Modular Reasoning about Separation for Concurrent Data Structures

Kasper Svendsen, Lars Birkedal, Matthew Parkinson

March 20, 2013

Modular Reasoning in SL

Basic separation logic assertions (resources) consists of

- **information** about some part of the heap
- and **rights** to modify this part of the heap

For instance, $x.f \mapsto v$ asserts

- that the field $x.f$ contains the value v
- and the **exclusive** right to modify this field

Rights implicitly restricts how the **environment** is allowed to modify the heap and what the owner can know about the current state of the heap

Modular Reasoning about ADTs in SL

To lift this to ADTs we need **ADT resources** with rights expressed in terms of the abstraction provided by the ADT rather than the underlying data representation.

Let $\mathit{htable}(x, f)$ represent a partial hash table resource that

- asserts **exclusive** right to modify all keys $k \in \mathit{dom}(f)$
- and that the current value of key $k \in \mathit{dom}(f)$ is $f(k)$

then

$$\mathit{htable}(x, f_1 \uplus f_2) \Leftrightarrow \mathit{htable}(x, f_1) * \mathit{htable}(x, f_2)$$

Modular Reasoning about ADTs in SL

To lift this to ADTs we need **ADT resources** with rights expressed in terms of the abstraction provided by the ADT rather than the underlying data representation.

Let $\mathit{htable}(x, f)$ represent a partial hash table resource that

- asserts **exclusive** right to modify all keys $k \in \mathit{dom}(f)$
- and that the current value of key $k \in \mathit{dom}(f)$ is $f(k)$

then

$$\mathit{htable}(x, f_1 \uplus f_2) \Leftrightarrow \mathit{htable}(x, f_1) * \mathit{htable}(x, f_2)$$

Related work

Fictional Separation Logic [Jensen 2012],
Superficially Substructural Types [Krishnaswami 2012]

- Client defines a monoid on abstract state
- **Sequential setting** without reentrancy

Concurrent Abstract Predicates [Dinsdale-Young 2010]

- Shared regions with protocols governing sharing
- **Module implementers** can construct ADT resources
- First-order, concurrent setting

Our contribution

We present a **new separation logic and specification pattern** that allow **clients** to construct ADT resources, in a **concurrent** higher-order setting with reentrancy

The logic combines and extends a lot of previous work

- Higher-order Separation Logic
- Concurrent Abstract Predicates
- Guarded Recursion

Our contribution

We present a **new separation logic and specification pattern** that allow **clients** to construct ADT resources, in a **concurrent** higher-order setting with reentrancy

The logic combines and extends a lot of previous work

- Higher-order Separation Logic
- Concurrent Abstract Predicates
- Guarded Recursion

Bag example

```
interface Bag {
  public Bag();
  public void Push(Object obj);
  public Object Pop();
}
```

Bag example

Sequential Specification

$$\{\mathbf{emp}\} \quad \texttt{new Bag()} \quad \{\mathbf{ret}.\ \mathbf{bag}_e(\mathbf{ret}, \emptyset)\}$$

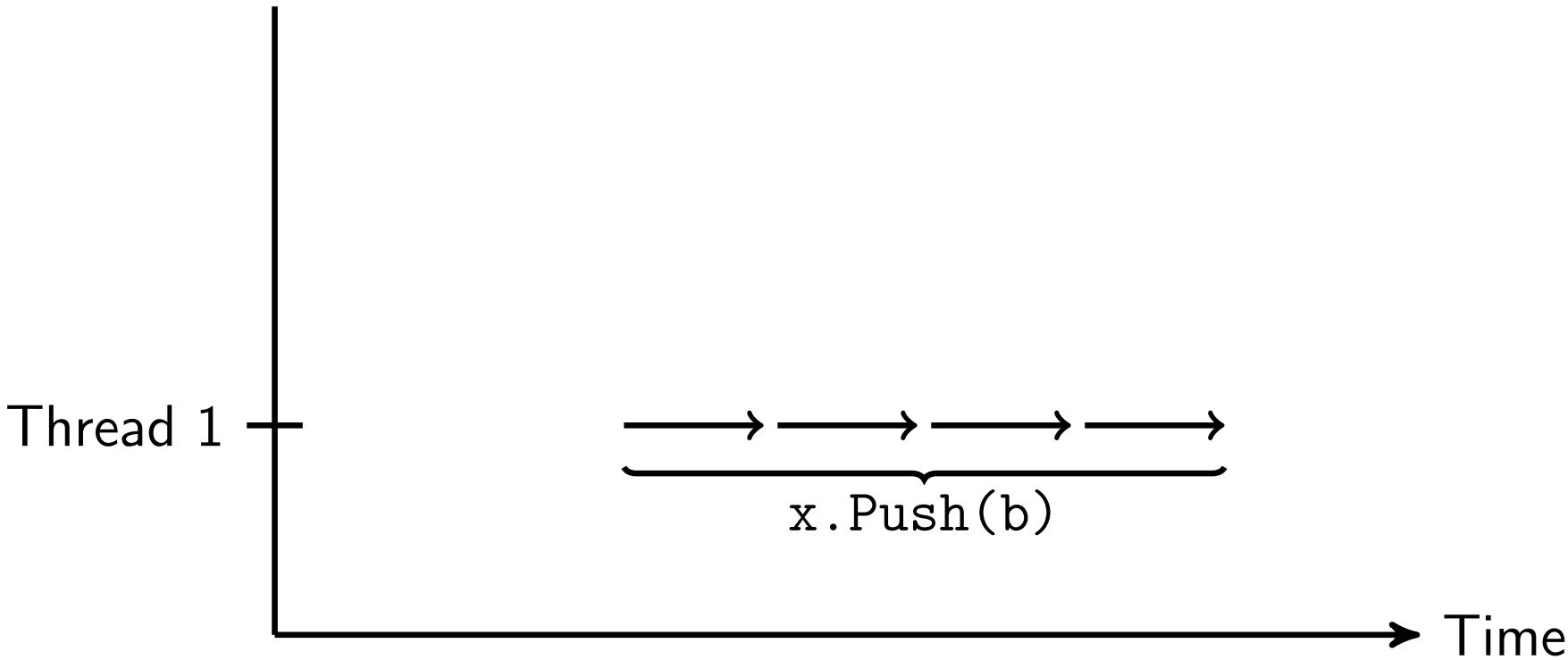
$$\{\mathbf{bag}_e(\mathbf{this}, A)\} \quad \texttt{Push(x)} \quad \{\mathbf{bag}_e(\mathbf{this}, A \cup \{x\})\}$$

$$\{\mathbf{bag}_e(\mathbf{this}, \emptyset)\} \quad \texttt{Pop()} \quad \{\mathbf{ret}.\ \mathbf{bag}_e(\mathbf{this}, \emptyset) * \mathbf{ret} = \mathbf{null}\}$$

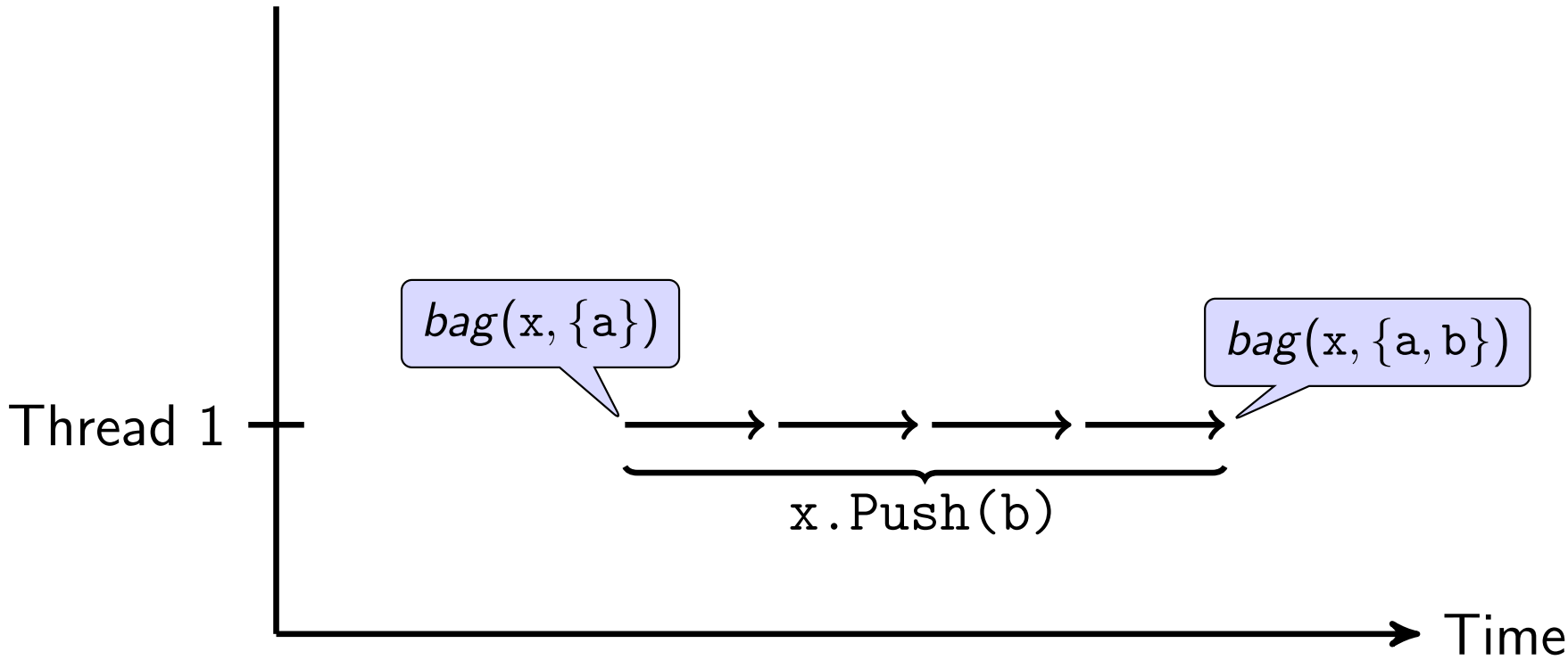
$$\{\mathbf{bag}_e(\mathbf{this}, A) * A \neq \emptyset\} \quad \texttt{Pop()} \quad \{\mathbf{ret}.\ \mathbf{ret} \in A * \mathbf{bag}_e(A \setminus \{\mathbf{ret}\})\}$$

$$\mathbf{bag}_e(x, A) * \mathbf{bag}_e(x, B) \Rightarrow \bot$$

Bag example

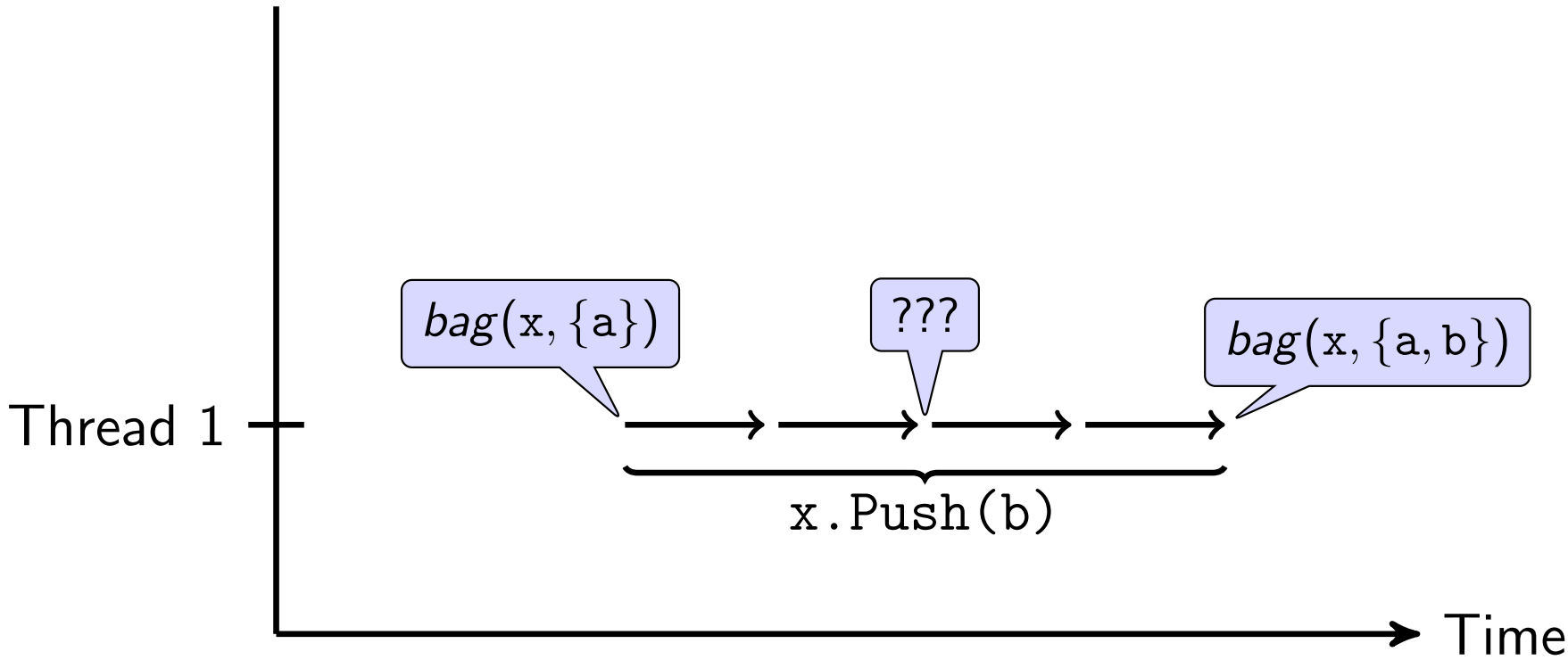


Bag example



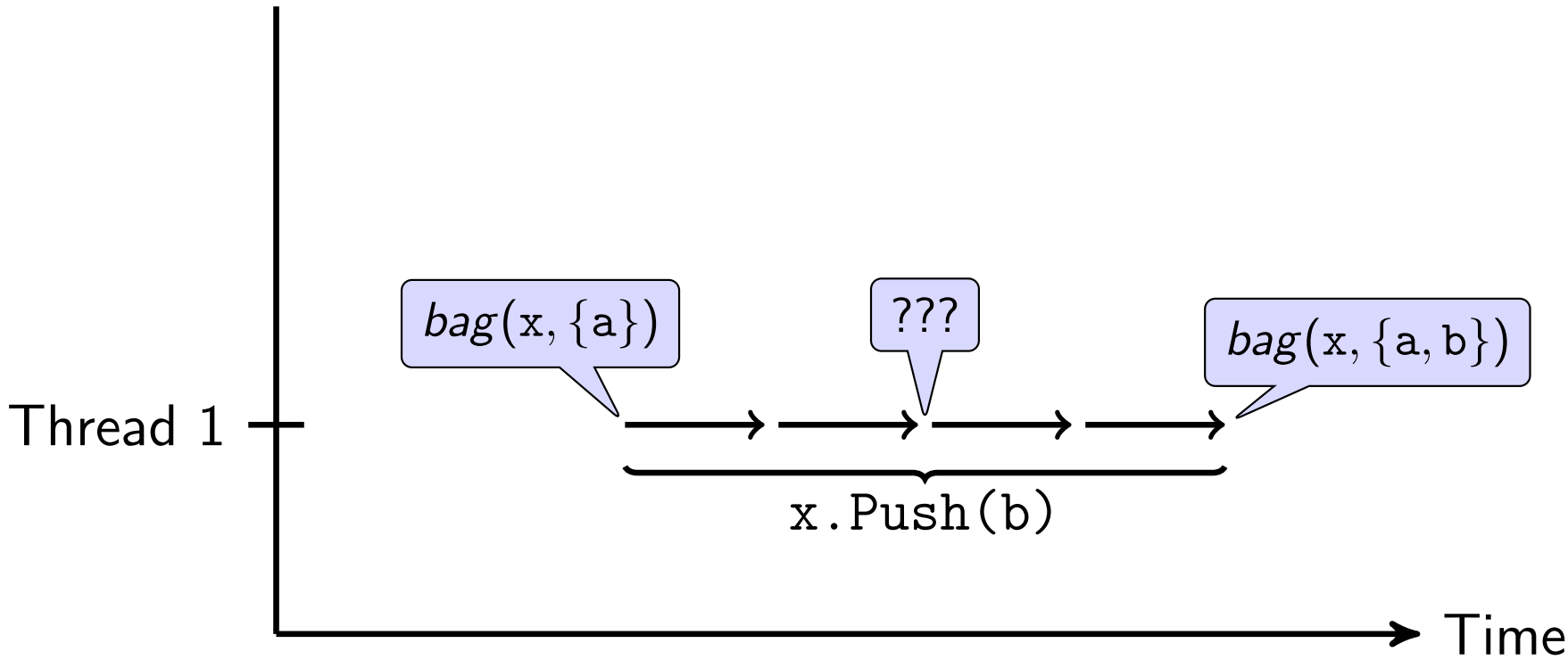
The sequential spec. just relates the initial and terminal state.

Bag example



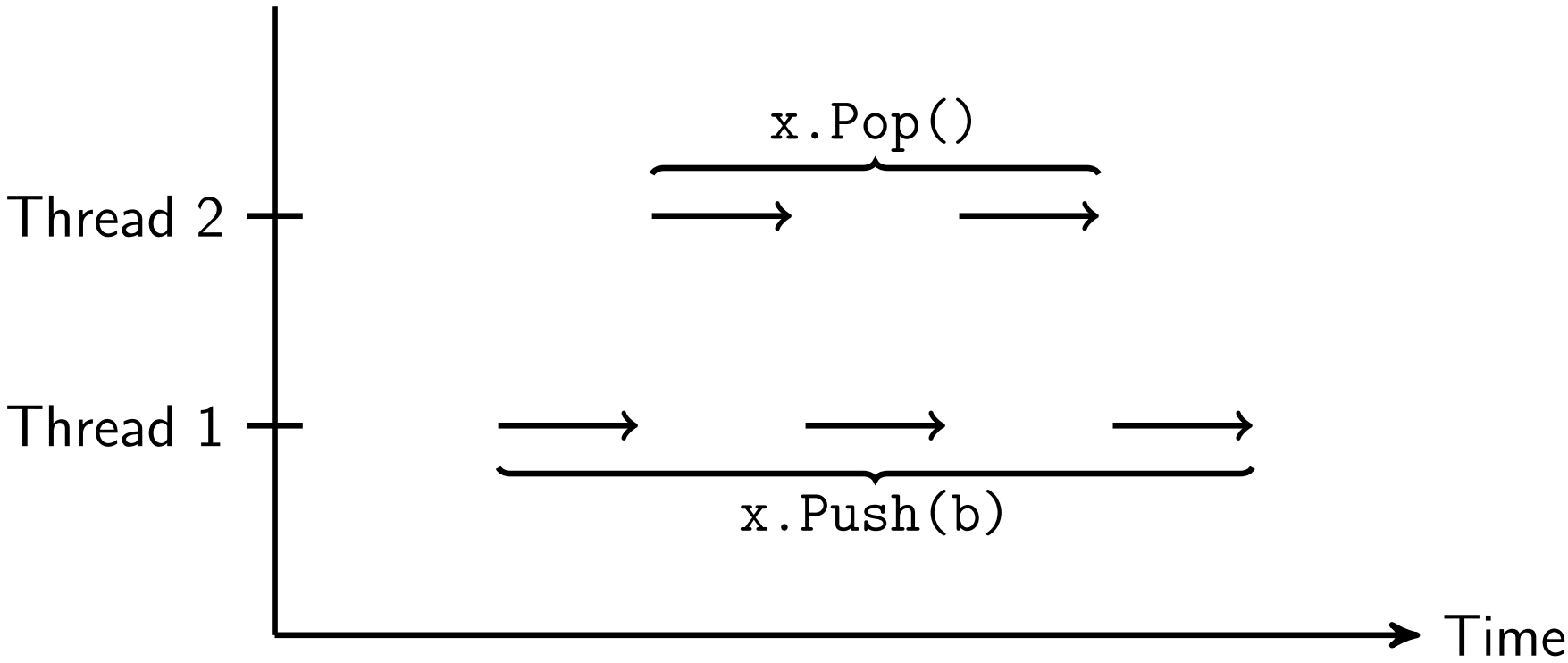
The sequential spec. just relates the initial and terminal state.
It says nothing about intermediate states.

Bag example



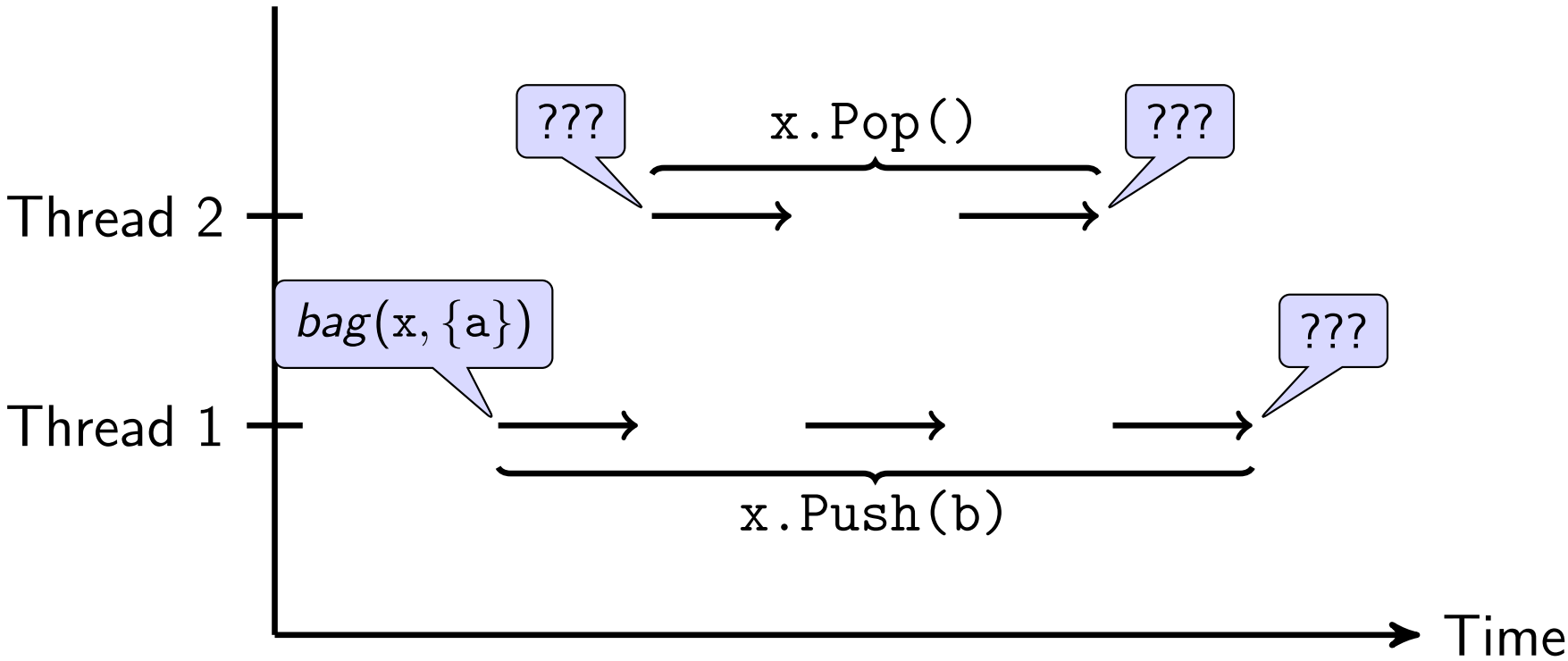
This is sufficient in a sequential setting without reentrancy, as we do not have to consider "interference" inside method calls.

Bag example



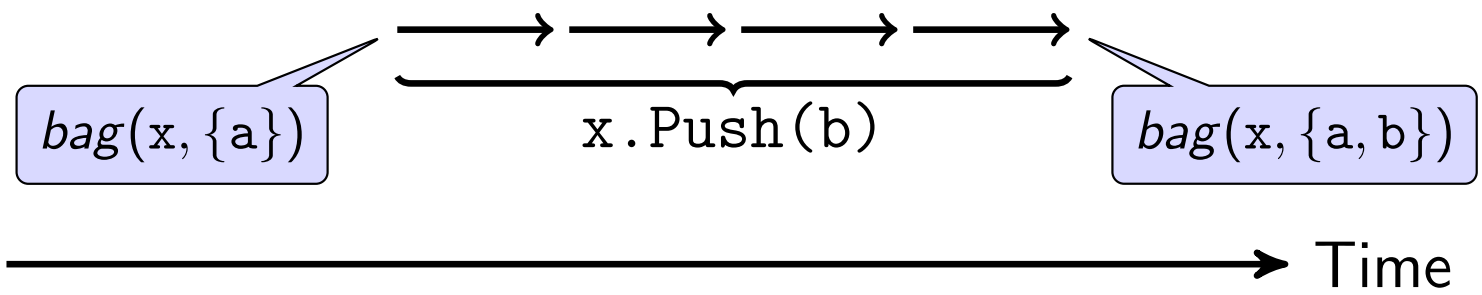
But it is **not sufficient** for a concurrent setting, as we do have to consider interference inside method calls.

Bag example



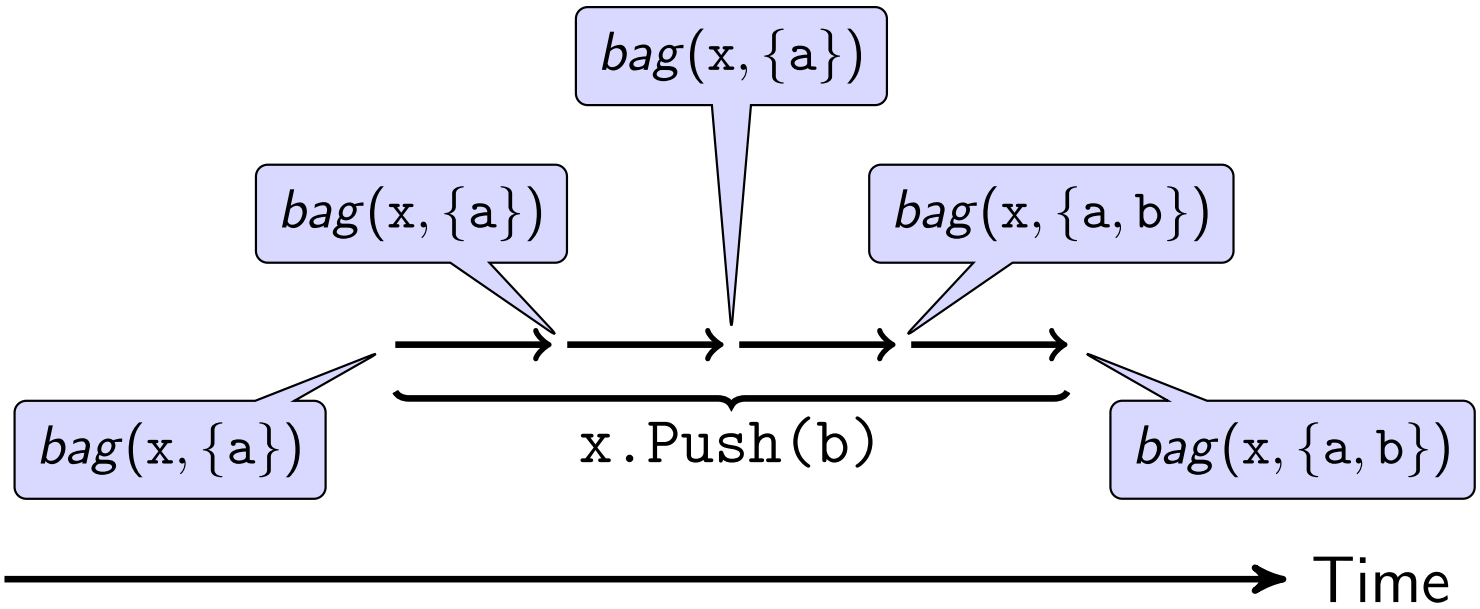
But it is **not sufficient** for a concurrent setting, as we do have to consider interference inside method calls.

Bag example



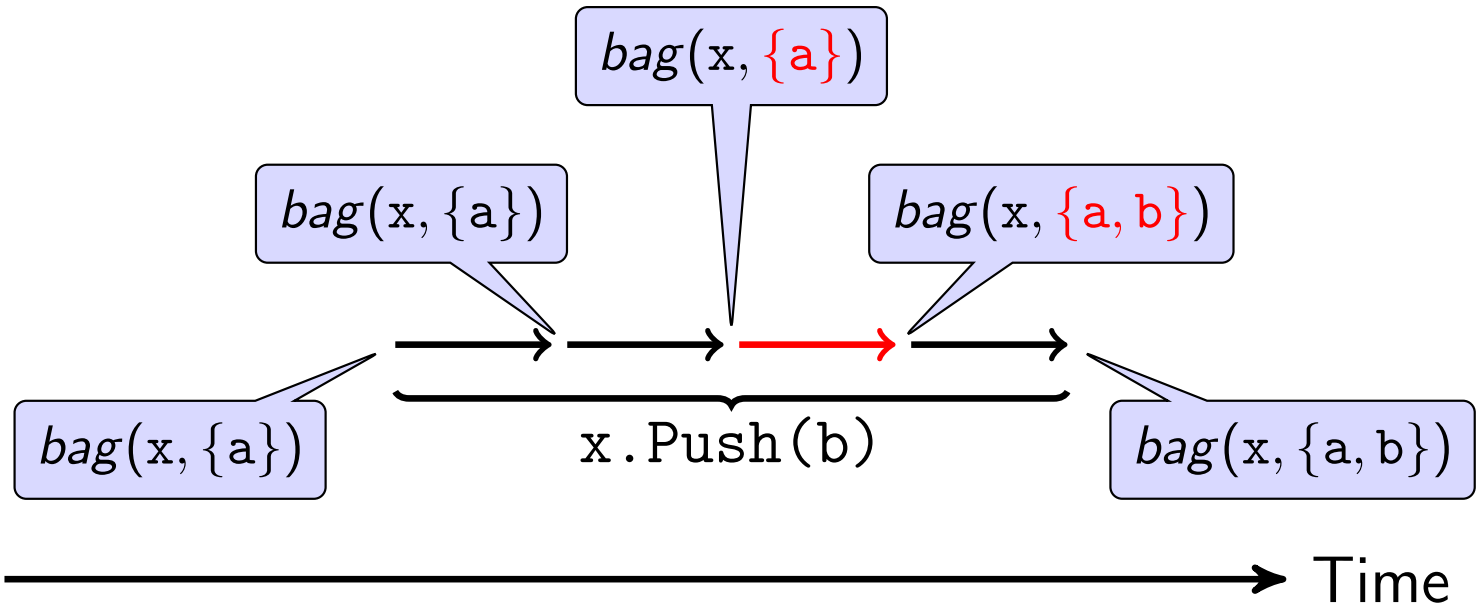
We need to restrict attention to thread-safe implementations.

Bag example



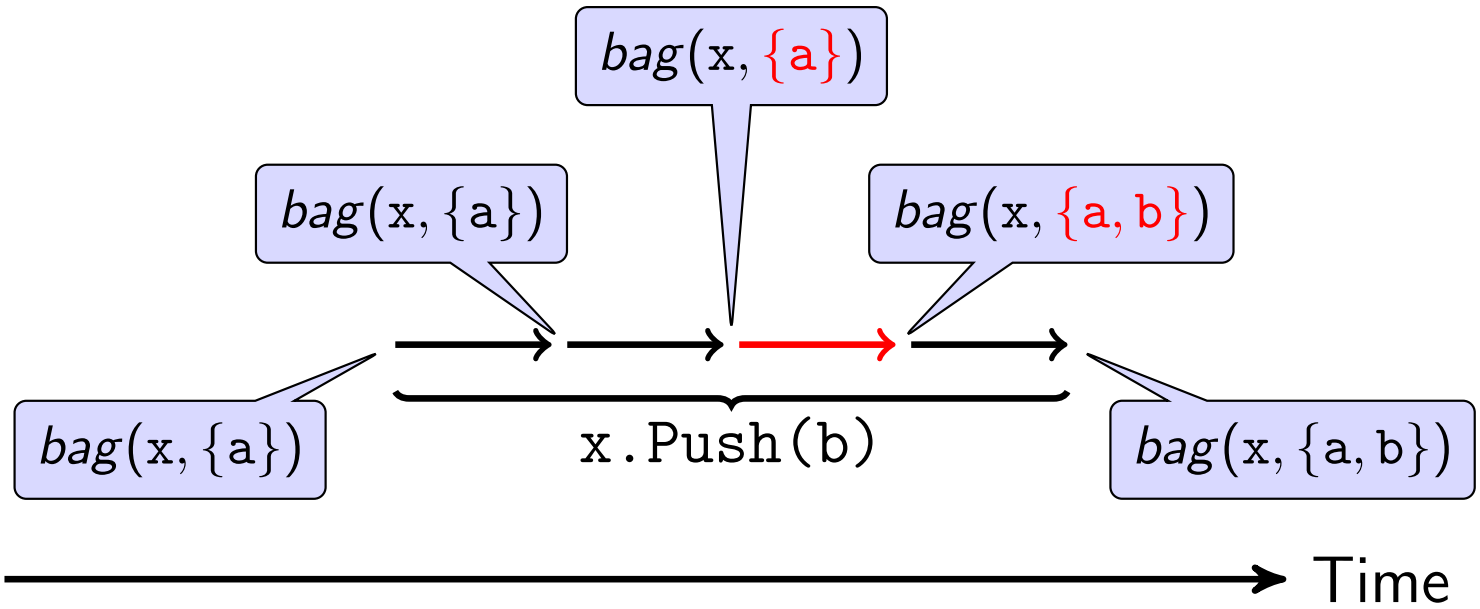
We consider a method thread-safe with respect to an abstraction if, for every intermediate state, there exists some abstract state that describes the concrete state of the ADT.

Bag example



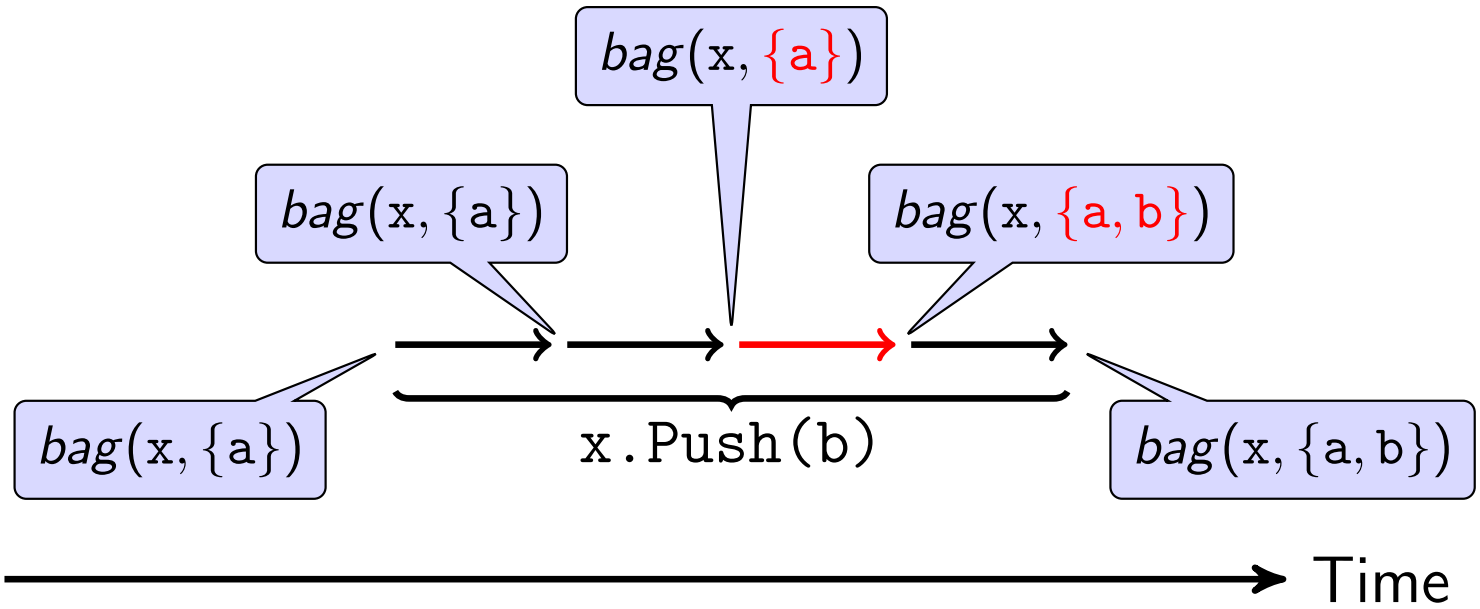
Each method execution thus contains one or more **atomic instructions** that modify the abstract state.

Bag example



Goal: A specification that allow **clients** to reason about the abstract initial and terminal state of each of these atomic instructions.

Bag example



Store the abstract state that describes the concrete state of the ADT in a **phantom field**.

Phantom fields and view-shifts

Phantom fields are a logical counterpart to auxiliary fields.

Phantom fields can be split using fractional permissions

$$x_f \overset{p+q}{\mapsto} v \Leftrightarrow x_f \overset{p}{\mapsto} v * x_f \overset{q}{\mapsto} v$$

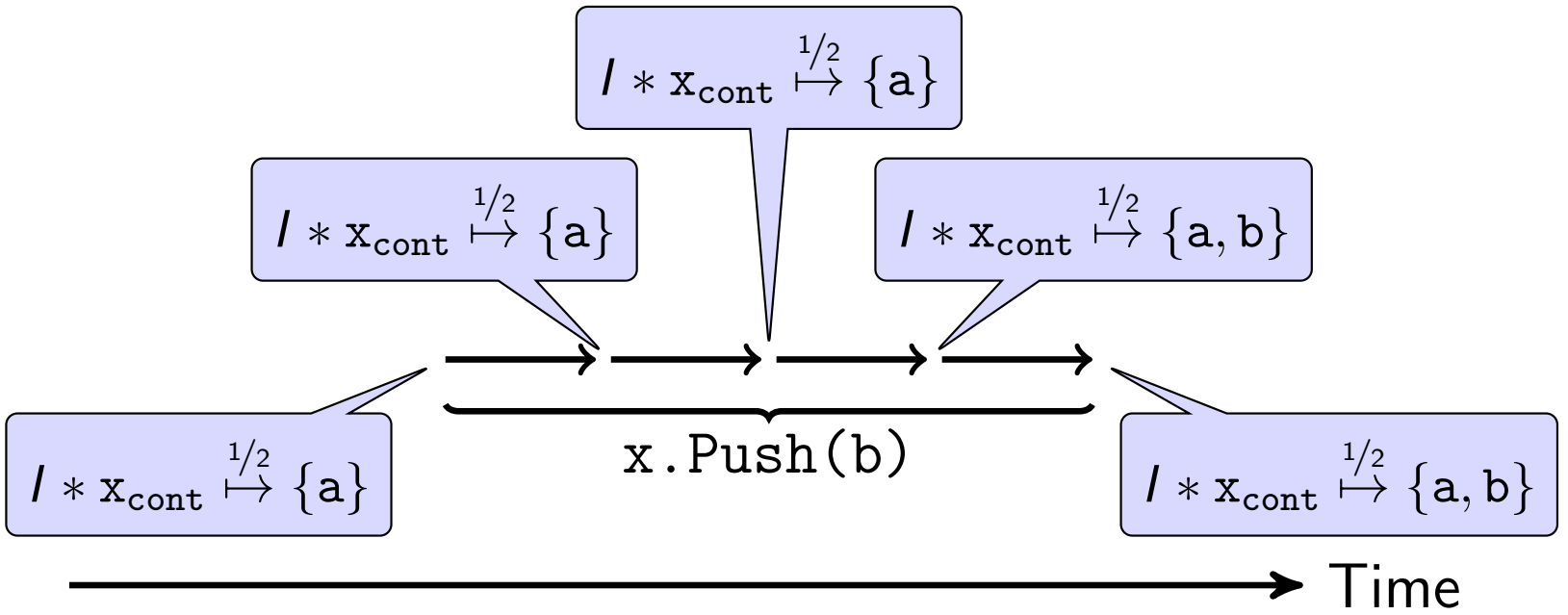
and updated through **view-shifts**:

$$x_f \overset{1}{\mapsto} v_1 \sqsubseteq x_f \overset{1}{\mapsto} v_2$$

View-shifts describe steps that do not modify the state:

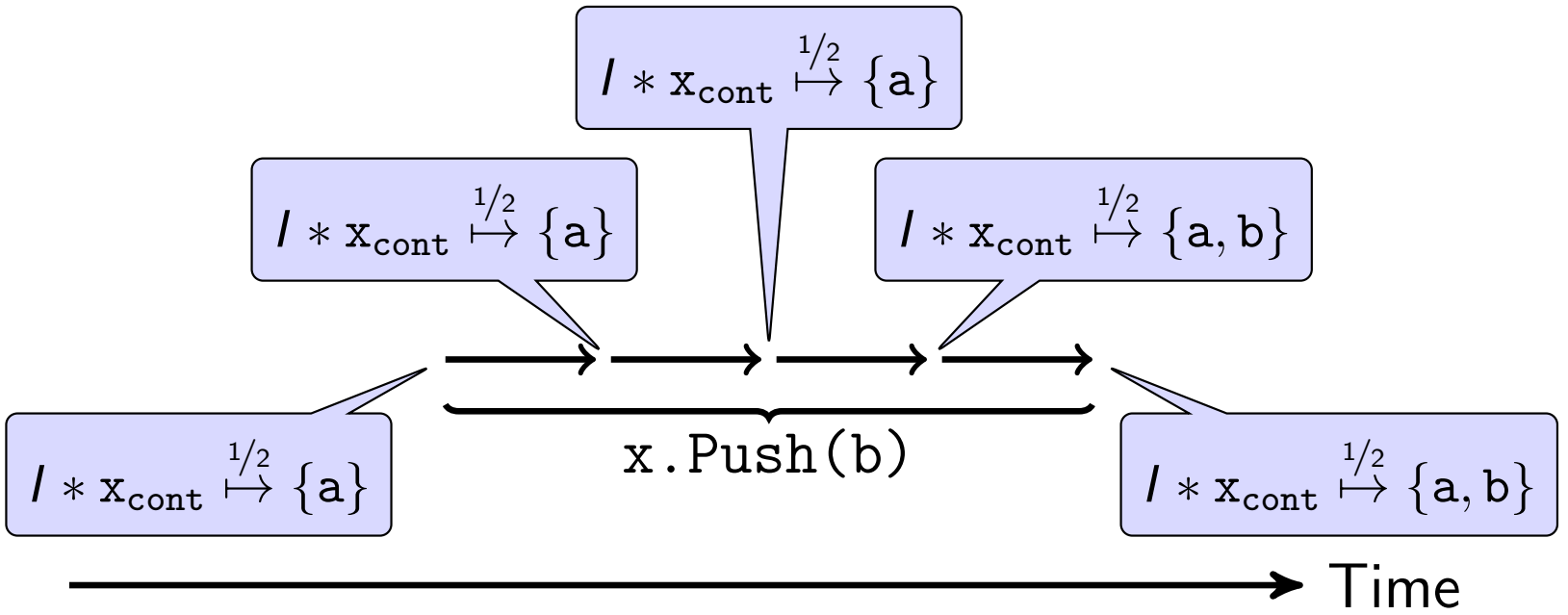
$$p \sqsubseteq q \quad \approx \quad \{p\}skip\{q\}$$

Bag example



where $I \overset{\text{def}}{=} \exists A.\ \mathbf{bag}(\mathtt{x}, A) * \mathtt{x}_{\mathtt{cont}} \overset{1/2}{\mapsto} A$

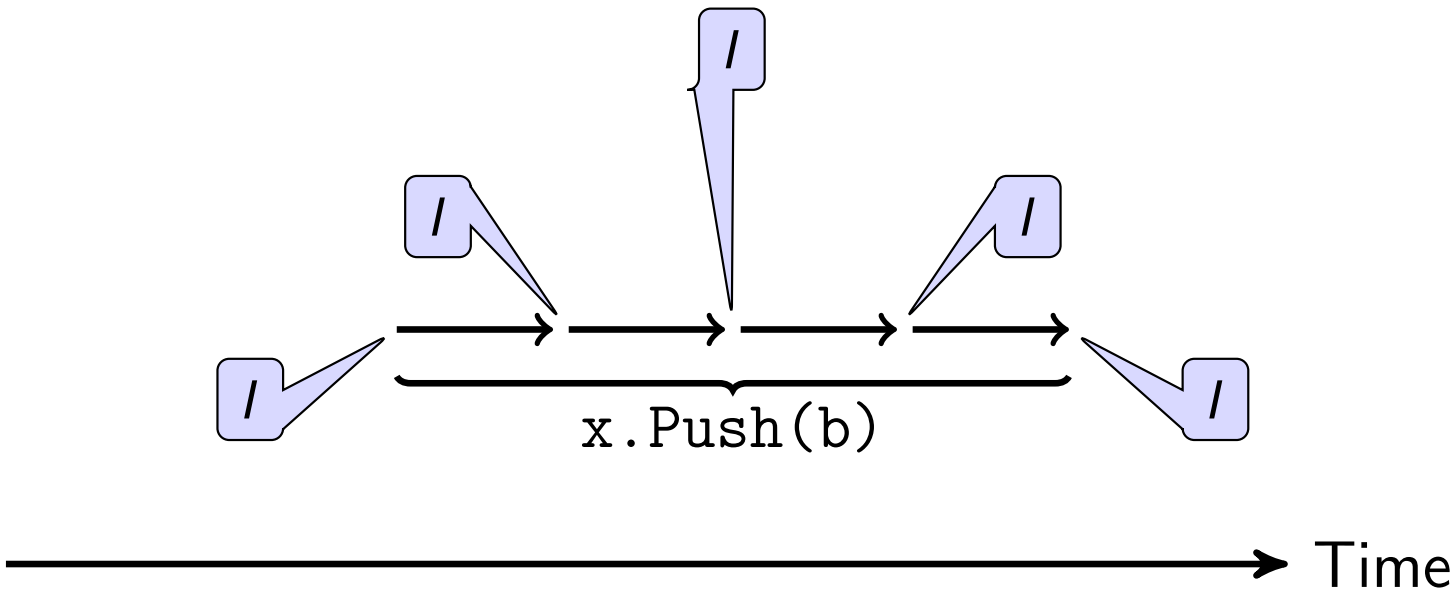
Bag example



where $I \overset{\text{def}}{=} \exists A.\ \mathbf{bag}(\mathtt{x}, A) * \mathtt{x}_{\mathtt{cont}} \overset{1/2}{\mapsto} A$

The ADT keeps half the `cont` phantom field.

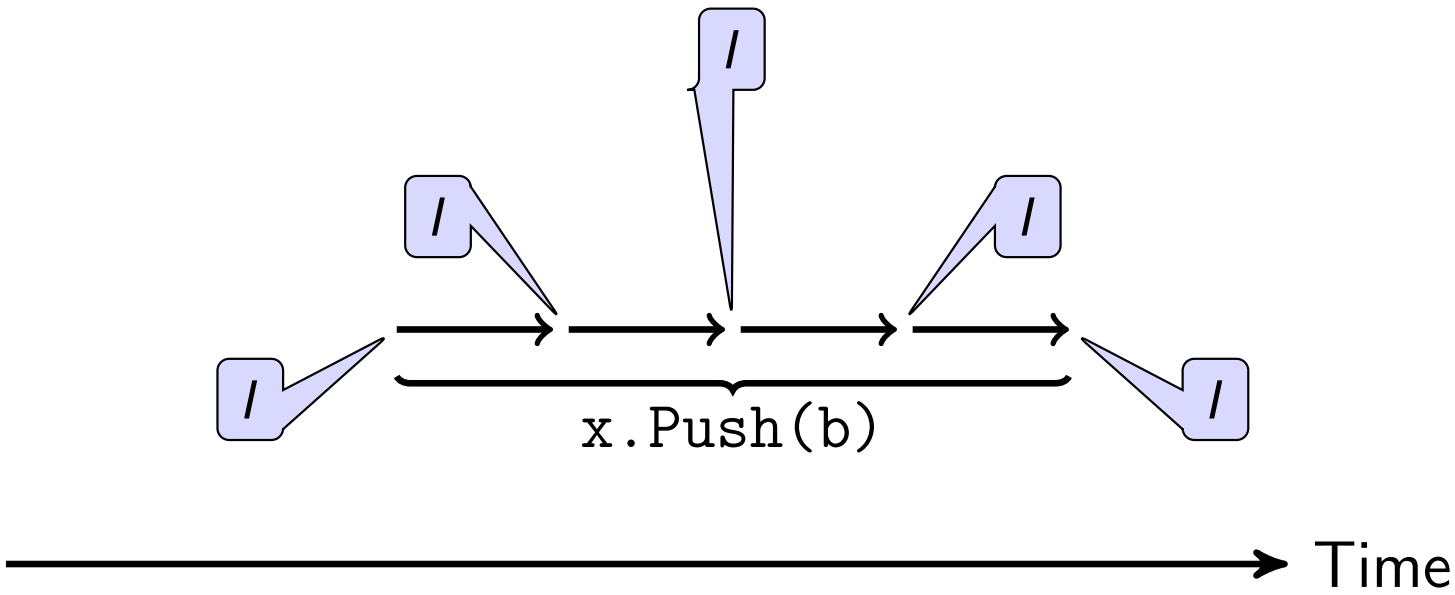
Bag example



where $I \stackrel{\text{def}}{=} \exists A.\ \mathbf{bag}(\mathtt{x}, A) * \mathtt{x}_{\mathtt{cont}} \overset{1/2}{\mapsto} A$

Clients share the other half.

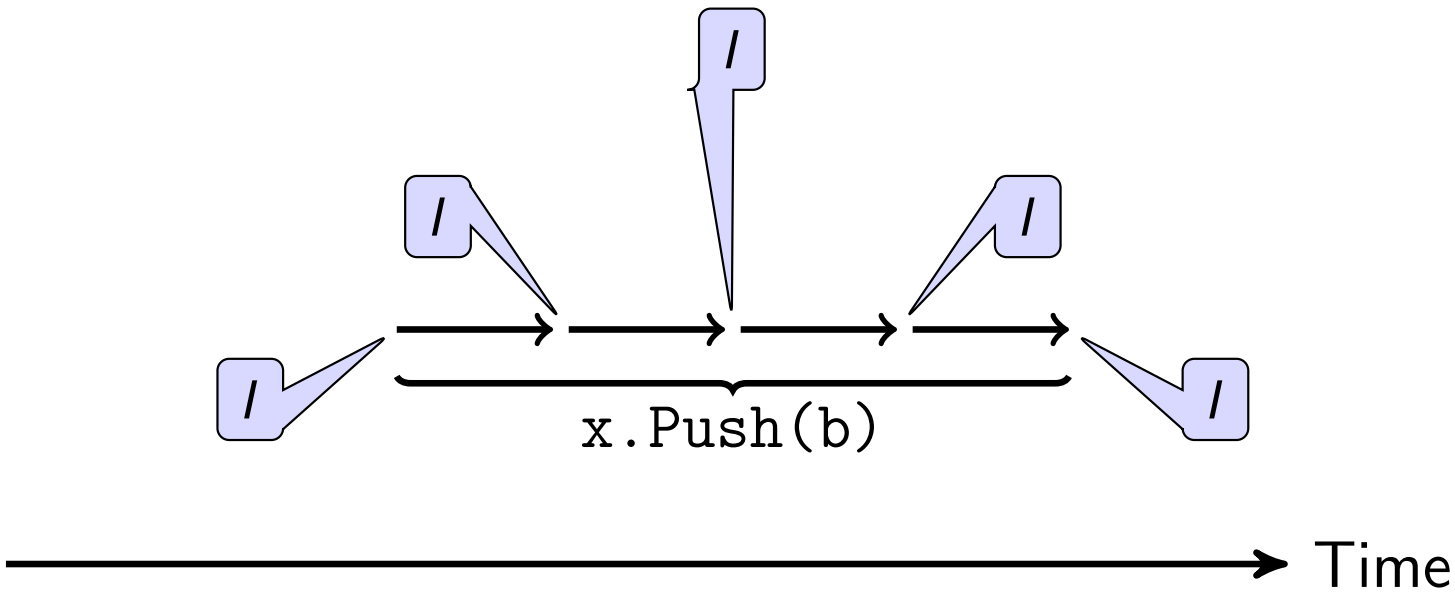
Bag example



where $I \stackrel{\text{def}}{=} \exists A.\ \mathbf{bag}(\mathtt{x}, A) * \mathtt{x}_{\mathtt{cont}} \overset{1/2}{\mapsto} A$

Idea: Let clients express abstract ADT rights as rights to update their half of the phantom field.

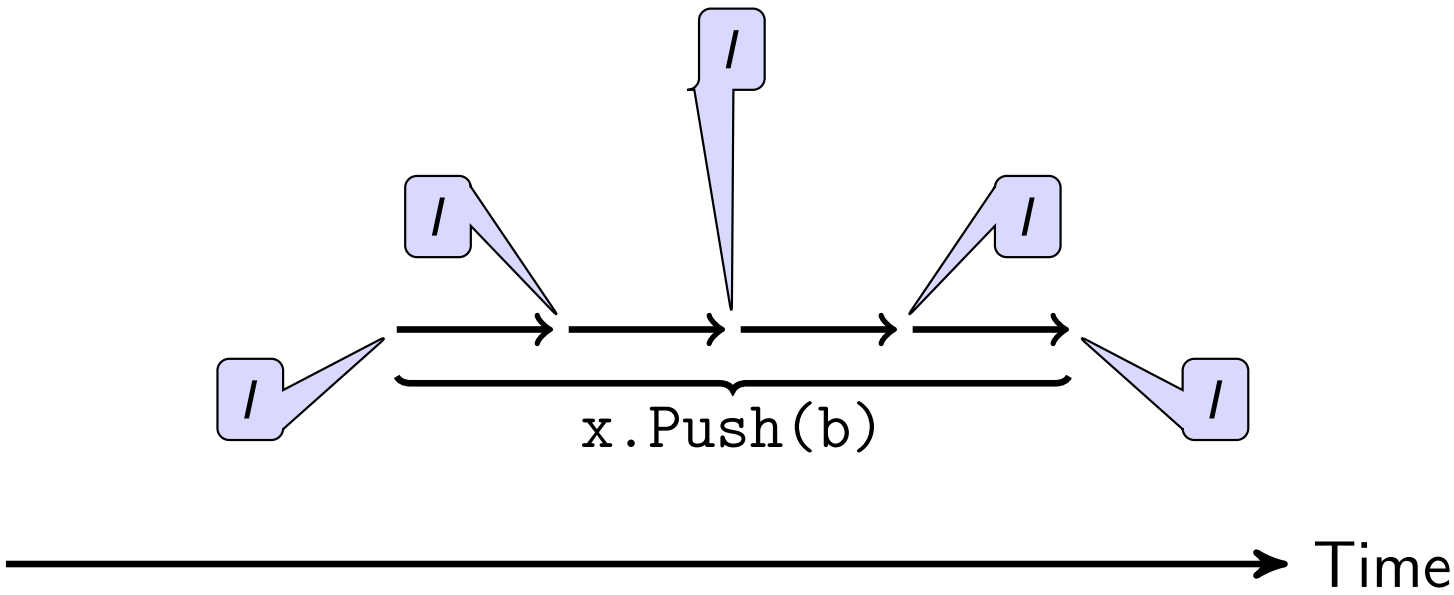
Bag example



where $I \stackrel{\text{def}}{=} \exists A.\ \mathbf{bag}(\mathtt{x}, A) * \mathtt{x}_{\mathtt{cont}} \overset{1/2}{\mapsto} A$

Updating the phantom field requires both halves, thus ensuring clients and ADT agree on the current abstract state.

Bag example



Parameterize the specification of ADT methods with **view-shifts** to atomically update the clients' half of the phantom field when the abstract state changes.

Refineable bag specification

$$\{\mathbf{emp}\} \quad \texttt{new Bag()} \quad \{\mathbf{ret}.\ \mathbf{bag}(\mathbf{ret}) * \mathbf{ret}_{cont} \overset{1/2}{\mapsto} \emptyset\}$$

$$\frac{\forall Y.\ \mathbf{this}_{cont} \overset{1/2}{\mapsto} Y * P \sqsubseteq \mathbf{this}_{cont} \overset{1/2}{\mapsto} (Y \cup \{\texttt{x}\}) * Q}{\{\mathbf{bag}(\mathbf{this}) * P\} \quad \texttt{Push(x)} \quad \{\mathbf{bag}(\mathbf{this}) * Q\}}$$

Refineable bag specification

$$\frac{}{\{\mathbf{emp}\}\quad \texttt{new Bag()}\quad \{\mathbf{ret}.\ \mathbf{bag}(\mathbf{ret}) * \mathbf{ret}_{cont} \overset{1/2}{\mapsto} \emptyset\}}$$

$$\frac{\forall Y.\ \mathbf{this}_{cont} \overset{1/2}{\mapsto} Y * P \sqsubseteq \mathbf{this}_{cont} \overset{1/2}{\mapsto} (Y \cup \{\texttt{x}\}) * Q}{\{\mathbf{bag}(\mathbf{this}) * P\}\quad \texttt{Push(x)}\quad \{\mathbf{bag}(\mathbf{this}) * Q\}}$$

$$\frac{\begin{array}{c}\mathbf{this}_{cont} \overset{1/2}{\mapsto} \emptyset * P \sqsubseteq \mathbf{this}_{cont} \overset{1/2}{\mapsto} \emptyset * Q(\mathit{null}) \\ \forall X.\ \forall y.\ \mathbf{this}_{cont} \overset{1/2}{\mapsto} X \cup \{y\} * P \sqsubseteq \mathbf{this}_{cont} \overset{1/2}{\mapsto} X * Q(y)\end{array}}{\{\mathbf{bag}(\mathbf{this}) * P\}\texttt{Pop()}\{\mathbf{ret}.\ \mathbf{bag}(\mathbf{this}) * Q(\mathbf{ret})\}}$$

Bag example

Given the refinable bag specification **clients** can **derive** the standard sequential specification

$$\{\mathbf{emp}\} \quad \texttt{new Bag()} \quad \{\mathbf{ret}.\ \mathbf{bag}_e(\mathbf{ret}, \emptyset)\}$$

$$\{\mathbf{bag}_e(\mathbf{this}, A)\} \quad \texttt{Push(x)} \quad \{\mathbf{bag}_e(\mathbf{this}, A \cup \{x\})\}$$

$$\{\mathbf{bag}_e(\mathbf{this}, \emptyset)\} \quad \texttt{Pop()} \quad \{\mathbf{ret}.\ \mathbf{bag}_e(\mathbf{this}, \emptyset) * \mathbf{ret} = \mathbf{null}\}$$

$$\{\mathbf{bag}_e(\mathbf{this}, A) * A \neq \emptyset\} \quad \texttt{Pop()} \quad \{\mathbf{ret}.\ \mathbf{ret} \in A * \mathbf{bag}_e(A \setminus \{\mathbf{ret}\})\}$$

$$\mathbf{bag}_e(x, A) * \mathbf{bag}_e(x, B) \Rightarrow \bot$$

which enforces a single exclusive owner.

Bag example

But **clients** can also **derive** more interesting specifications like the following shared bag specification

$$\{\mathbf{emp}\} \quad \texttt{new Bag()} \quad \{\mathbf{ret}.\ \mathbf{bag}_s(\mathbf{ret}, P)\}$$

$$\{\mathbf{bag}_s(\mathbf{this}, P) * P(x)\} \quad \texttt{Push(x)} \quad \{\mathbf{bag}_s(\mathbf{this}, P)\}$$

$$\{\mathbf{bag}_s(\mathbf{this}, P)\} \quad \texttt{Pop()} \quad \{\mathbf{ret}.\ \mathbf{bag}_s(\mathbf{this}, P) * (\mathbf{ret} = \mathit{null} \vee P(\mathbf{ret}))\}$$

$$\mathbf{bag}_s(x, P) \Leftrightarrow \mathbf{bag}_s(x, P) * \mathbf{bag}_s(x, P)$$

which allows unrestricted sharing and associates ownership of additional resources with each element.

Why is this difficult?

ADT resources allow us to introduce sharing of the ADT, but they also allow us to use the ADT to govern ownership of other mutable data structures.

For instance, the $\mathbf{bag}_s(-, P)$ resource allows us to transfer ownership of additional resources through the bag.

What if the client instantiates P in $\mathbf{bag}_s(-, P)$ with an assertion that refers to the bag itself?

Conclusion

We've presented a **new separation logic and specification pattern** that allow **clients** to construct ADT resources, in a concurrent higher-order setting with reentrancy.

The refinement pattern is just one application of the logic; in addition we have verified

- A Concurrent Runner library (used to parallelize divide-and-conquer algorithms)
- The C# Joins library (provides a declarative concurrency model for implementing synchronization primitives)

Questions?

Relation to linearizability

Linearizability aims to establish a fiction of atomicity; our specification pattern does not!

For instance, the method $\mathit{Push}2(x, y)$ implemented as $\mathit{Push}(x); \mathit{Push}(y)$ is **not** linearizable; but using our specification pattern we can give it a sensible specification:

$$\frac{\begin{array}{l}\forall X.\ \mathbf{this}_{\mathit{cont}} \overset{1/2}{\mapsto} X * P \sqsubseteq \mathbf{this}_{\mathit{cont}} \overset{1/2}{\mapsto} (X \cup \{\mathtt{x}\}) * Q \\ \forall X.\ \mathbf{this}_{\mathit{cont}} \overset{1/2}{\mapsto} X * Q \sqsubseteq \mathbf{this}_{\mathit{cont}} \overset{1/2}{\mapsto} (X \cup \{\mathtt{y}\}) * R\end{array}}{\{\mathbf{bag}(\mathbf{this}) * P\} \quad \texttt{Push2(x, y)} \quad \{\mathbf{bag}(\mathbf{this}) * R\}}$$