RWX}, \cdots)]W)$$
$$sharedResources([b+1 := \text{Uninitialized}(\text{RWX}, \cdots)]W)$$



Returning to our Example

```
f1: prepstack r_stk
    loadU r0 r_stk -1
    push r_env
    load r_env r_env
    assert r_env 2
    rclear RegName\{PC,r0}
    jmp r0
```

Context:

$$\mathcal{V}(W)(\textsc{urwlx}, \textsc{directed}, b, e, b+1)$$

$$\mathcal{V}(W)(retv) * \mathsf{MonoReq}(\mathsf{W}, \mathcal{V}, retv, \sqsupseteq^{b})$$

Need to establish:

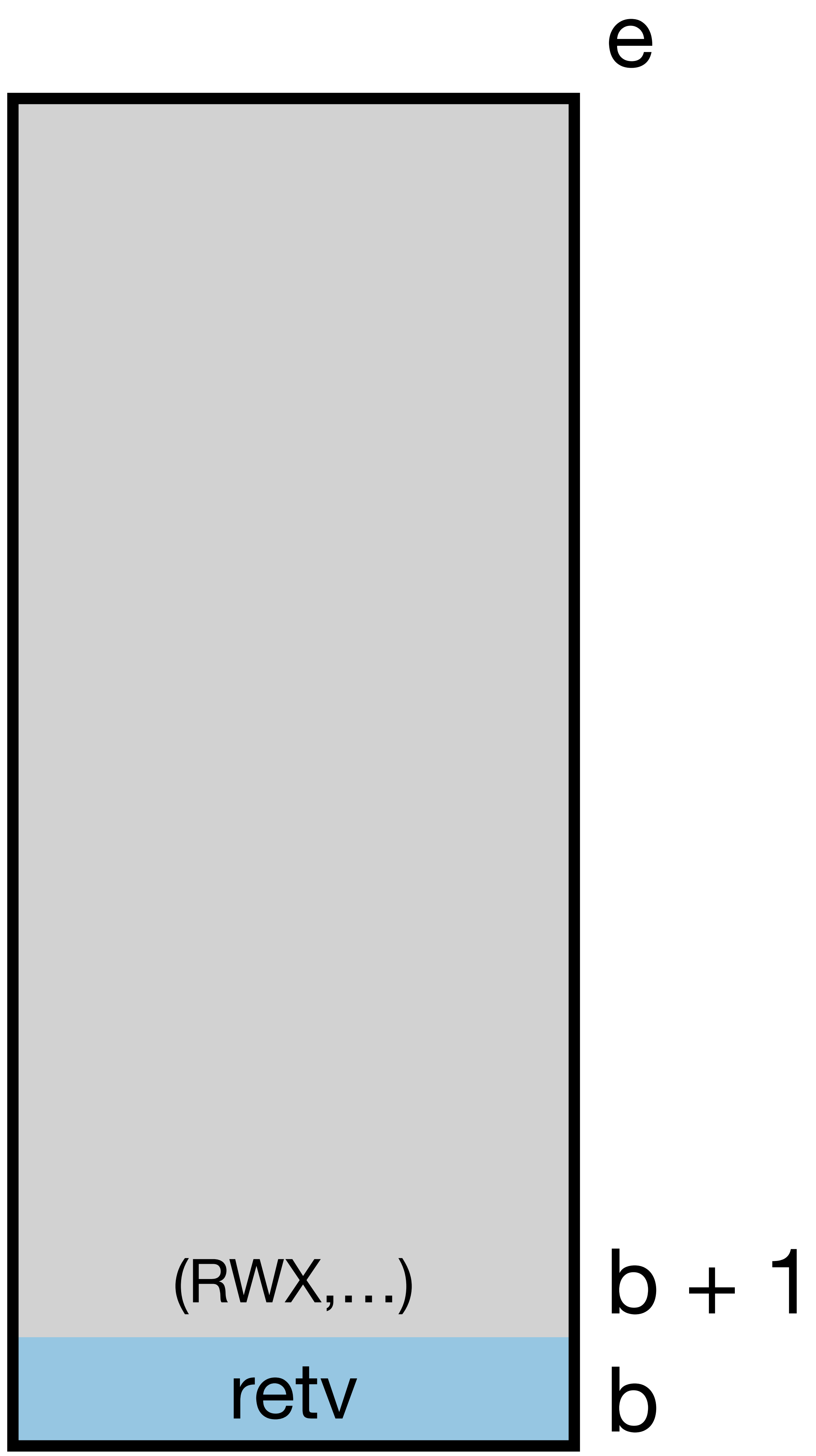
$$\mathcal{V}([b+1 := \mathsf{Uninitialized}(\textsc{rwx}, \cdots)]W)(retv)$$

We begin with:

$$stsCollection(W)$$
$$sharedResources(W)$$

We end with:

$$stsCollection([b+1 := \mathsf{Uninitialized}(\textsc{rwx}, \cdots)]W)$$
$$sharedResources([b+1 := \mathsf{Uninitialized}(\textsc{rwx}, \cdots)]W)$$



Conclusion

Summary of the Mechanized Verification

- Unary logical relation
  - Parametrized by a Kripke world to distinguish between valid heap and valid stack capabilities
  - A new kind of temporal transition to changes that may be safely observed only by the relative callers
  - A relative future world relation $\sqsupseteq^{a}$

Final Remarks

Are directed capabilities feasible?

- Uninitialized directed capabilities require only two additional bits
  - **CHERI concentrate** [Woodruff et. al. 2019] employs a rigorous compression scheme that reserves 2 and 7 bits in the CHERI-64 and CHERI-128 compression formats
- The semantics of load(U), store(U) and lea require additional bounds checks, however these bounds checks are in the same style as existing ones, and the same optimisation patters ought to apply
- The calling convention uses no stack clearing at all!

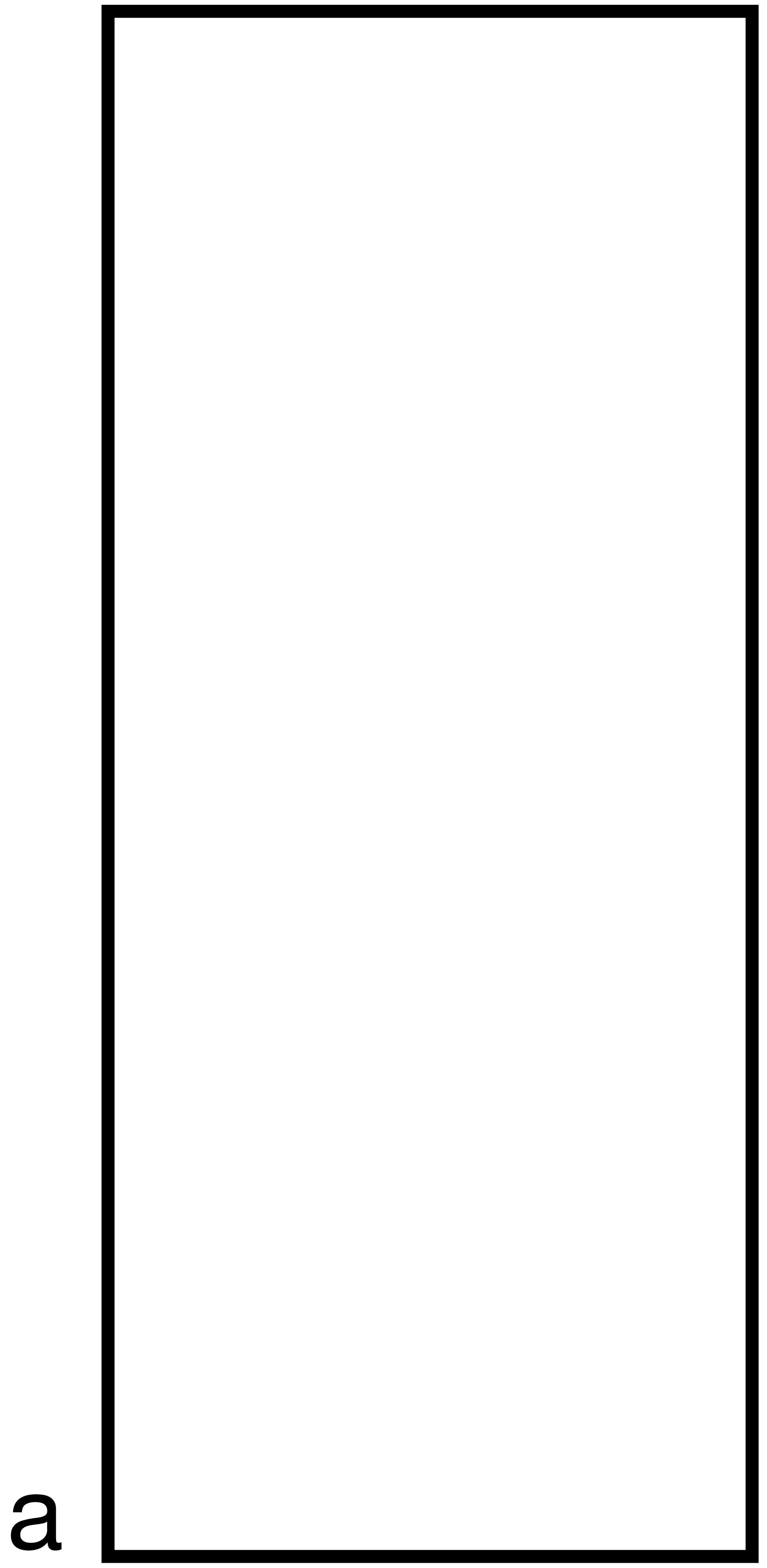
Thank you!

Final Remarks - Metrics

- In total: around 60,000 LOC, where 10,000 are for the overlay semantics and FA proof, and 14,000 is the binary model
- Around 1.5 to 2 hours to compile
- There is room for improvement!
  - Alternatives to carrying around the Kripke World
  - Using the new SSWP to get single atomic steps for the program logic

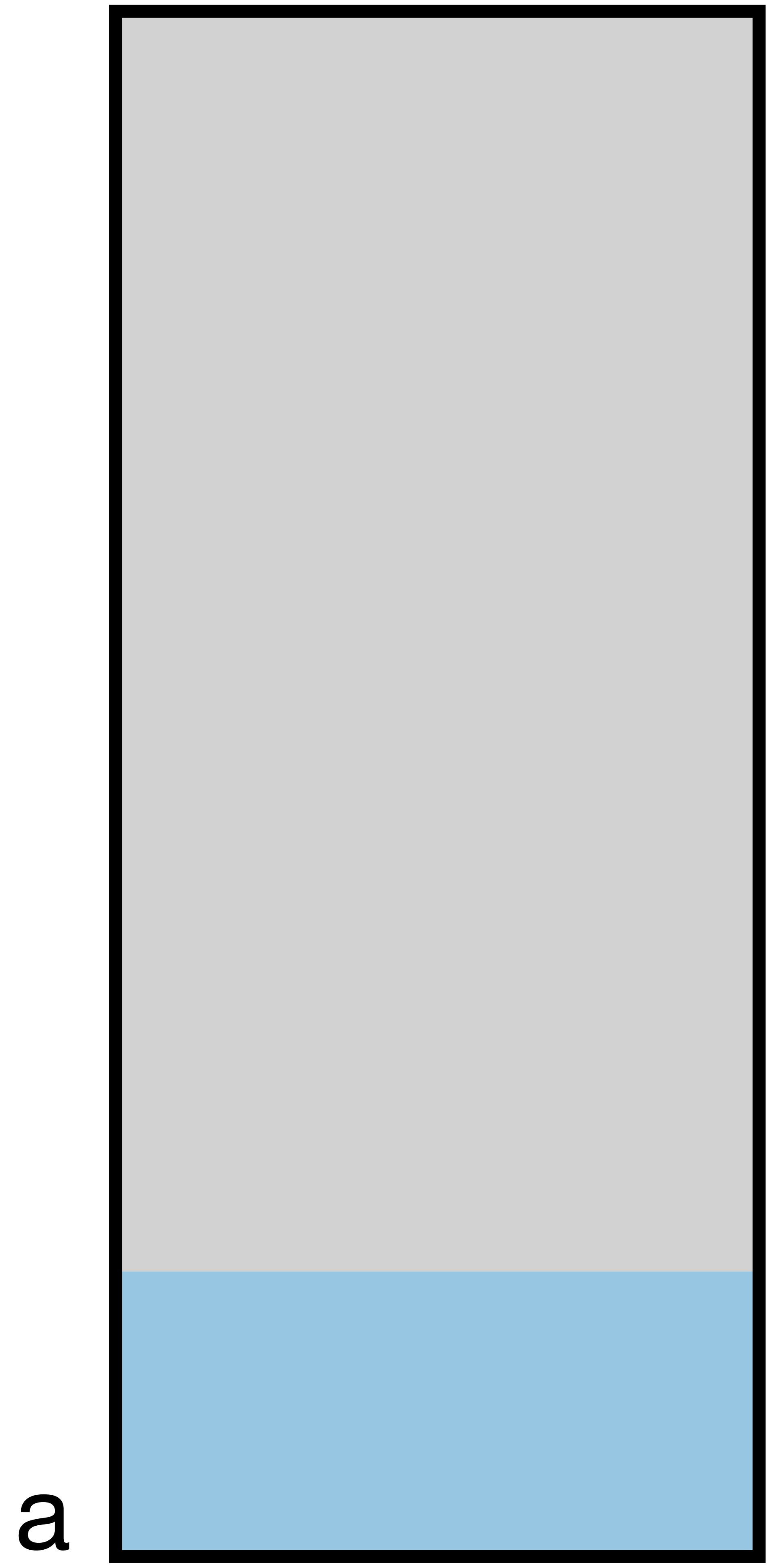
Modelling Lifetime Behaviour of Stack and Heap

Example



Modelling Lifetime Behaviour of Stack and Heap

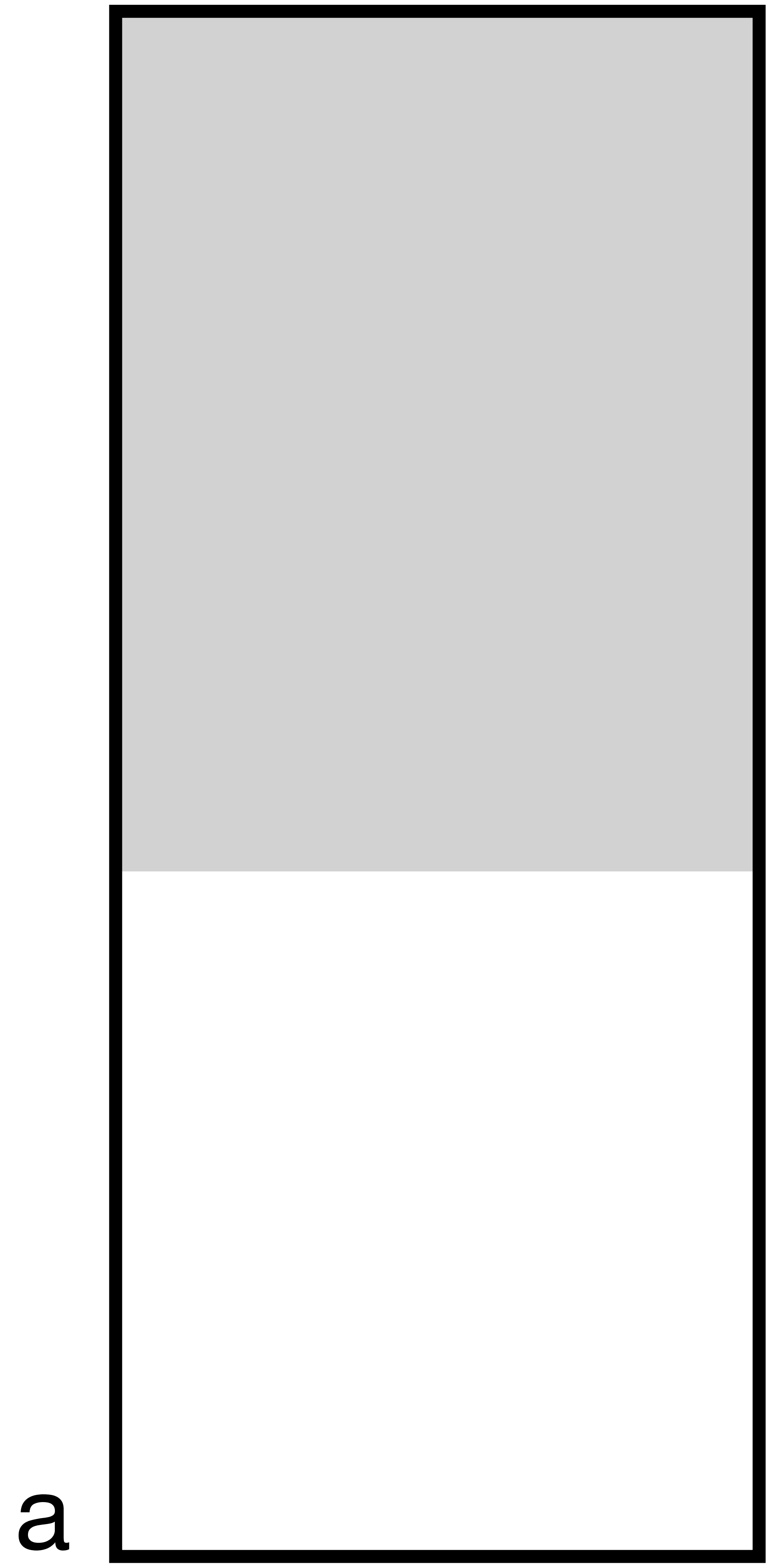
Example



- An **uninitialized** stack, with **temporary** parameters at the bottom

Modelling Lifetime Behaviour of Stack and Heap

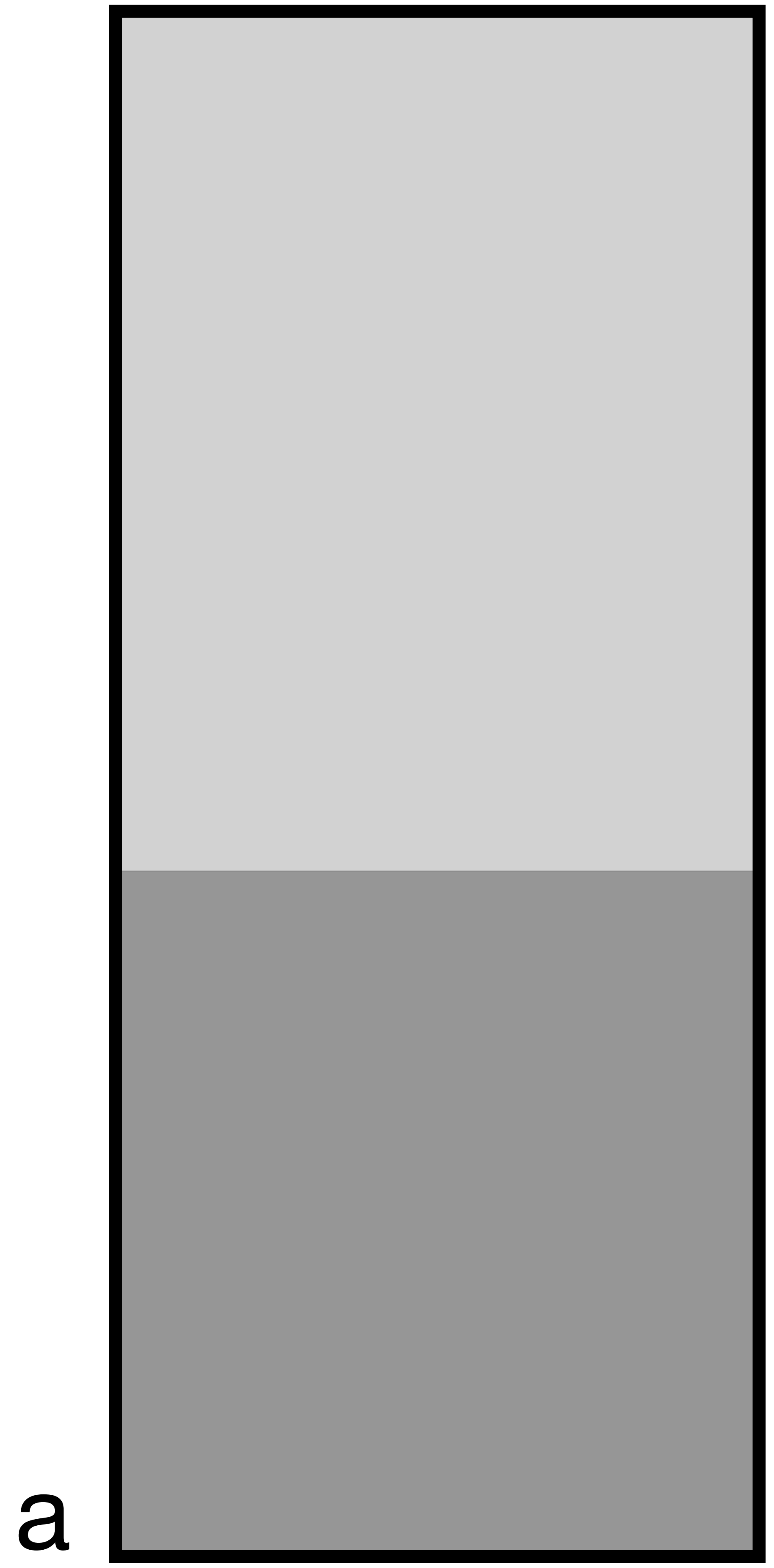
Example



- An **uninitialized** stack, with **temporary** parameters at the bottom
- We claim ownership of the stack and change the state of our stack frame

Modelling Lifetime Behaviour of Stack and Heap

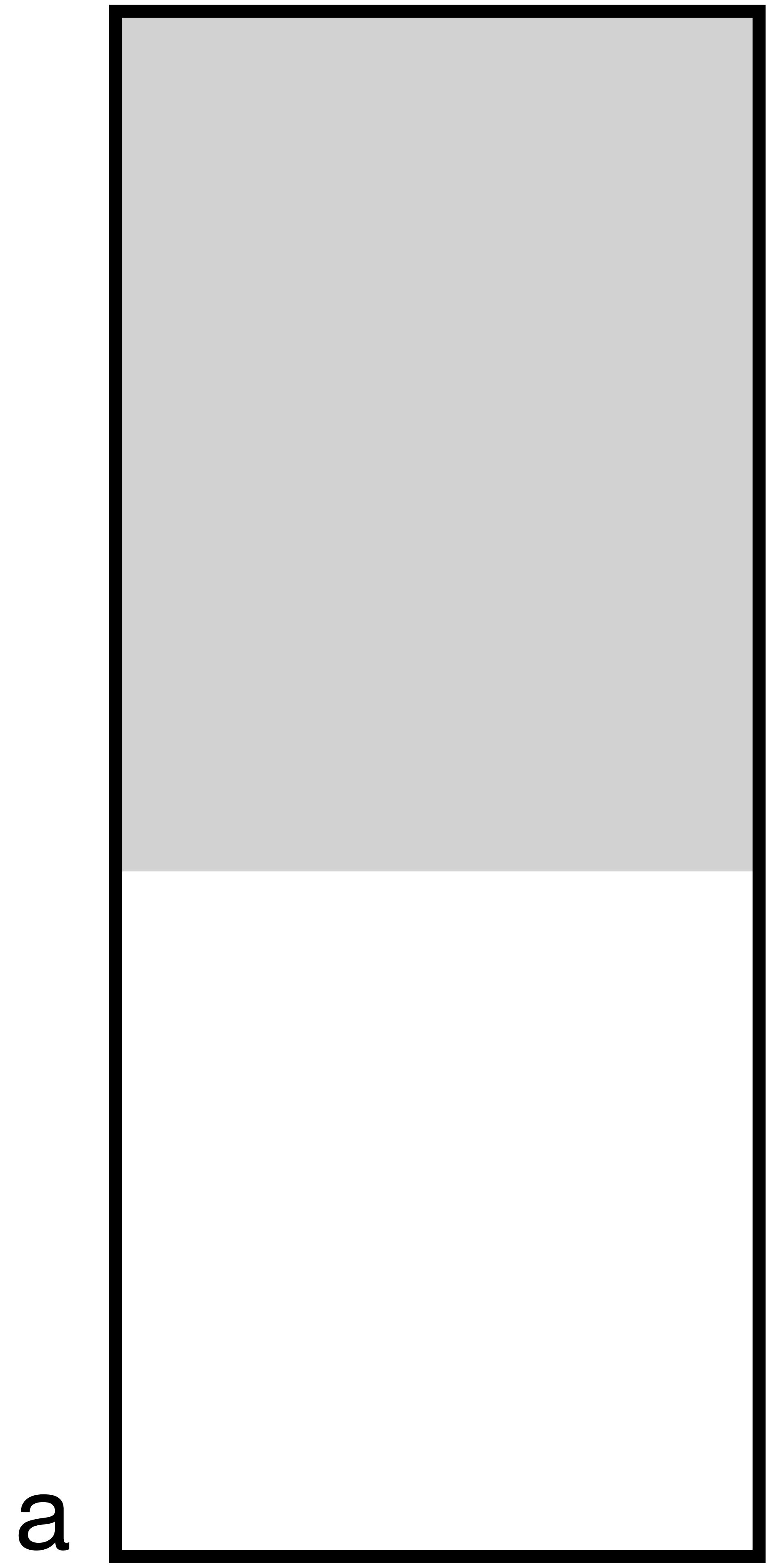
Example



- An **uninitialized** stack, with **temporary** parameters at the bottom
- We claim ownership of the stack and change the state of our stack frame
- We freeze the lower stack frame and call a new callee

Modelling Lifetime Behaviour of Stack and Heap

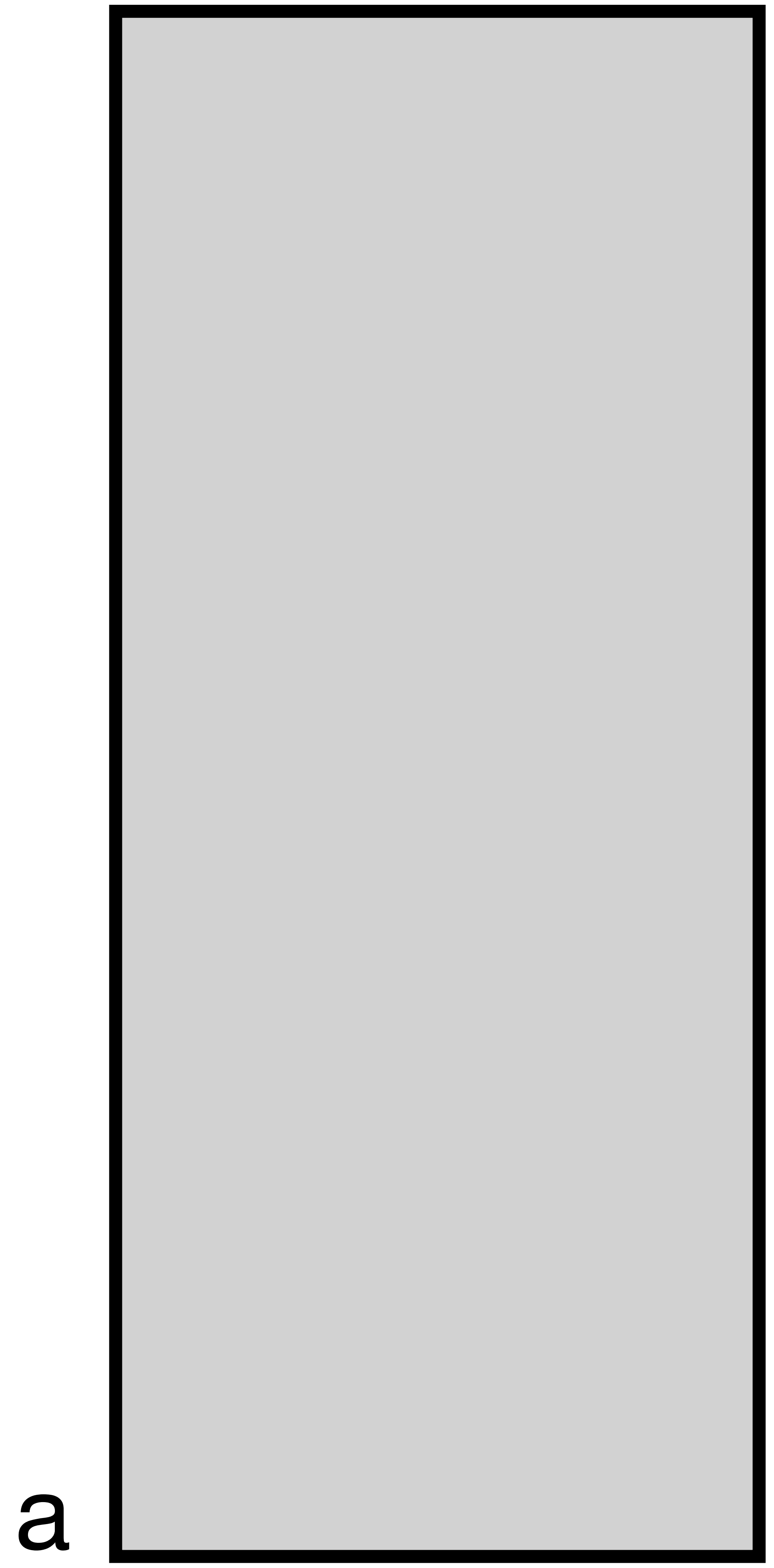
Example



- An **uninitialized** stack, with **temporary** parameters at the bottom
- We claim ownership of the stack and change the state of our stack frame
- We freeze the lower stack frame and call a new callee
- Upon return: we "thaw" the frozen frame, and pop it

Modelling Lifetime Behaviour of Stack and Heap

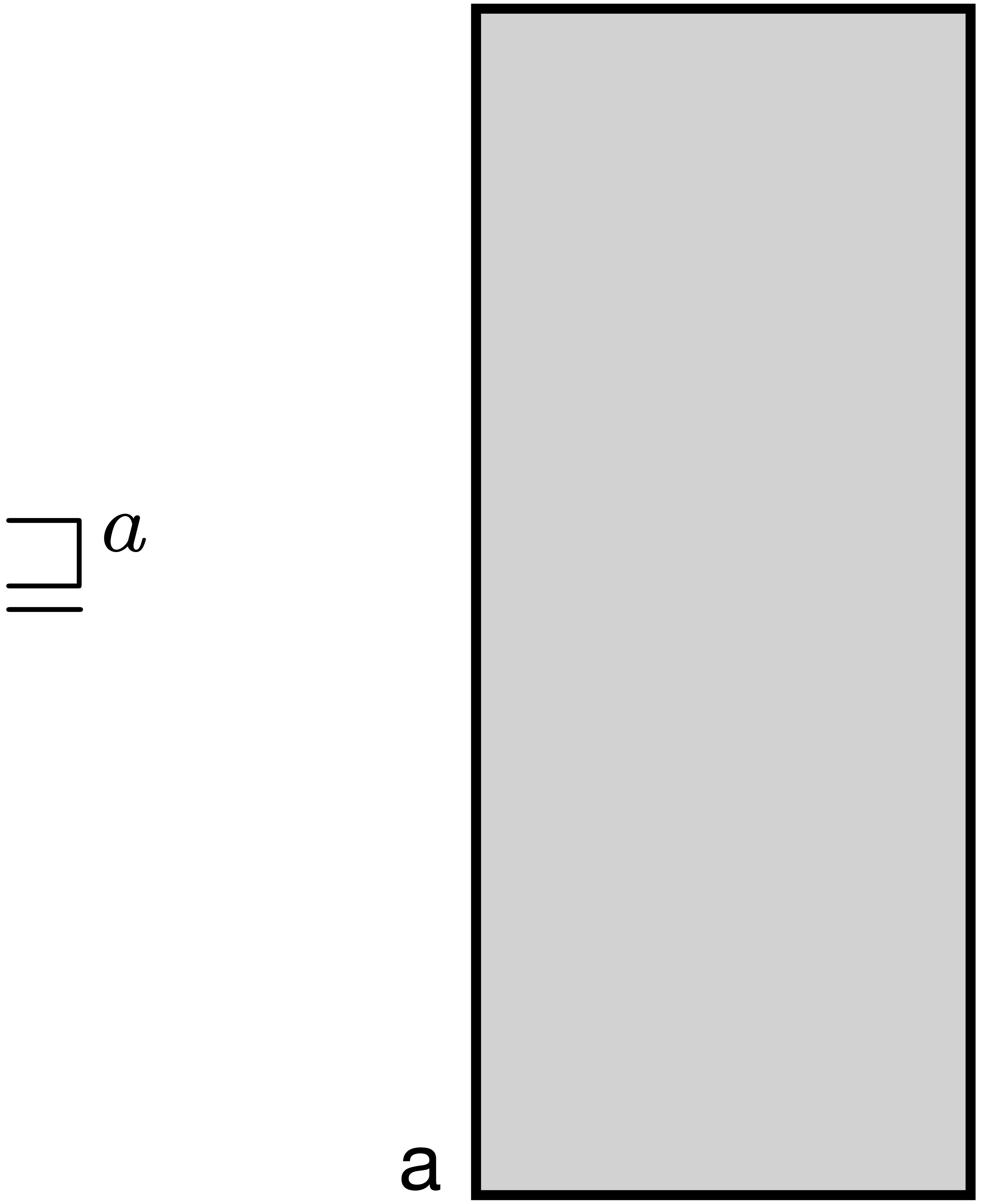
Example



- An **uninitialized** stack, with **temporary** parameters at the bottom
- We claim ownership of the stack and change the state of our stack frame
- We freeze the lower stack frame and call a new callee
- Upon return: we “thaw” the frozen frame, and pop it

Modelling Lifetime Behaviour of Stack and Heap

Example



- An **uninitialized** stack, with **temporary** parameters at the bottom
- We claim ownership of the stack and change the state of our stack frame
- We freeze the lower stack frame and call a new callee
- Upon return: we “thaw” the frozen frame, and pop it

The Instrumented Machine State

Standard resources

a0	Temporary
a1	Uninitialized(w)
...	...

Standard map

a0	γ_0
a1	γ_1
...	...

Interpretation map

$\gamma_0 \Rrightarrow \phi_0$

$\gamma_1 \Rrightarrow \phi_1$

WORLD $\to$ Word $\to$ $iProp$

- stsCollection(W) : the authoritative view of the standard map
- sharedResources(W) : the authoritative view of the interpretation map, AND the *standard resource* for each address in the map, according to its standard state
- rel(a,Φ) : the fragmental view of the association between a and Φ in the interpretation map

Back to the Logical Relation

Associating memory invariants with a standard state

$$\mathcal{V}(\text{RWLX}, \text{DIRECTED}, b, e, -) \triangleq \mathop{\Large\ast}_{a \in [b,e)} \exists P, \boxed{\exists w\ \sigma, a \mapsto w * \textsf{state}\ \sigma * P(\sigma, w)}^{\mathcal{N}.a}$$

$$* \triangleright \square \forall \sigma\ w, P(\sigma, w) \twoheadrightarrow \sigma = \textsf{Temporary} * \mathcal{V}(w)$$

Back to the Logical Relation

Associating memory invariants with a standard state

$$\mathcal{V}(\text{RWLX}, \text{DIRECTED}, b, e, -) \triangleq \mathop{\Huge *}_{a \in [b,e)} \exists P, \boxed{\exists w\ \sigma, a \mapsto w * \textsf{state}\ \sigma * P(\sigma, w)}^{\mathcal{N}.a}$$

$$* \triangleright \square \forall \sigma\ w, P(\sigma, w) \twoheadrightarrow \sigma = \textsf{Temporary} * \mathcal{V}(w)$$

$$\mathcal{V}(\text{E}, \text{GLOBAL}, \cdots) \triangleq \square \triangleright \mathcal{E}(\text{RX}, \text{GLOBAL}, \cdots)$$

$$\mathcal{V}(\text{E}, \text{DIRECTED}, \cdots) \triangleq \square \triangleright \mathcal{E}(\text{RX}, \text{DIRECTED}, \cdots)$$

Back to the Logical Relation

Associating memory invariants with a standard state

$$\mathcal{V}(\text{RWLX}, \text{DIRECTED}, b, e, -) \triangleq \mathop{\Large *}_{a \in [b,e)} \exists P, \boxed{\exists w\ \sigma, a \mapsto w * \mathsf{state}\ \sigma * P(\sigma, w)}^{\mathcal{N}.a}$$

$$* \triangleright \Box \forall \sigma\ w, P(\sigma, w) \mathrel{-\!\!*} \sigma = \mathsf{Temporary} * \mathcal{V}(w)$$

$$\mathcal{V}(\text{E}, \text{GLOBAL}, \cdots) \triangleq \Box \triangleright \mathcal{E}(\text{RX}, \text{GLOBAL}, \cdots)$$

$$\mathcal{V}(\text{E}, \text{DIRECTED}, \cdots) \triangleq \Box \triangleright \mathcal{E}(\text{RX}, \text{DIRECTED}, \cdots)$$

How to distinguish between the two?