Verifying Concurrent Data Structures

Roland Meyer

Joint work with Thomas Wies and Sebastian Wolff

Motivation

Programs = Algorithms + Data Structures

–Niklaus Wirth, 1978

Motivation

Programs = Algorithms + Data Structures

–Niklaus Wirth, 1978

Verification

Motivation

Programs = Algorithms + Data Structures

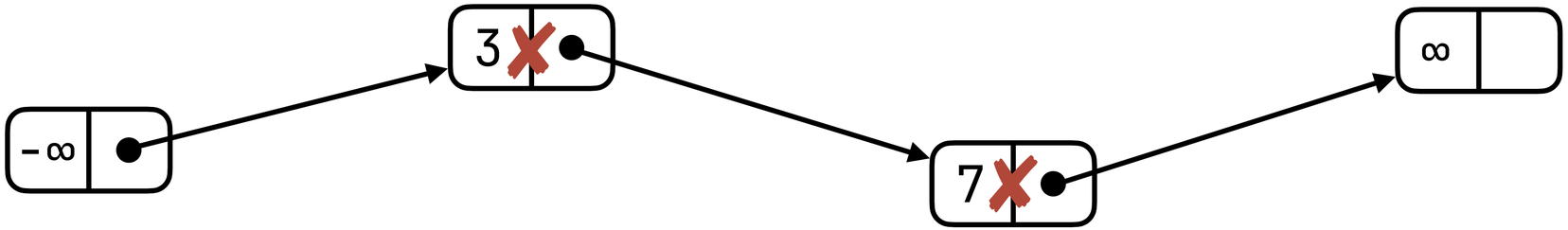
–Niklaus Wirth, 1978

Verification

Automation

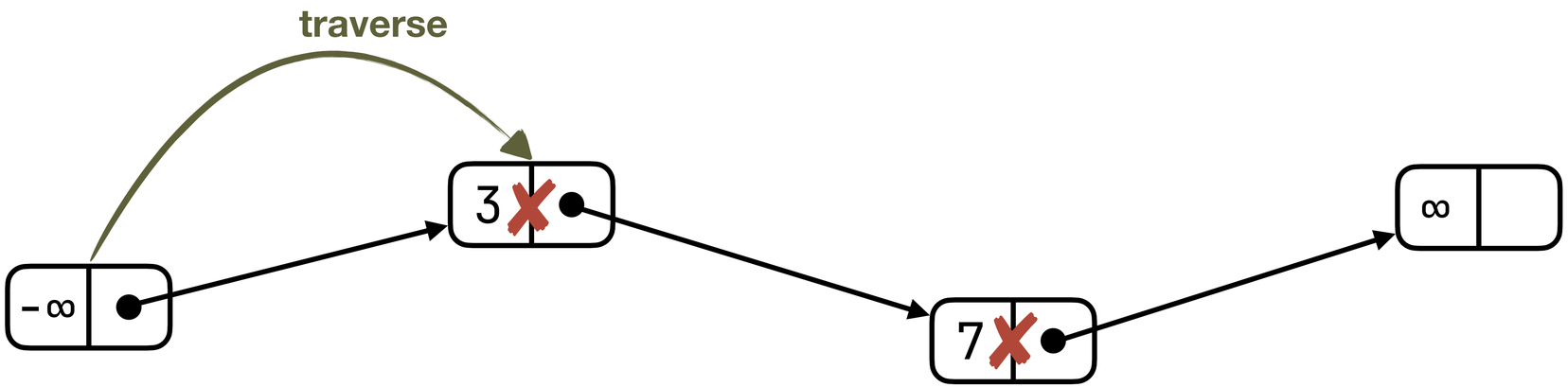
Example: `contains(5)`

`set({}):`



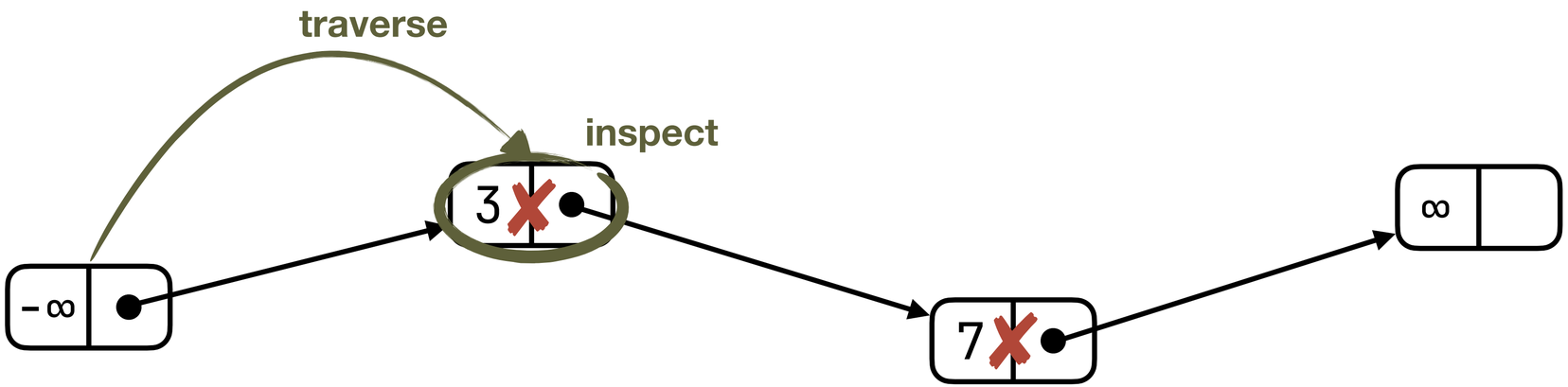
Example: `contains(5)`

`set({}):`



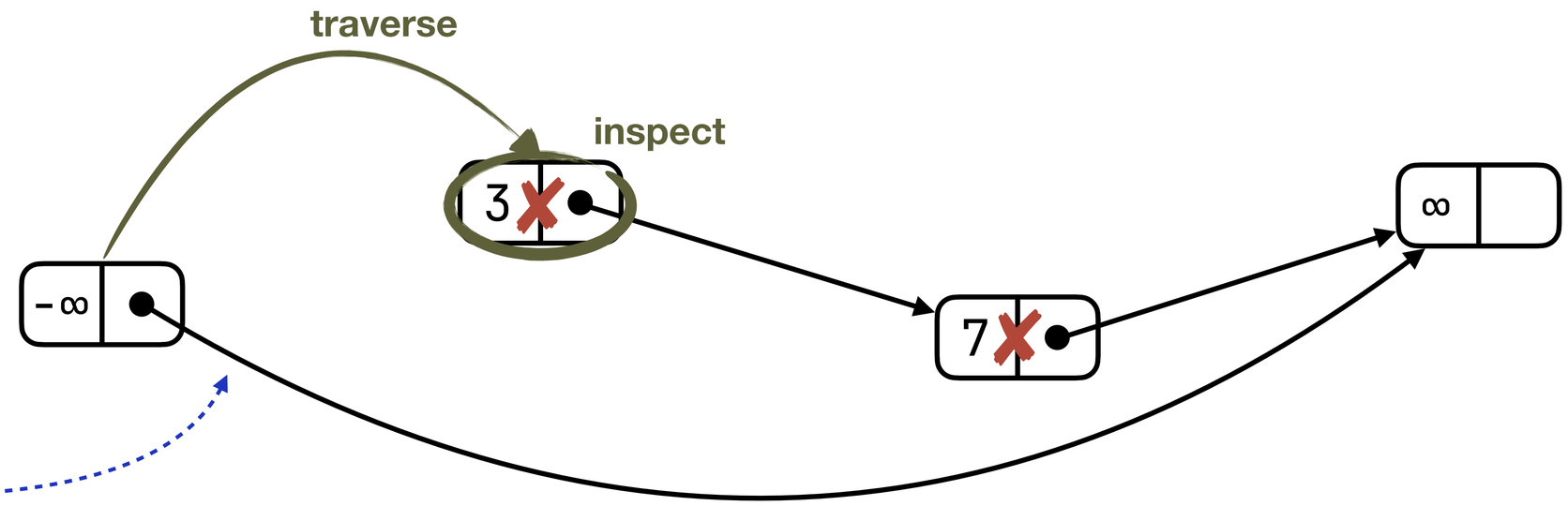
Example: `contains(5)`

`set({}):`



Example: `contains(5)`

`set({}):`

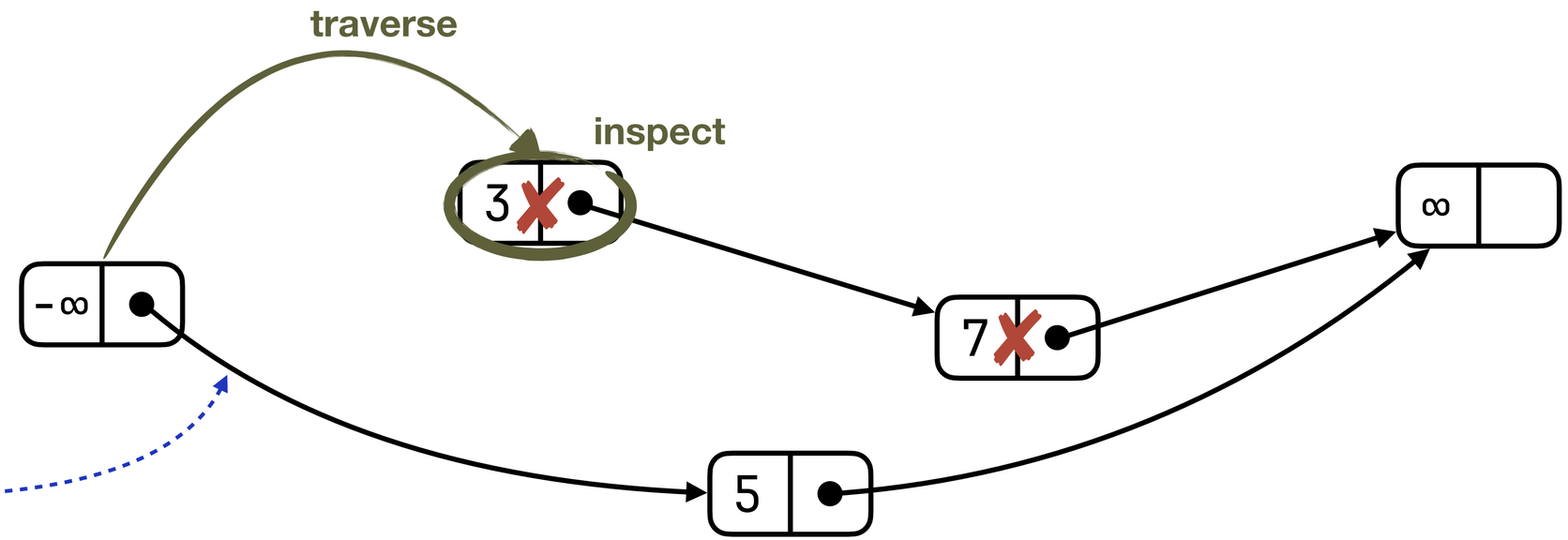


interference

Example: `contains(5)`

~~`set({}):`~~

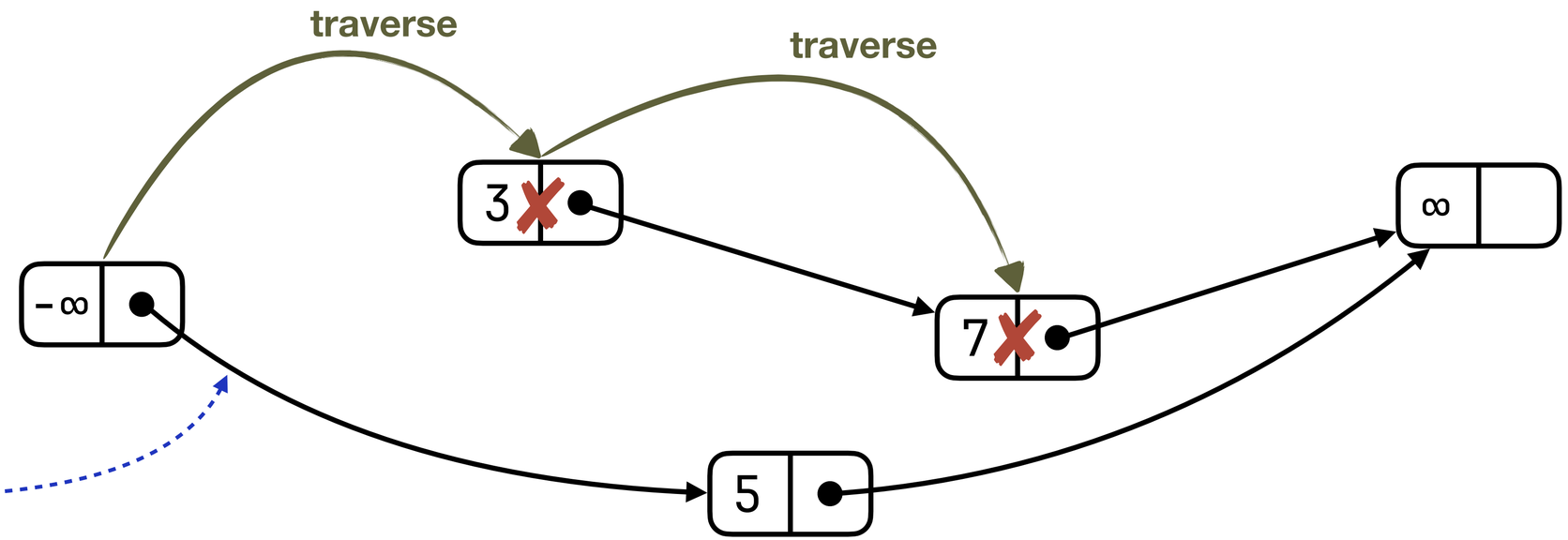
`set({5}):`



Example: `contains(5)`

~~`set({}):`~~

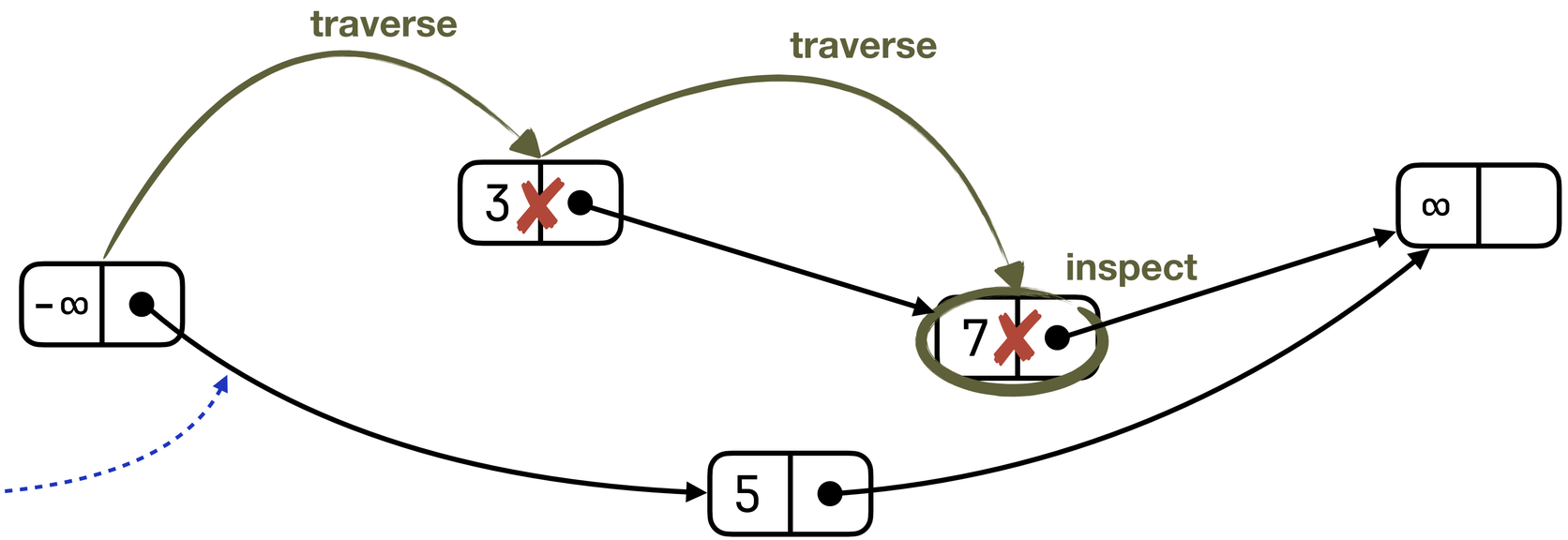
`set({5}):`



Example: `contains(5)`

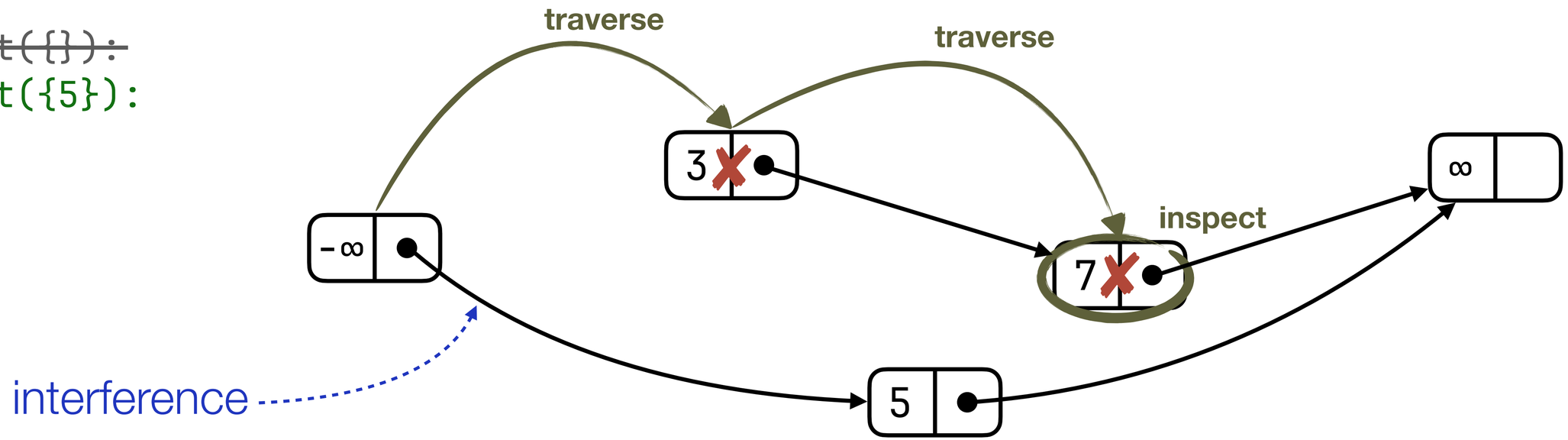
~~`set({}):`~~

`set({5}):`



Example: `contains(5)`

~~`set({}):`~~
`set({5}):`



- Result: `contains(5)` = **`false`**
  - ➡ linearizable (i.e. correct); proof obligation: `traversal` ⊨ `set(M), 5∉M` *at some point*

Approach: Program Logic

manual/mechanized/automated proof construction

- Owicki-Gries
  - ➡ Thread modularity
  - ➡ Implicit interference handling

compact proofs

- Separation logic
  - ➡ Parametric in the domain
  - ➡ Local reasoning
  - ➡ Arsenal of proof techniques

applicability

Contributions

Flow Framework

Reasoning about heap graphs without recursive predicates.

[POPL'18, ESOP'20, TACAS'23]

Hindsight

Non-fixed linearization points without prophecies.

[OOPSLA'22, PLDI'23]

Decomposing Updates

Unbounded footprints without induction.

[OOPSLA'22, *under submission*]

Contributions

Flow Framework

Reasoning about heap graphs without recursive predicates.

[POPL'18, ESOP'20, TACAS'23]

Hindsight

Non-fixed linearization points without prophecies.

[OOPSLA'22, PLDI'23]

Decomposing Updates

Unbounded footprints without induction.

[OOPSLA'22, *under submission*]

Goal

Reason about heap graphs of concurrent data structures.

Problem

No inductive definition, no recursive predicates.

Solution

Fixed points, data-flow analysis on heap graphs.

Flow Framework

Flow Framework

flow graphs = heap graphs + data flow values (ghost)

Flow Framework

flow graphs = heap graphs + data flow values (ghost)

- Flow values from commutative monoid $(\mathbb{M}, +, 0)$

Flow Framework

flow graphs = heap graphs + data flow values (ghost)

- Flow values from
  commutative monoid $(\mathbb{M}, +, 0)$

- Flow propagation via
  continuous edge functions

Flow Framework

flow graphs = heap graphs + data flow values (ghost)

- Flow values from
  commutative monoid $(\mathbb{M}, +, 0)$

- Flow propagation via
  continuous edge functions

- ***The flow***:
  fixed point, wrt. initial value

Flow Framework

flow graphs = heap graphs + data flow values (ghost)

- Flow values from
  commutative monoid $(\mathbb{M}, +, 0)$
- Flow propagation via
  continuous edge functions
- ***The flow***:
  some fixed point, wrt. initial value

$m \in$

$m \in$

$m \in$ $\in \mathbb{M}$

[POPL'18]

Go with the Flow: Compositional Abstractions for Concurrent Data Structures*

SIDDHARTH KRISHNA, New York University, USA
DENNIS SHASHA, New York University, USA
THOMAS WIES, New York University, USA

[ESOP'20]

Local Reasoning for Global Graph Properties

Siddharth Krishna[1], Alexander J. Summers[2], and Thomas Wies[1]

[1] New York University, New York, NY, USA, {siddharth,wies}@cs.nyu.edu
[2] ETH Zürich, Zurich, Switzerland, alexander.summers@inf.ethz.ch

Abstract. Separation logics are widely used for verifying programs that manipulate complex heap-based data structures. These logics build on so-called *separation algebras*, which allow expressing properties of heap regions such that modifications to a region do not invalidate properties stated about the remainder of the heap. This concept is key to enabling modular reasoning and also extends to concurrency. While heaps are naturally related to mathematical graphs, many ubiquitous graph properties are non-local in character, such as reachability between nodes, path lengths, acyclicity and other structural invariants, as well as data invariants which combine with these notions. Reasoning modularly about such graph properties remains notoriously difficult, since a local modification can have side-effects on a global property that cannot be easily confined to a small region.
In this paper, we address the question: What separation algebra can be used to avoid proof arguments reverting back to tedious global reasoning in such cases? To this end, we consider a general class of global graph properties expressed as fixpoints of algebraic equations over graphs. We present mathematical foundations for reasoning about this class of properties, imposing minimal requirements on the underlying theory that allow us to define a suitable separation algebra. Building on this theory, we develop a general proof technique for modular reasoning about global graph properties expressed over program heaps, in a way which can be directly integrated with existing separation logics. To demonstrate our approach, we present local proofs for two challenging examples: a priority inheritance protocol and the non-blocking concurrent Harris list.

1 Introduction

Separation logic (SL) [31,37] provides the basis of many successful verification tools that can verify programs manipulating complex data structures [1,4,17,29]. This success is due to the logic's support for reasoning modularly about modifications to heap-based data. For simple inductive data structures such as lists and trees, much of this reasoning can be automated [2,11,20,33]. However, these techniques often fail when data structures are less regular (e.g. multiple overlaid data structures) or provide multiple traversal

Flow Framework

flow graphs = heap graphs + data flow values (ghost)

- Flow values from
  commutative monoid $(\mathbb{M}, +, 0)$
- Flow propagation via
  continuous edge functions
- ***The flow***:
  some fixed point, wrt. initial value

$m \in$

$m \in$

$m \in$ $\in \mathbb{M}$

[POPL'18]

Go with the Flow: Compositional Abstractions for Concurrent Data Structures*

SIDDHARTH KRISHNA, New York University, USA
DENNIS SHASHA, New York University, USA
THOMAS WIES, New York University, USA

[ESOP'20]

Local Reasoning for Global Graph Properties

Siddharth Krishna[1], Alexander J. Summers[2], and Thomas Wies[1]

[1] New York University, New York, NY, USA, {siddharth,wies}@cs.nyu.edu
[2] ETH Zürich, Zurich, Switzerland, alexander.summers@inf.ethz.ch

Abstract. Separation logics are widely used for verifying programs that manipulate complex heap-based data structures. These logics build on so-called *separation algebras*, which allow expressing properties of heap regions such that modifications to a region do not invalidate properties stated about the remainder of the heap. This concept is key to enabling modular reasoning and also extends to concurrency. While heaps are naturally related to mathematical graphs, many ubiquitous graph properties are non-local in character, such as reachability between nodes, path lengths, acyclicity and other structural invariants, as well as data invariants which combine with these notions. Reasoning modularly about such graph properties remains notoriously difficult, since a local modification can have side-effects on a global property that cannot be easily confined to a small region.
In this paper, we address the question: What separation algebra can be used to avoid proof arguments reverting back to tedious global reasoning in such cases? To this end, we consider a general class of global graph properties expressed as fixpoints of algebraic equations over graphs. We present mathematical foundations for reasoning about this class of properties, imposing minimal requirements on the underlying theory that allow us to define a suitable separation algebra. Building on this theory, we develop a general proof technique for modular reasoning about global graph properties expressed over program heaps, in a way which can be directly integrated with existing separation logics. To demonstrate our approach, we present local proofs for two challenging examples: a priority inheritance protocol and the non-blocking concurrent Harris list.

1 Introduction

Separation logic (SL) [31,37] provides the basis of many successful verification tools that can verify programs manipulating complex data structures [1,4,17,29]. This success is due to the logic's support for reasoning modularly about modifications to heap-based data. For simple inductive data structures such as lists and trees, much of this reasoning can be automated [2,11,20,33]. However, these techniques often fail when data structures are less regular (e.g. multiple overlaid data structures) or provide multiple traversal

Flow Framework

flow graphs = heap graphs + data flow values (ghost)

- Flow values from
  commutative monoid $(\mathbb{M}, +, 0)$
- Flow propagation via
  continuous edge functions
- ***The flow***:
  some fixed point, wrt. initial value

$m \in$

$m \in$

$m \in$ $\in \mathbb{M}$

[POPL'18]

Go with the Flow: Compositional Abstractions for Concurrent Data Structures*

SIDDHARTH KRISHNA, New York University, USA
DENNIS SHASHA, New York University, USA
THOMAS WIES, New York University, USA

[ESOP'20]

Local Reasoning for Global Graph Properties

Siddharth Krishna[1], Alexander J. Summers[2], and Thomas Wies[1]

[1] New York University, New York, NY, USA, {siddharth,wies}@cs.nyu.edu
[2] ETH Zürich, Zurich, Switzerland, alexander.summers@inf.ethz.ch

Abstract. Separation logics are widely used for verifying programs that manipulate complex heap-based data structures. These logics build on so-called *separation algebras*, which allow expressing properties of heap regions such that modifications to a region do not invalidate properties stated about the remainder of the heap. This concept is key to enabling modular reasoning and also extends to concurrency. While heaps are naturally related to mathematical graphs, many ubiquitous graph properties are non-local in character, such as reachability between nodes, path lengths, acyclicity and other structural invariants, as well as data invariants which combine with these notions. Reasoning modularly about such graph properties remains notoriously difficult, since a local modification can have side-effects on a global property that cannot be easily confined to a small region.
In this paper, we address the question: What separation algebra can be used to avoid proof arguments reverting back to tedious global reasoning in such cases? To this end, we consider a general class of global graph properties expressed as fixpoints of algebraic equations over graphs. We present mathematical foundations for reasoning about this class of properties, imposing minimal requirements on the underlying theory that allow us to define a suitable separation algebra. Building on this theory, we develop a general proof technique for modular reasoning about

are less regular (e.g. multiple overlaid data structures) or provide multiple traversal

Lower bounds meaningless (flow may vanish in compositions).
Computation has to restrict graph structure (e.g. acyclicity).

Flow Framework

flow graphs = heap graphs + data flow values (ghost)

- Flow values from
  commutative monoid $(\mathbb{M}, +, 0)$
- Flow propagation via
  continuous edge functions
- ***The flow***:
  some fixed point, wrt. initial value

$m \in$

$m \in$

$m \in$ $\in \mathbb{M}$

[POPL'18]

Go with the Flow: Compositional Abstractions for Concurrent Data Structures*

SIDDHARTH KRISHNA, New York University, USA
DENNIS SHASHA, New York University, USA
THOMAS WIES, New York University, USA

[ESOP'20]

Local Reasoning for Global Graph Properties

Siddharth Krishna[1], Alexander J. Summers[2], and Thomas Wies[1]

[1] New York University, New York, NY, USA, {siddharth,wies}@cs.nyu.edu
[2] ETH Zürich, Zurich, Switzerland, alexander.summers@inf.ethz.ch

Abstract. Separation logics are widely used for verifying programs that manipulate complex heap-based data structures. These logics build on so-called *separation algebras*, which allow expressing properties of heap regions such that modifications to a region do not invalidate properties stated about the remainder of the heap. This concept is key to enabling modular reasoning and also extends to concurrency. While heaps are naturally related to mathematical graphs, many ubiquitous graph properties are non-local in character, such as reachability between nodes, path lengths, acyclicity and other structural invariants, as well as data invariants which combine with these notions. Reasoning modularly about such graph properties remains notoriously difficult, since a local modification can have side-effects on a global property that cannot be easily confined to a small region.
In this paper, we address the question: What separation algebra can be used to avoid proof arguments reverting back to tedious global reasoning in such cases? To this end, we consider a general class of global graph properties expressed as fixpoints of algebraic equations over graphs. We present mathematical foundations for reasoning about this class of properties, imposing minimal requirements on the underlying theory that allow us to define a suitable separation algebra. Building on this theory, we develop a general proof technique for modular reasoning about

are less regular (e.g. multiple overlaid data structures) or provide multiple traversal

Lower bounds meaningless (flow may vanish in compositions).
Computation has to restrict graph structure (e.g. acyclicity).

Flow Framework

flow graphs = heap graphs + data flow values (ghost)

- Flow values from
  commutative monoid $(\mathbb{M}, +, 0)$ $m \in$
- Flow propagation via
  continuous edge functions $m \in$
- ***The flow***:
  fixed point, wrt. initial value $m \in$


[POPL'18]

Go with the Flow: Compositional Abstractions for Concurrent Data Structures*

SIDDHARTH KRISHNA, New York University, USA
DENNIS SHASHA, New York University, USA
THOMAS WIES, New York University, USA

[ESOP'20]

Local Reasoning for Global Graph Properties

Siddharth Krishna[1], Alexander J. Summers[2], and Thomas Wies[1]

[1] New York University, New York, NY, USA, {siddharth,wies}@cs.nyu.edu
[2] ETH Zürich, Zurich, Switzerland, alexander.summers@inf.ethz.ch

[TACAS'23]

Make flows small again: revisiting the flow framework

Roland Meyer[1], Thomas Wies[2], and Sebastian Wolff[2(✉)]

[1] TU Braunschweig, Braunschweig, Germany, meyer@tu-bs.de
[2] New York University, New York, USA, {wies,sebastian.wolff}@cs.nyu.edu

Abstract We present a new flow framework for separation logic reasoning about programs that manipulate general graphs. The framework overcomes problems in earlier developments: it is based on standard fixed point theory, guarantees least flows, rules out vanishing flows, and has an easy to understand notion of footprint as needed for soundness of the frame rule. In addition, we present algorithms for

Lower bounds!
Computation has mild assumptions
about monoid + edge functions.

Flow Framework

flow graphs = heap graphs + data flow values (ghost)

- Flow values from
  commutative monoid $(\mathbb{M}, +, 0)$

$m \in$

- Flow propagation via
  continuous edge functions

$m \in$

- ***The flow***:
  least fixed point, wrt. initial value

$m \in$

[POPL'18]

Go with the Flow: Compositional Abstractions for Concurrent Data Structures*

SIDDHARTH KRISHNA, New York University, USA
DENNIS SHASHA, New York University, USA
THOMAS WIES, New York University, USA

[ESOP'20]

Local Reasoning for Global Graph Properties

Siddharth Krishna[1], Alexander J. Summers[2], and Thomas Wies[1]

[1] New York University, New York, NY, USA, {siddharth,wies}@cs.nyu.edu
[2] ETH Zürich, Zurich, Switzerland, alexander.summers@inf.ethz.ch

[TACAS'23]

Make flows small again: revisiting the flow framework

Roland Meyer[1], Thomas Wies[2], and Sebastian Wolff[2(✉)]

[1] TU Braunschweig, Braunschweig, Germany, meyer@tu-bs.de
[2] New York University, New York, USA, {wies,sebastian.wolff}@cs.nyu.edu

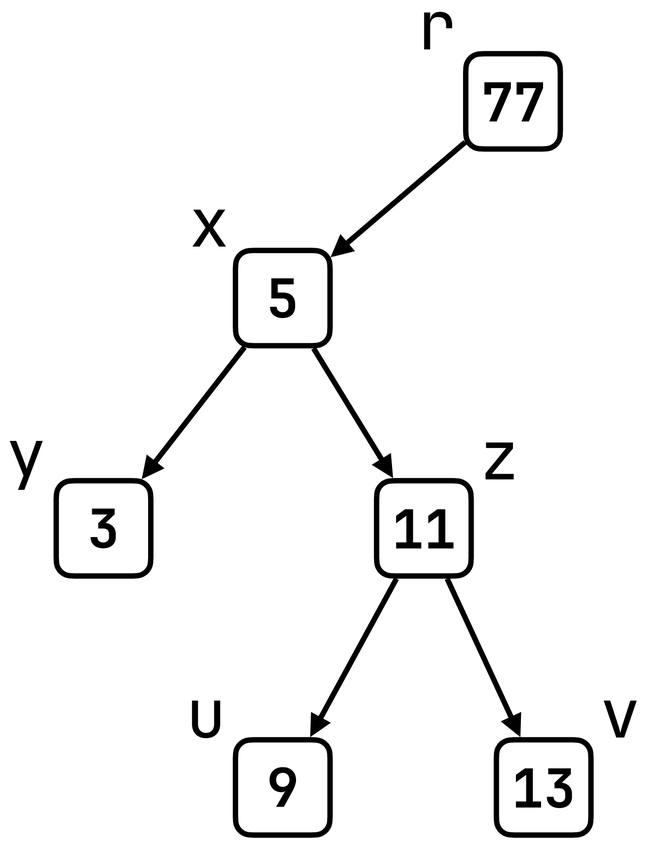
Abstract We present a new flow framework for separation logic reasoning about programs that manipulate general graphs. The framework overcomes problems in earlier developments: it is based on standard fixed point theory, guarantees least flows, rules out vanishing flows, and has an easy to understand notion of footprint as needed for soundness of the frame rule. In addition, we present algorithms for

NEW

Lower bounds!
Computation has mild assumptions
about monoid + edge functions.

Flow Framework

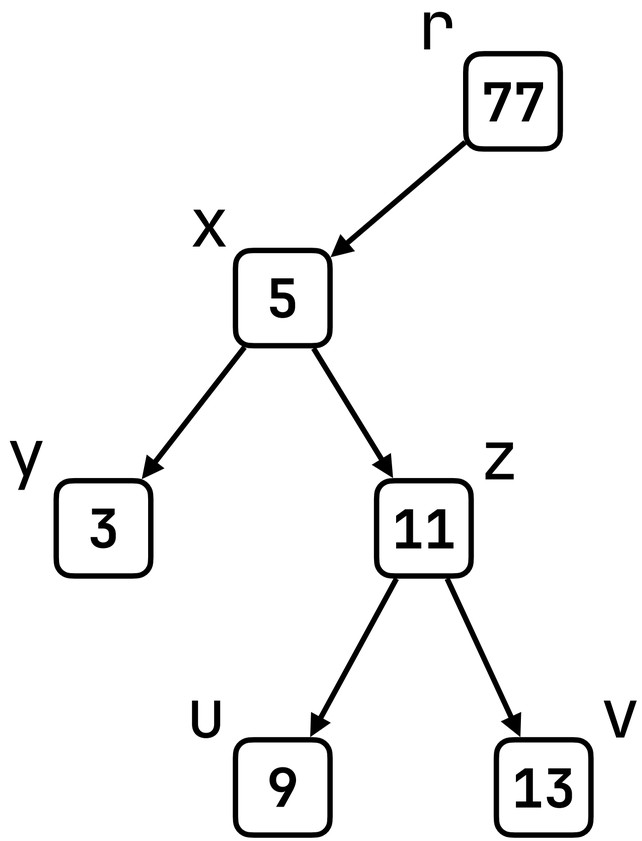
- Flow values from
- Flow propagation via
- ***The flow***:



Flow Framework

flow graphs = heap graphs + data flow values (ghost)

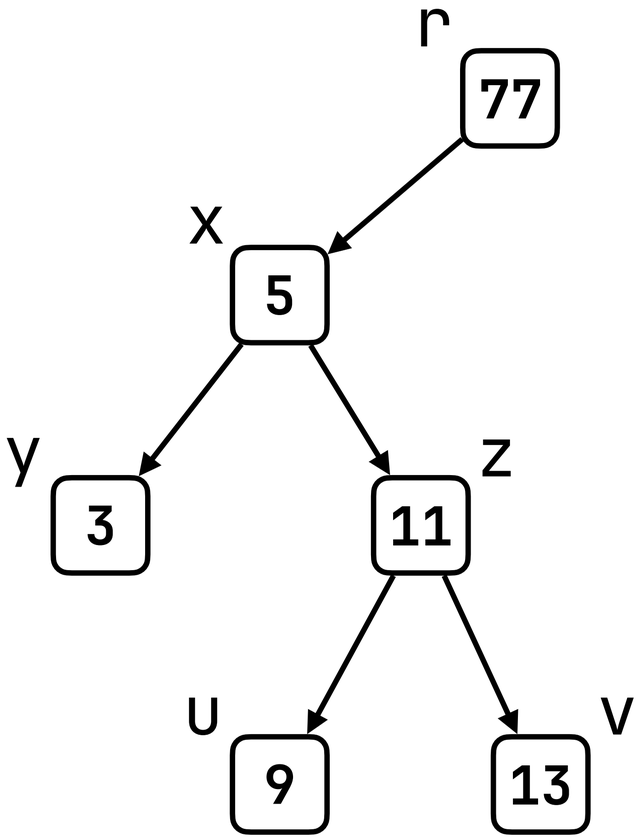
- Flow values from
- Flow propagation via
- ***The flow***:



Flow Framework

flow graphs = heap graphs + data flow values (ghost)

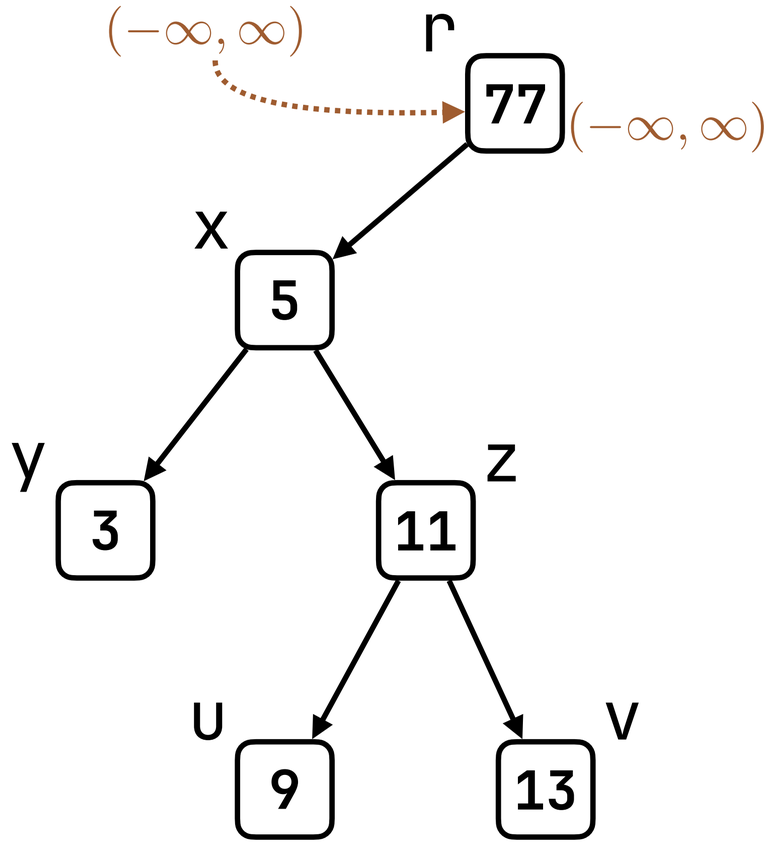
- Flow values from
  search path monoid $(2^{\mathbb{Z}\cup\{-\infty,\infty\}}, \cup, \varnothing)$
- Flow propagation via
- ***The flow***:



Flow Framework

flow graphs = heap graphs + data flow values (ghost)

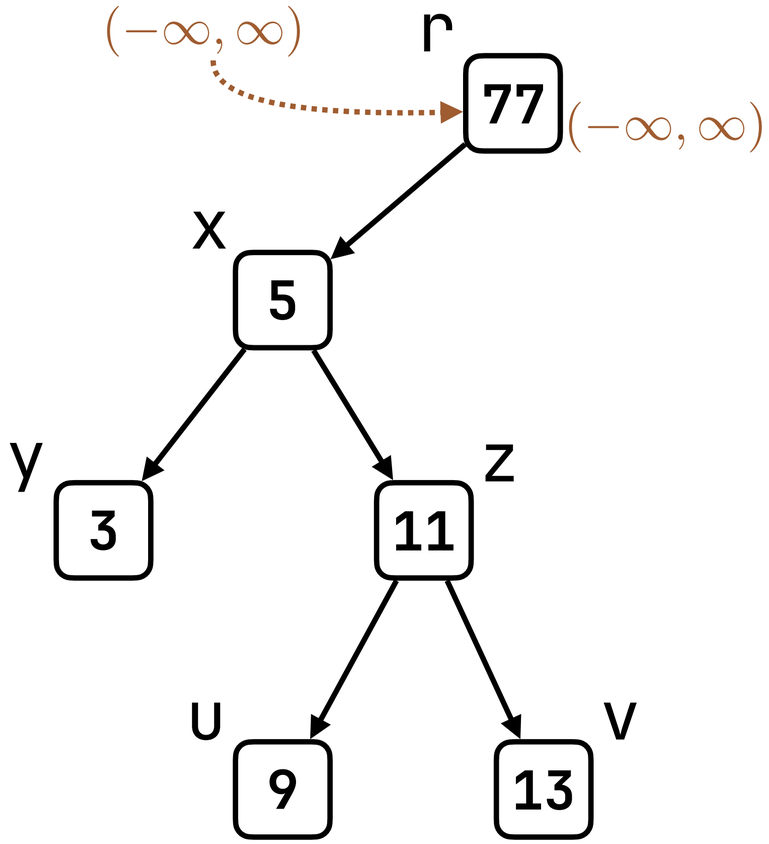
- Flow values from
  search path monoid $(2^{\mathbb{Z}\cup\{-\infty,\infty\}}, \cup, \varnothing)$
- Flow propagation via
- ***The flow***:



Flow Framework

flow graphs = heap graphs + data flow values (ghost)

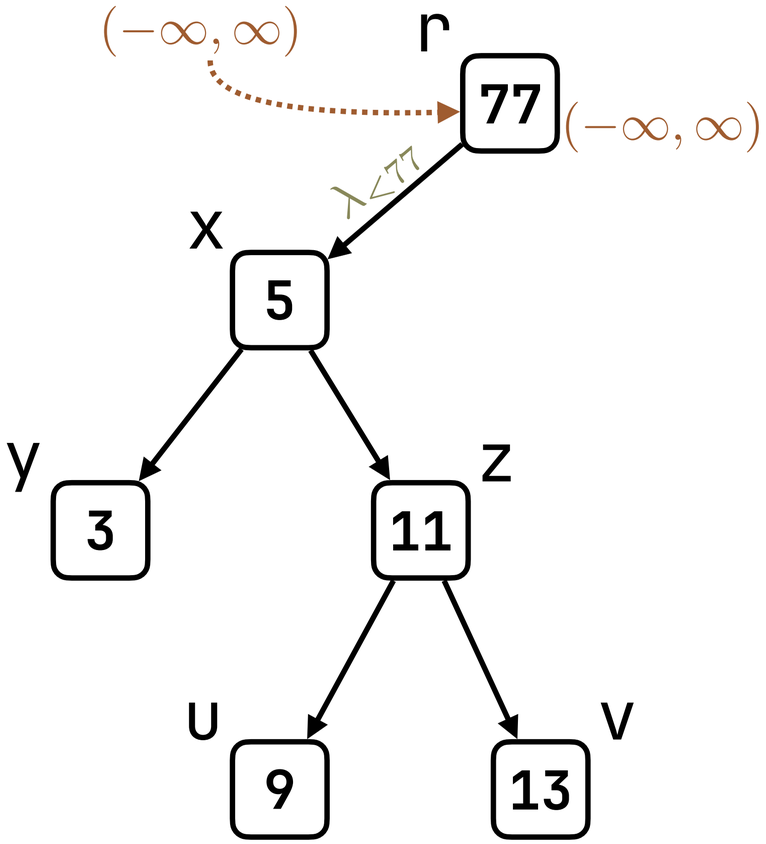
- Flow values from
  search path monoid $(2^{\mathbb{Z}\cup\{-\infty,\infty\}}, \cup, \varnothing)$
- Flow propagation via
  $\lambda_{<k} := \lambda m.\, m \cap (-\infty, k)$
  $\lambda_{>k} := \lambda m.\, m \cap (k, \infty)$
- ***The flow***:



Flow Framework

flow graphs = heap graphs + data flow values (ghost)

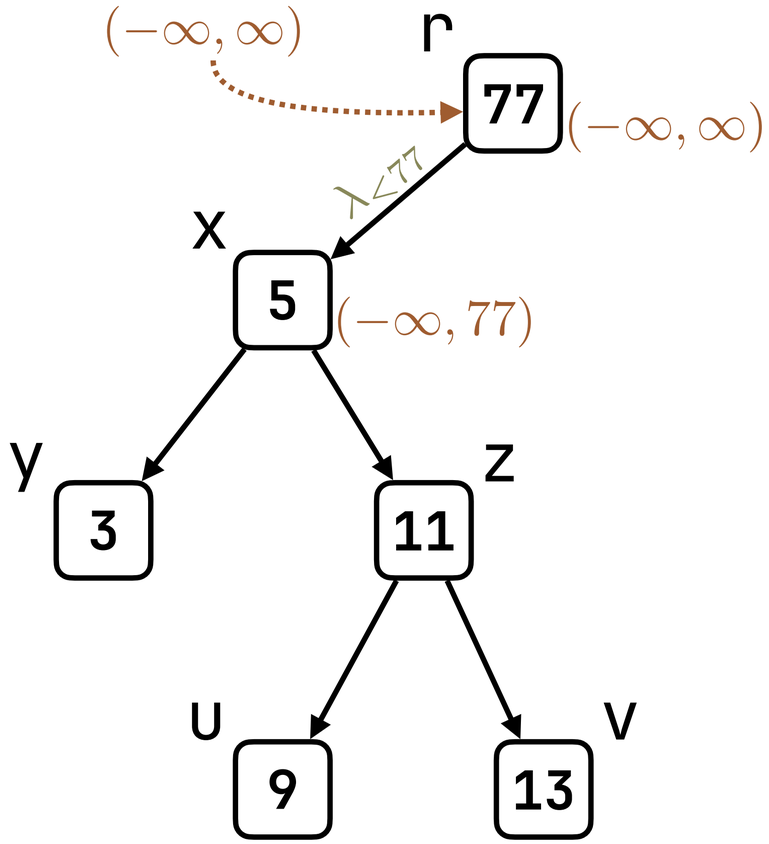
- Flow values from
  search path monoid $(2^{\mathbb{Z}\cup\{-\infty,\infty\}}, \cup, \varnothing)$
- Flow propagation via
  $\lambda_{<k} := \lambda m.\, m \cap (-\infty, k)$
  $\lambda_{>k} := \lambda m.\, m \cap (k, \infty)$
- ***The flow***:



Flow Framework

flow graphs = heap graphs + data flow values (ghost)

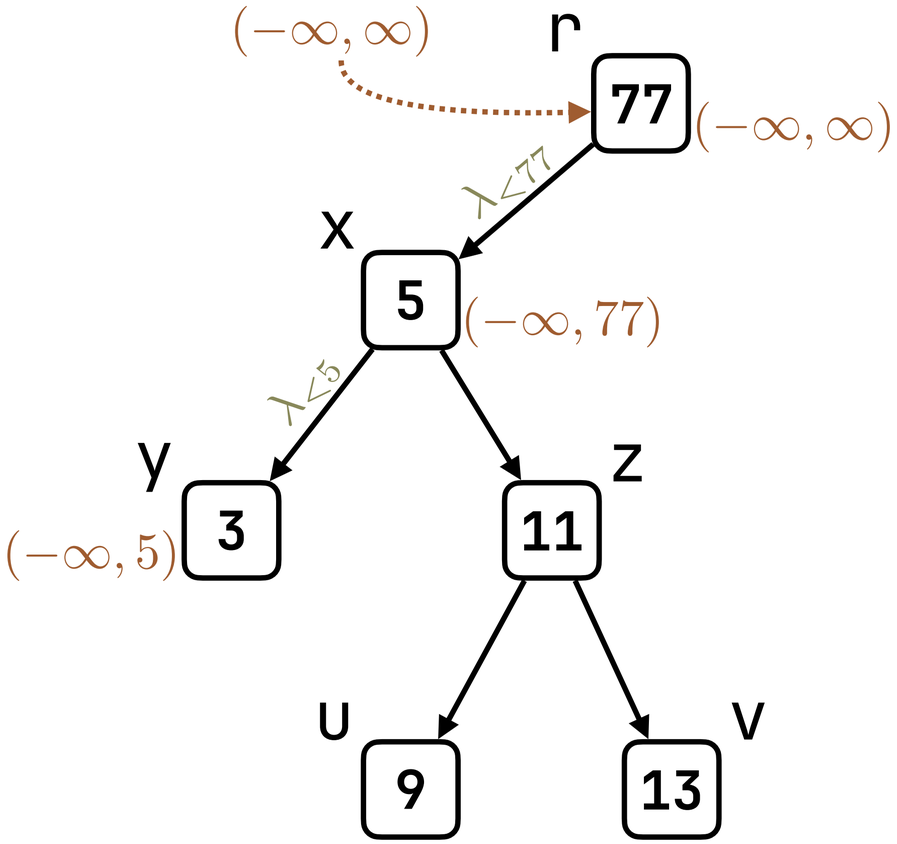
- Flow values from
  search path monoid $(2^{\mathbb{Z}\cup\{-\infty,\infty\}}, \cup, \varnothing)$
- Flow propagation via
  $\lambda_{<k} := \lambda m.\, m \cap (-\infty, k)$
  $\lambda_{>k} := \lambda m.\, m \cap (k, \infty)$
- ***The flow***:



Flow Framework

flow graphs = heap graphs + data flow values (ghost)

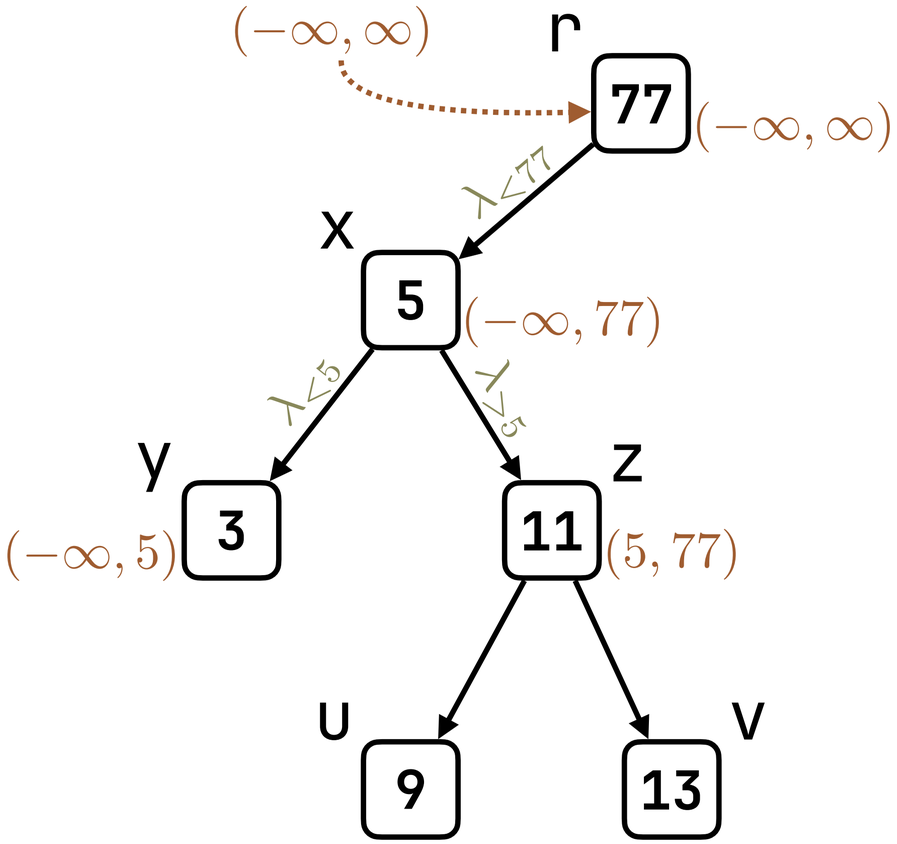
- Flow values from
  search path monoid $(2^{\mathbb{Z}\cup\{-\infty,\infty\}}, \cup, \varnothing)$
- Flow propagation via
  $\lambda_{<k} := \lambda m.\, m \cap (-\infty, k)$
  $\lambda_{>k} := \lambda m.\, m \cap (k, \infty)$
- ***The flow***:



Flow Framework

flow graphs = heap graphs + data flow values (ghost)

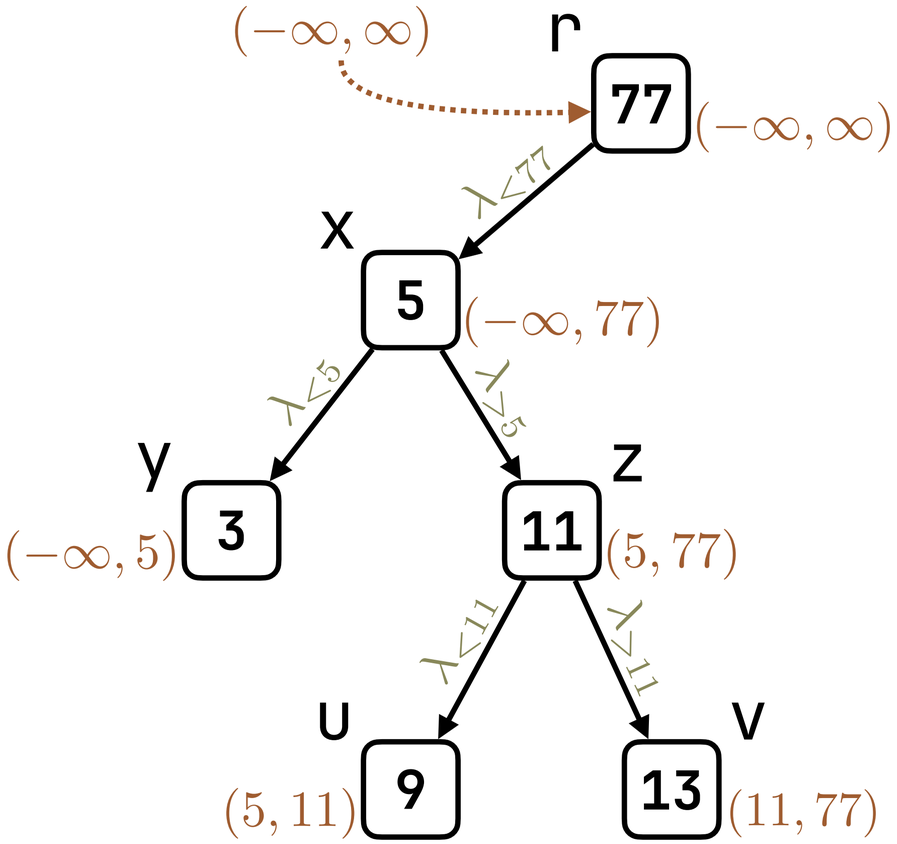
- Flow values from
  search path monoid $(2^{\mathbb{Z}\cup\{-\infty,\infty\}}, \cup, \varnothing)$
- Flow propagation via
  $\lambda_{<k} := \lambda m.\, m \cap (-\infty, k)$
  $\lambda_{>k} := \lambda m.\, m \cap (k, \infty)$
- ***The flow***:



Flow Framework

flow graphs = heap graphs + data flow values (ghost)

- Flow values from
  search path monoid $(2^{\mathbb{Z}\cup\{-\infty,\infty\}}, \cup, \varnothing)$
- Flow propagation via
  $\lambda_{<k} := \lambda m.\, m \cap (-\infty, k)$
  $\lambda_{>k} := \lambda m.\, m \cap (k, \infty)$
- ***The flow***:



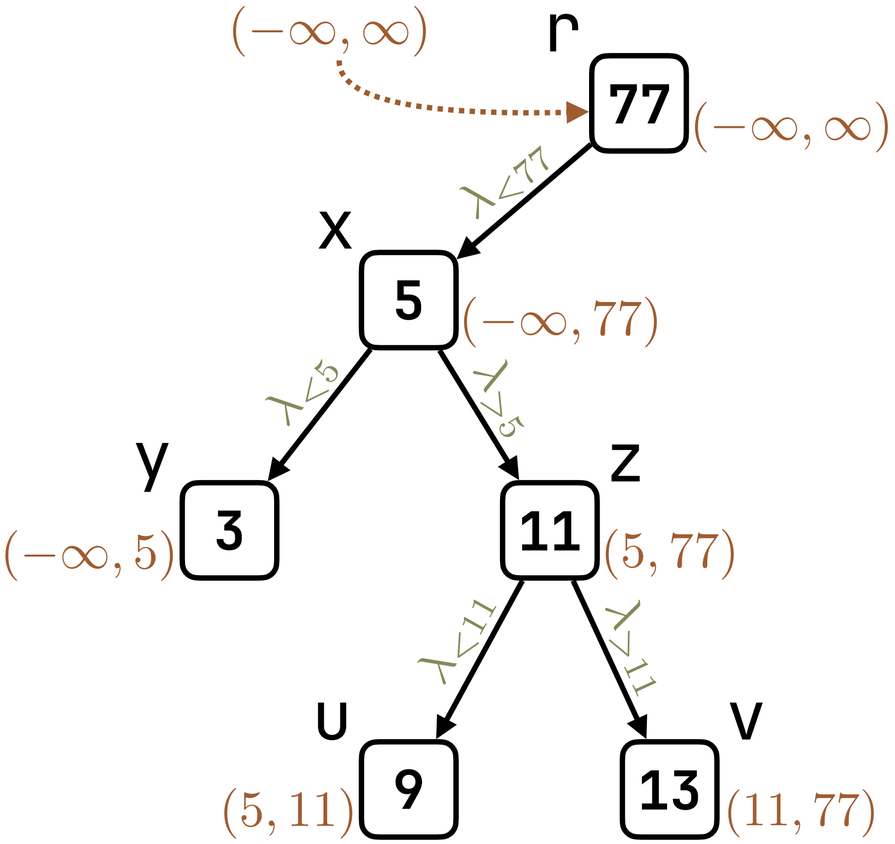
Flow Framework

flow graphs = heap graphs + data flow values (ghost)

- Flow values from
  search path monoid $(2^{\mathbb{Z}\cup\{-\infty,\infty\}}, \cup, \varnothing)$

- Flow propagation via
  $\lambda_{<k} := \lambda m.\, m \cap (-\infty, k)$
  $\lambda_{>k} := \lambda m.\, m \cap (k, \infty)$

- ***The flow***:
  Flow(x) = "*search paths that x lies on*"
  Outflow(x) = "*searches that continue from x*"
  Flow(x) $\setminus$ Outflow(x) = "*responsibility of x*"

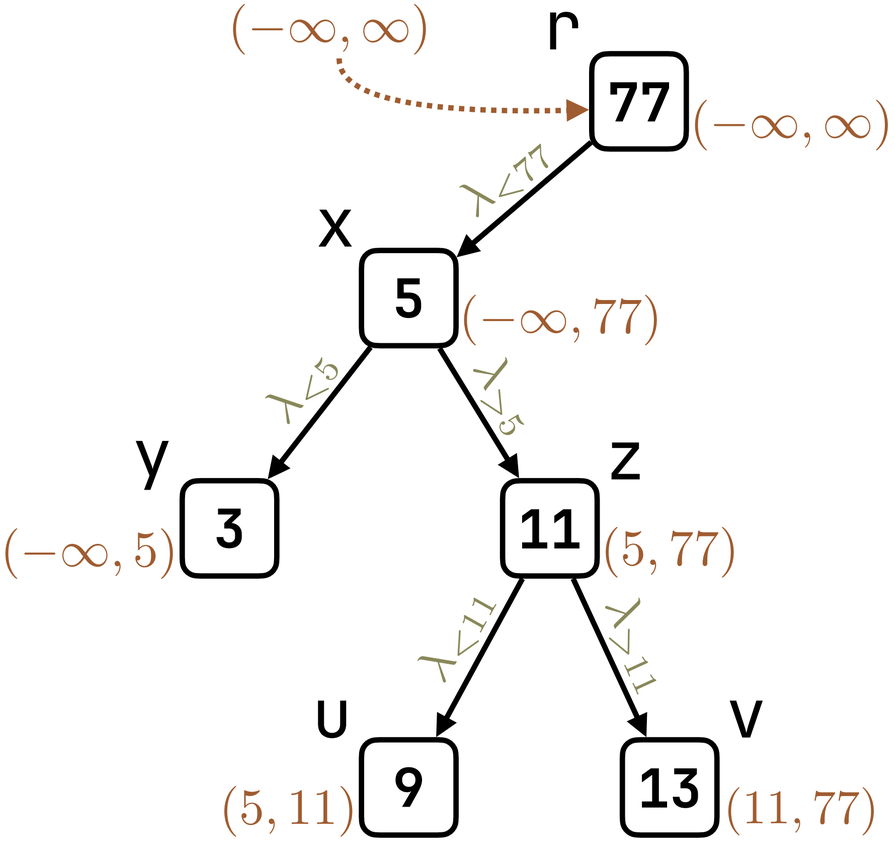


Flow Framework

flow graphs = heap graphs + data flow values (ghost)

- Flow values from
  search path monoid $(2^{\mathbb{Z}\cup\{-\infty,\infty\}}, \cup, \varnothing)$
- Flow propagation
  $\lambda_{<k} := \lambda m.\, m$
  $\lambda_{>k} := \lambda m.\, m \cap (k, \infty)$
- ***The flow***:
  Flow(x) = "*search paths that x lies on*"
  Outflow(x) = "*searches that continue from x*"
  Flow$(x) \setminus$ Outflow(x) = "*responsibility of x*"

Sufficient for
functional correctness.



Separation

Separation

- Flow graphs form a **separation algebra:**

Separation

$(\Sigma, \star, \mathrm{emp})$ with

$\star : \Sigma \times \Sigma \nrightarrow \Sigma$ associative, commutative, cancellative

emp set of units

- Flow graphs form a **separation algebra:**

Separation

$(\Sigma, \star, \mathrm{emp})$ with

$\star : \Sigma \times \Sigma \nrightarrow \Sigma$ associative, commutative, cancellative

emp set of units

- Flow graphs form a **separation algebra:**
  - ➡ framing: cut the graph, maintain inflow from **each node** outside (assoc. + cancel.).

Separation

$(\Sigma, \star, \text{emp})$ with

$\star : \Sigma \times \Sigma \nrightarrow \Sigma$ associative, commutative, cancellative

emp set of units

- Flow graphs form a **separation algebra:**
  - ➡ framing: cut the graph, maintain inflow from **each node** outside (assoc. + cancel.).
  - ➡ composition: defined if inflow & outflow match + **no vanishing flow.**

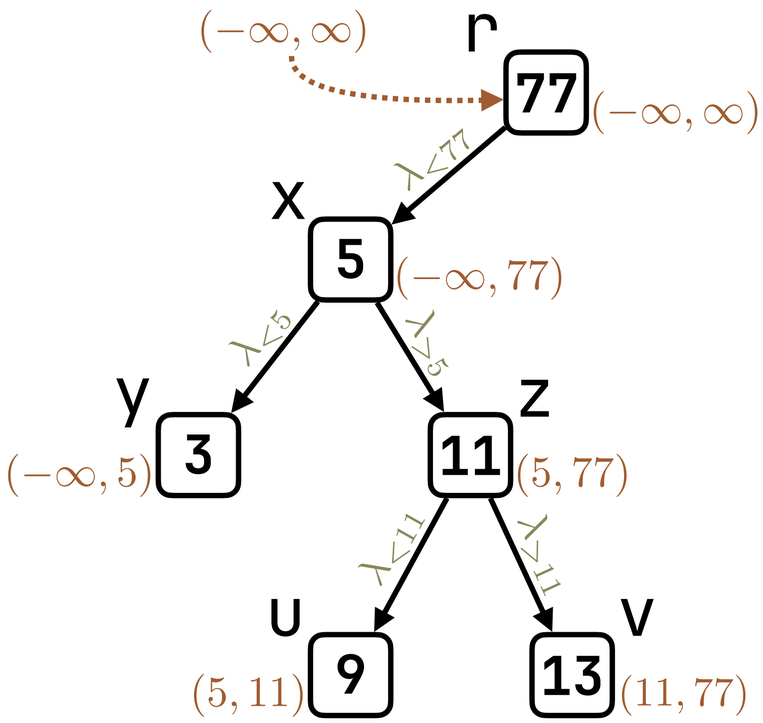
Separation

$(\Sigma, \star, \text{emp})$ with

$\star : \Sigma \times \Sigma \nrightarrow \Sigma$ associative, commutative, cancellative

emp set of units

- Flow graphs form a **separation algebra:**
  - ➡ framing: cut the graph, maintain inflow from **each node** outside (assoc. + cancel.).
  - ➡ composition: defined if inflow & outflow match + **no vanishing flow.**



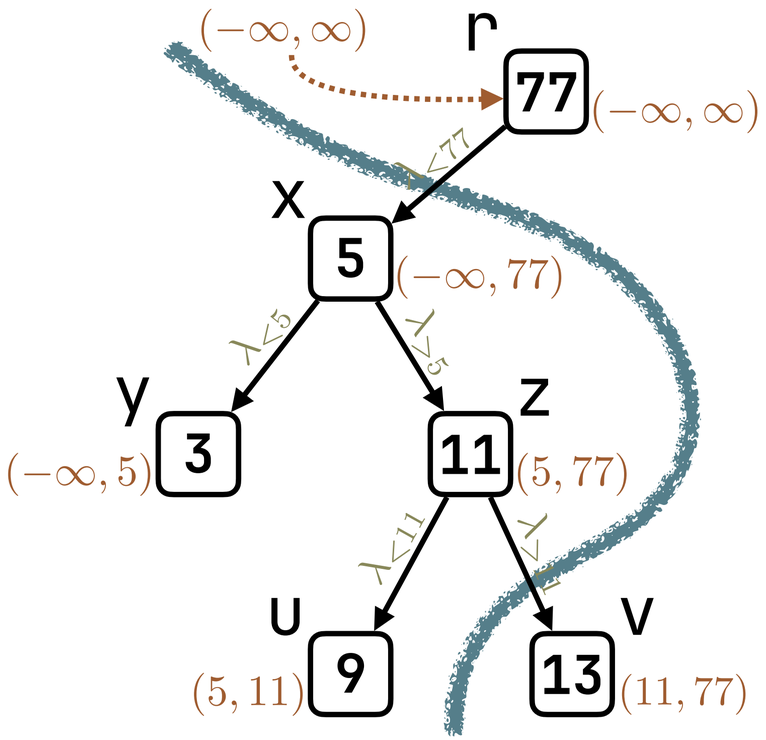
Separation

$(\Sigma, \star, \text{emp})$ with

$\star : \Sigma \times \Sigma \nrightarrow \Sigma$ associative, commutative, cancellative

emp set of units

- Flow graphs form a **separation algebra:**
  - ➡ framing: cut the graph, maintain inflow from **each node** outside (assoc. + cancel.).
  - ➡ composition: defined if inflow & outflow match + **no vanishing flow.**



$=$

$*$

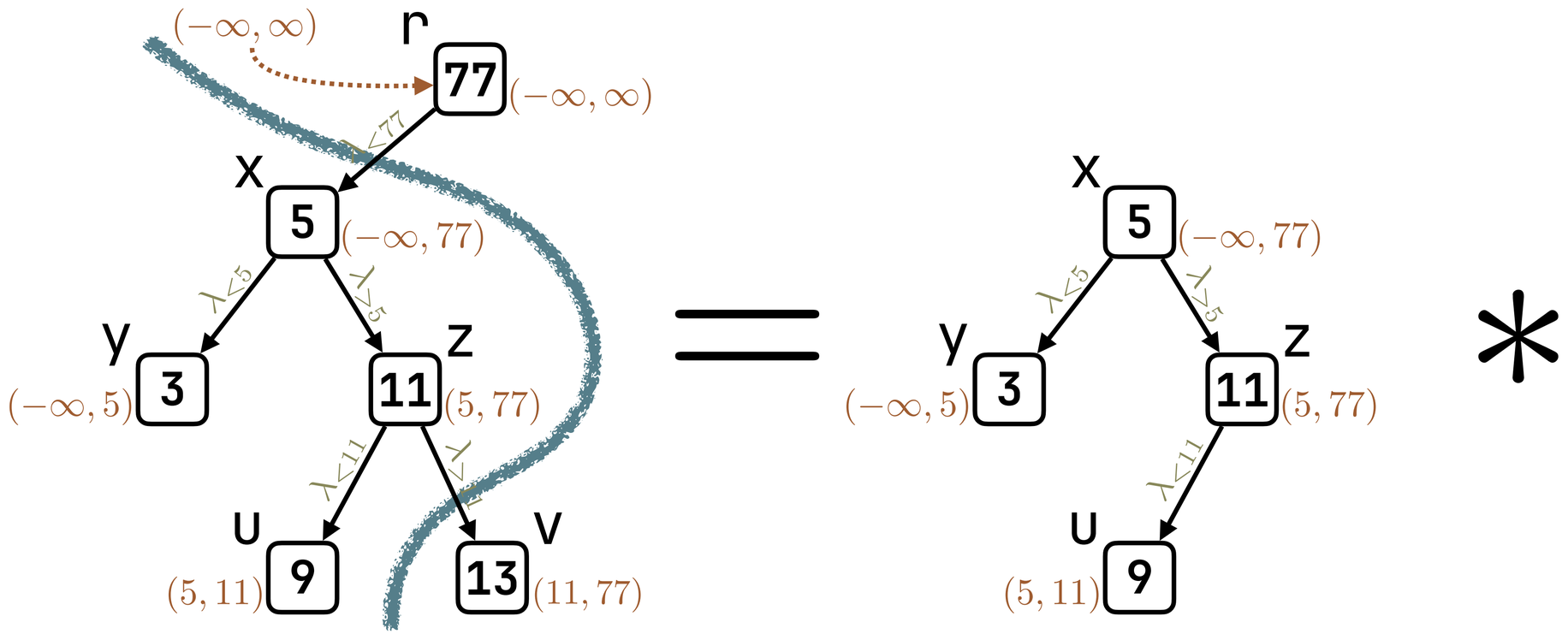
Separation

$(\Sigma, \star, \text{emp})$ with

$\star : \Sigma \times \Sigma \nrightarrow \Sigma$ associative, commutative, cancellative

emp set of units

- Flow graphs form a **separation algebra:**
  - ➡ framing: cut the graph, maintain inflow from **each node** outside (assoc. + cancel.).
  - ➡ composition: defined if inflow & outflow match + **no vanishing flow.**



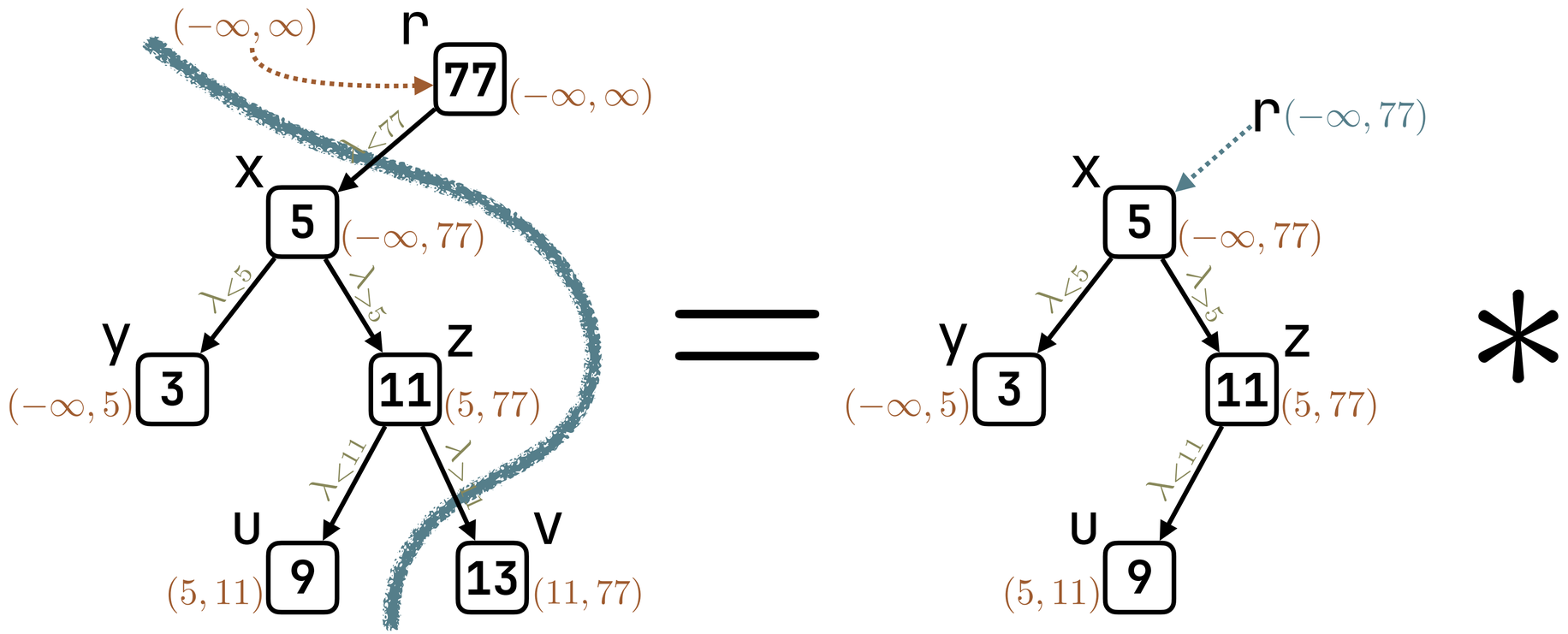
Separation

$(\Sigma, \star, \text{emp})$ with

$\star : \Sigma \times \Sigma \nrightarrow \Sigma$ associative, commutative, cancellative

emp set of units

- Flow graphs form a **separation algebra:**
  - ➡ framing: cut the graph, maintain inflow from **each node** outside (assoc. + cancel.).
  - ➡ composition: defined if inflow & outflow match + **no vanishing flow.**



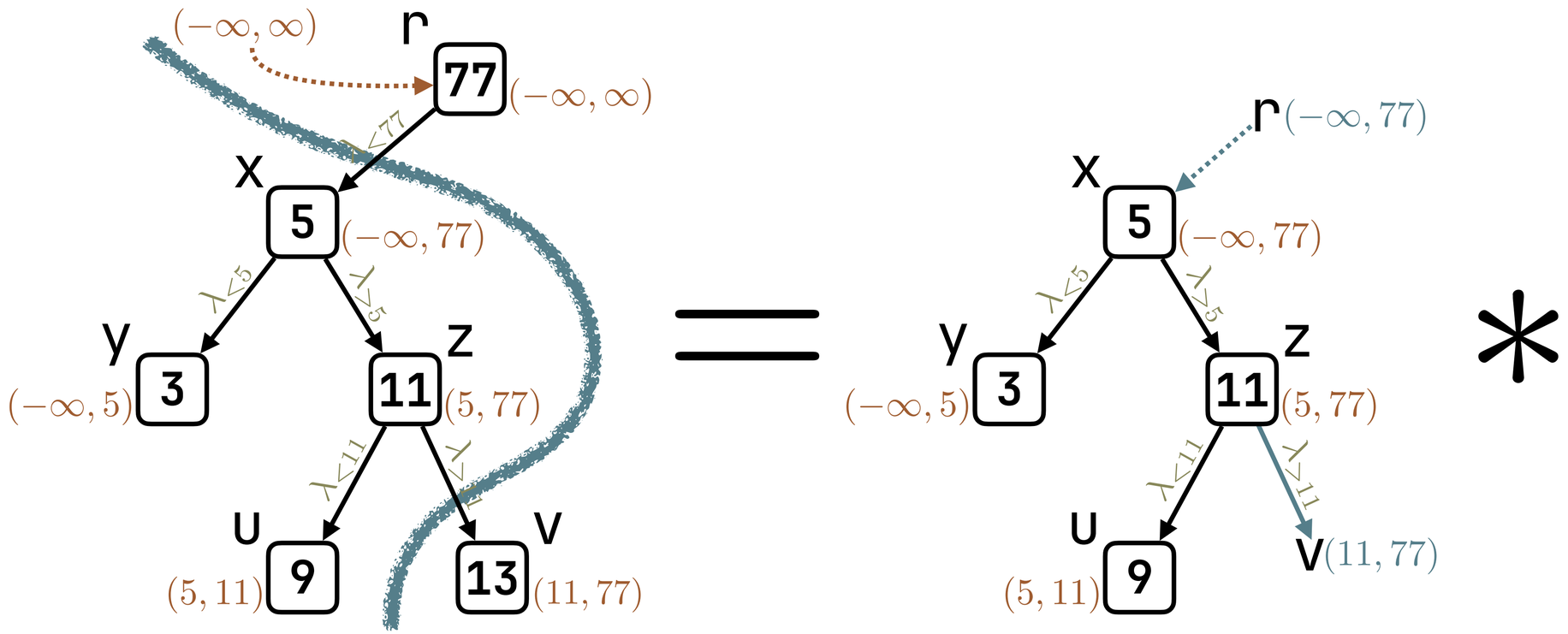
Separation

$(\Sigma, \star, \text{emp})$ with

$\star : \Sigma \times \Sigma \nrightarrow \Sigma$ associative, commutative, cancellative

emp set of units

- Flow graphs form a **separation algebra:**
  - ➡ framing: cut the graph, maintain inflow from **each node** outside (assoc. + cancel.).
  - ➡ composition: defined if inflow & outflow match + **no vanishing flow.**



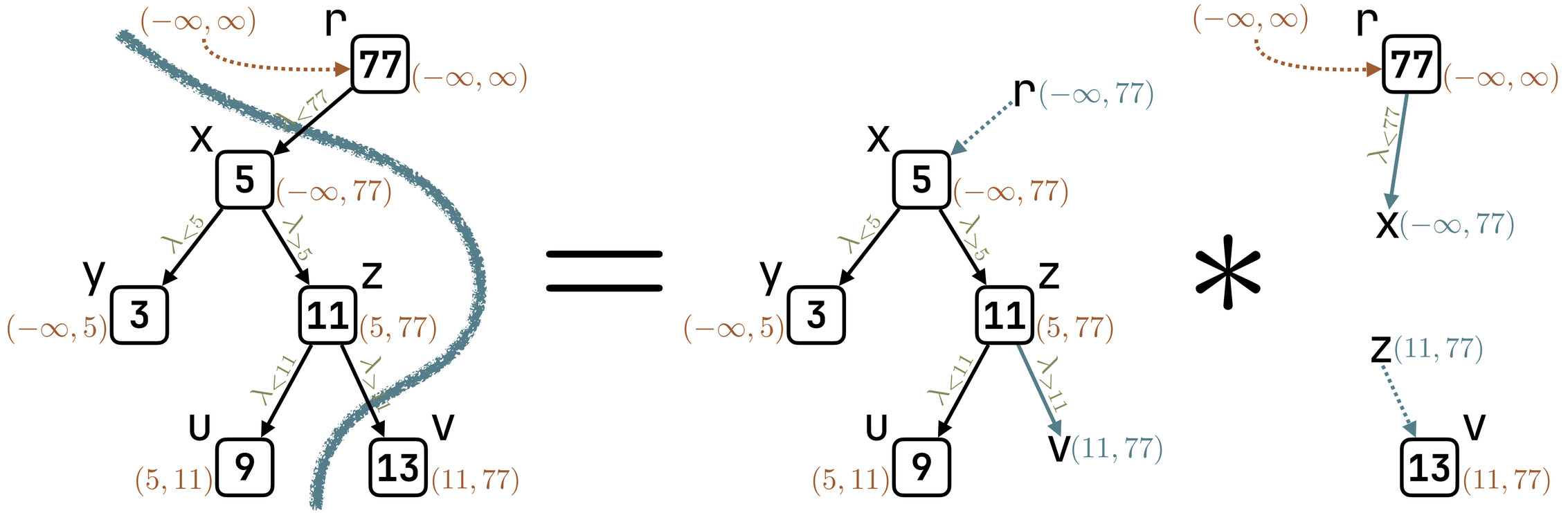
Separation

$(\Sigma, \star, \mathrm{emp})$ with

$\star : \Sigma \times \Sigma \nrightarrow \Sigma$ associative, commutative, cancellative

emp set of units

- Flow graphs form a **separation algebra:**
  - ➡ framing: cut the graph, maintain inflow from **each node** outside (assoc. + cancel.).
  - ➡ composition: defined if inflow & outflow match + **no vanishing flow.**



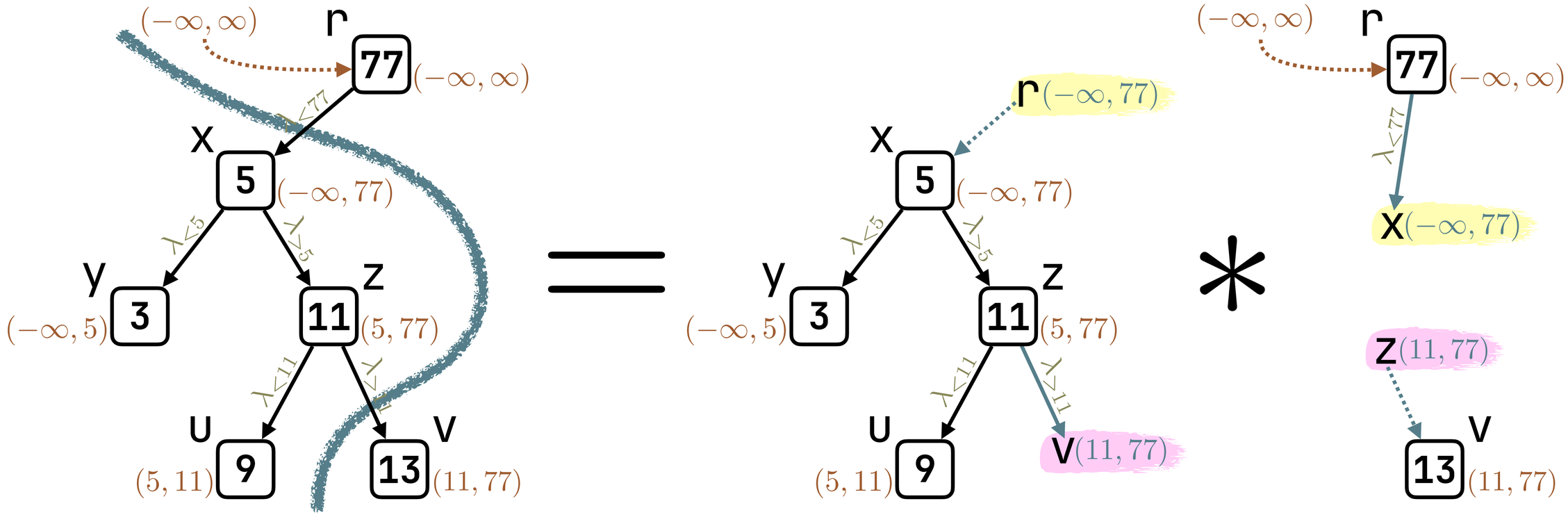
Separation

$(\Sigma, \star, \text{emp})$ with

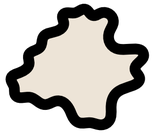
$\star : \Sigma \times \Sigma \nrightarrow \Sigma$ associative, commutative, cancellative

emp set of units

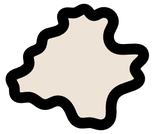
- Flow graphs form a **separation algebra:**
  - ➡ framing: cut the graph, maintain inflow from **each node** outside (assoc. + cancel.).
  - ➡ composition: defined if inflow & outflow match + **no vanishing flow.**

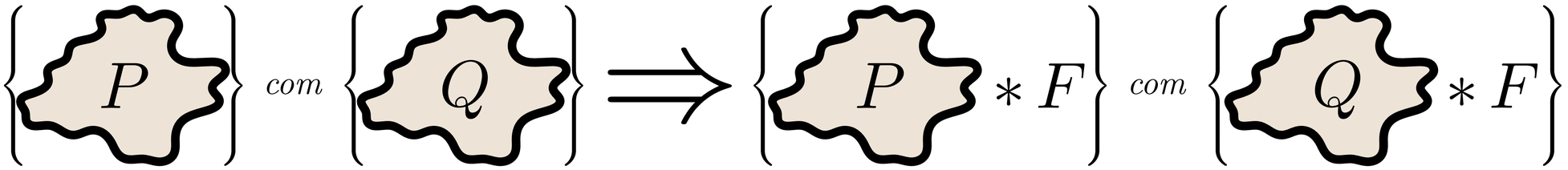


Frame-Preserving Updates

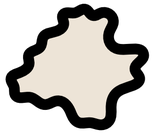
- Footprint
  - the region  affected by an update

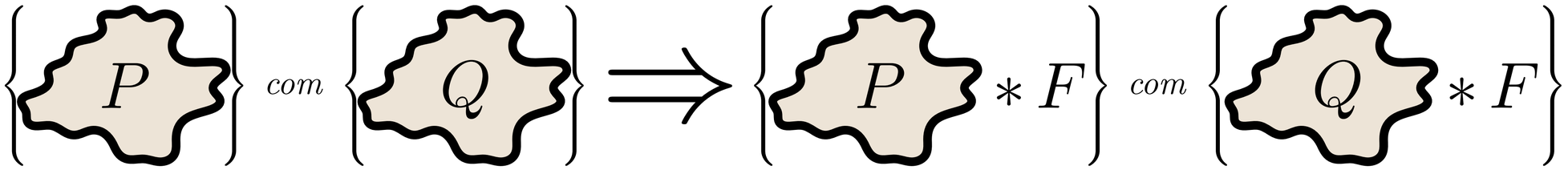
Frame-Preserving Updates

- Footprint
  - the region  affected by an update
  - is **frame-preserving** if, for all F:

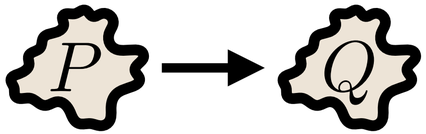


Frame-Preserving Updates

- Footprint
  - the region  affected by an update
  - is **frame-preserving** if, for all F:

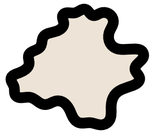


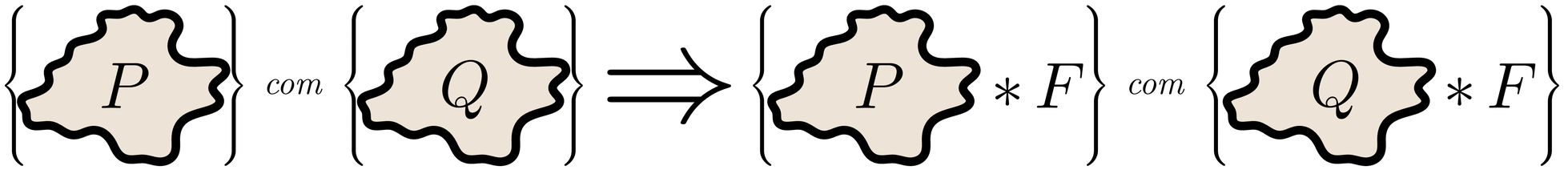
- Theorem:

  Update  frame-preserving

  if  and  have the same **inflow-outflow transformer** (up to lfp).

Frame-Preserving Updates

- Footprint
  - the region  affected by an update
  - is **frame-preserving** if, for all F:



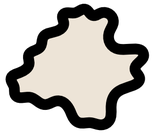
- Theorem:

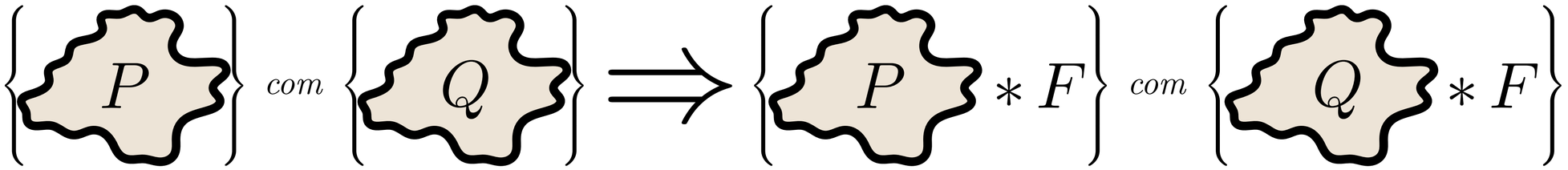
Bekić's lemma

Update frame-preserving

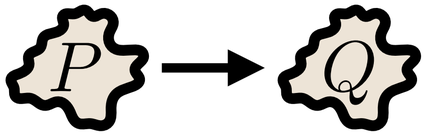
if and have the same **inflow-outflow transformer** (up to lfp).

Frame-Preserving Updates

- Footprint
  - the region  affected by an update
  - is **frame-preserving** if, for all F:



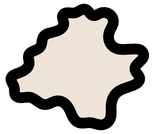
- Theorem:

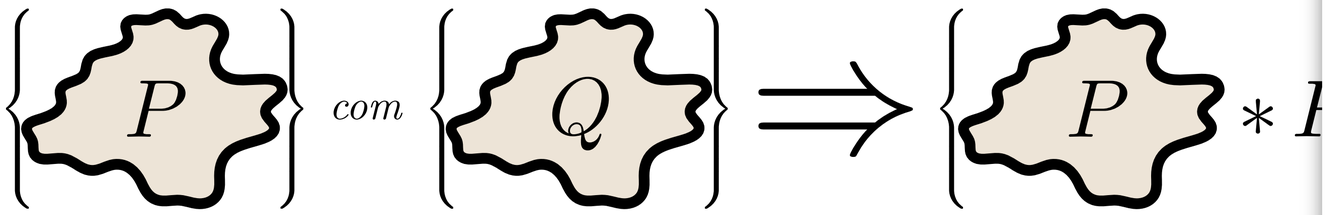
  Update  frame-preserving

  if  and  have the same **inflow-outflow transformer** (up to lfp).

Bekić's lemma

Frame-Preserving Updates

- Footprint
  - the region  affected by an update
  - is **frame-preserving** if, for all F:



- Theorem:

Bekić's lemma

Update frame-preserving

if and have the same **inflow-outflo**

[TACAS'23]

Make flows small again: revisiting the flow framework

Roland Meyer[1], Thomas Wies[2], and Sebastian Wolff[2(✉)]

[1] TU Braunschweig, Braunschweig, Germany, meyer@tu-bs.de
[2] New York University, New York, USA, {wies,sebastian.wolff}@cs.nyu.edu

Abstract We present a new flow framework for separation logic reasoning about programs that manipulate general graphs. The framework overcomes problems in earlier developments: it is based on standard fixed point theory, guarantees least flows, rules out vanishing flows, and has an easy to understand notion of footprint as needed for soundness of the frame rule. In addition, we present algorithms for automating the frame rule, which we evaluate on graph updates extracted from linearizability proofs for concurrent data structures. The evaluation demonstrates that our algorithms help to automate key aspects of these proofs that have previously relied on user guidance or heuristics.

Keywords: Separation Logic · Graph Algorithms · Frame Inference.

1 Introduction

The flow framework [23, 24] is an abstraction mechanism based on separation logic [5, 32, 40] that enables reasoning about global inductive invariants of general graphs in a local manner. The framework has proved useful to verify intricate algorithms that are difficult to handle by other techniques, such as the Priority Inheritance Protocol, object-oriented design patterns, and complex concurrent data structures [22, 24, 27, 34]. However, these efforts have also exposed some rough corners in the underlying meta theory that either limit expressivity or automation. In this paper, we propose a new meta theory for the flow framework that aims to strike a balance between these conflicting requirements. In addition, we present algorithms that aid proof automation.

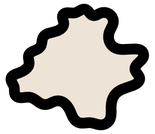
Background. The central notion of the flow framework is that of a *flow*. Given a commutative monoid $(\mathbb{M}, +, 0)$ (e.g. natural numbers with addition), and a graph with nodes X and an *edge function* $E\colon X \times X \to \mathbb{M} \to \mathbb{M}$, a flow is a function $\mathit{fl}\colon X \to \mathbb{M}$ that satisfies the *flow equation*:

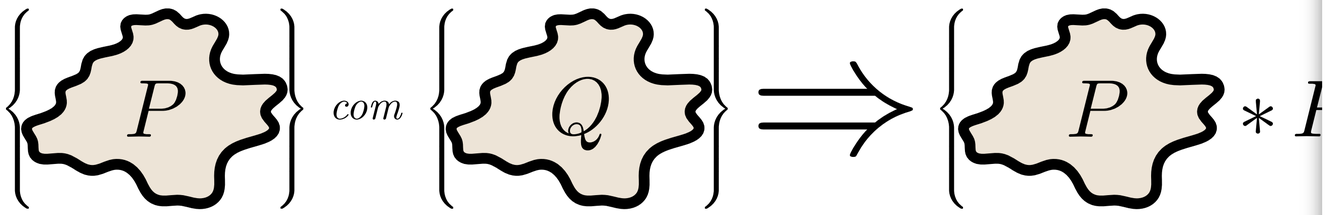
$$\forall x \in X.\quad \mathit{fl}(x) = \mathit{in}_x + \textstyle\sum_{y \in X} E_{(y,x)}(\mathit{fl}(y)) \ .$$

That is, fl is a fixed point of the function that assigns every node x an initial value $\mathit{in}_x \in \mathbb{M}$, its *inflow*, and then propagates these values through the graph according to the edge function. This is akin to a forward data flow analysis where the monoid operation $+$ is used as the join. By choosing an appropriate flow monoid, inflow, and edge function, one can express inductive properties of graphs (reachability, sortedness, etc.) in terms of conditions that refer only to each node's flow value $\mathit{fl}(x)$.

A graph endowed with an inflow and associated flow is a *flow graph*. An example flow graph h is shown on the right-hand side of Fig. 1a. Here, the flow value $\mathit{fl}(w)$ for

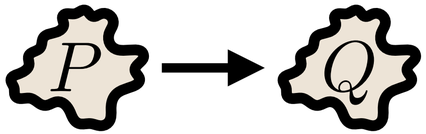
Frame-Preserving Updates

- Footprint
  - the region  affected by an update
  - is **frame-preserving** if, for all F:



Bekić's lemma

- Theorem:

  Update  frame-preserving

  if  and have the same **inflow-outflo**

[TACAS'23]

Make flows small again: revisiting the flow framework

Roland Meyer[1], Thomas Wies[2], and Sebastian Wolff[2(✉)]

[1] TU Braunschweig, Braunschweig, Germany, meyer@tu-bs.de
[2] New York University, New York, USA, {wies,sebastian.wolff}@cs.nyu.edu

Abstract We present a new flow framework for separation logic reasoning about programs that manipulate general graphs. The framework overcomes problems in earlier developments: it is based on standard fixed point theory, guarantees least flows, rules out vanishing flows, and has an easy to understand notion of footprint as needed for soundness of the frame rule. In addition, we present algorithms for automating the frame rule, which we evaluate on graph updates extracted from linearizability proofs for concurrent data structures. The evaluation demonstrates that our algorithms help to automate key aspects of these proofs that have previously relied on user guidance or heuristics.

Keywords: Separation Logic · Graph Algorithms · Frame Inference.

1 Introduction

The flow framework [23, 24] is an abstraction mechanism based on separation logic [5, 32, 40] that enables reasoning about global inductive invariants of general graphs in a local manner. The framework has proved useful to verify intricate algorithms that are difficult to handle by other techniques, such as the Priority Inheritance Protocol, object-oriented design patterns, and complex concurrent data structures [22, 24, 27, 34]. However, these efforts have also exposed some rough corners in the underlying meta theory that either limit expressivity or automation. In this paper, we propose a new meta theory for the flow framework that aims to strike a balance between these conflicting requirements. In addition, we present algorithms that aid proof automation.

Background. The central notion of the flow framework is that of a *flow*. Given a commutative monoid $(\mathbb{M}, +, 0)$ (e.g. natural numbers with addition), and a graph with nodes X and an *edge function* $E\colon X \times X \to \mathbb{M} \to \mathbb{M}$, a flow is a function $\mathit{fl}\colon X \to \mathbb{M}$ that satisfies the *flow equation*:

$$\forall x \in X.\quad \mathit{fl}(x) = \mathit{in}_x + \textstyle\sum_{y \in X} E_{(y,x)}(\mathit{fl}(y)) \ .$$

That is, fl is a fixed point of the function that assigns every node x an initial value $\mathit{in}_x \in \mathbb{M}$, its *inflow*, and then propagates these values through the graph according to the edge function. This is akin to a forward data flow analysis where the monoid operation $+$ is used as the join. By choosing an appropriate flow monoid, inflow, and edge function, one can express inductive properties of graphs (reachability, sortedness, etc.) in terms of conditions that refer only to each node's flow value $\mathit{fl}(x)$.

A graph endowed with an inflow and associated flow is a *flow graph*. An example flow graph h is shown on the right-hand side of Fig. 1a. Here, the flow value $\mathit{fl}(w)$ for

Contributions

Flow Framework

Reasoning about heap graphs without recursive predicates.

[POPL'18, ESOP'20, TACAS'23]

Hindsight

Non-fixed linearization points without prophecies.

[OOPSLA'22, PLDI'23]

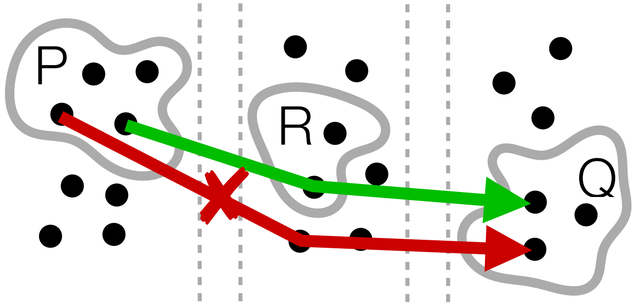
Decomposing Updates

Unbounded footprints without induction.

[OOPSLA'22, *under submission*]

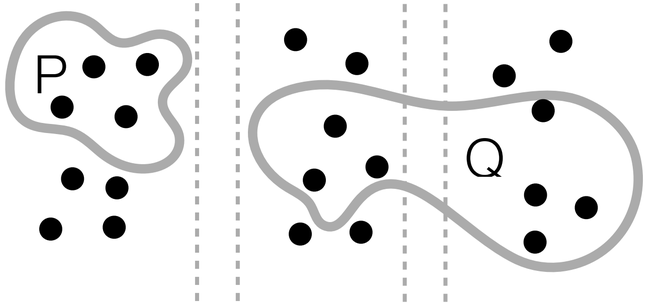
Goal

Show that a state (linearization point) has occurred.



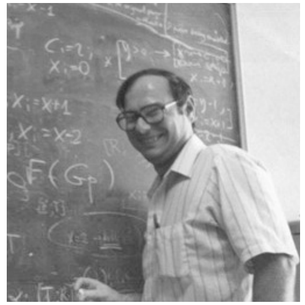
Problem

The state may undergo interference.
Its occurrence is not visible in the assertion.

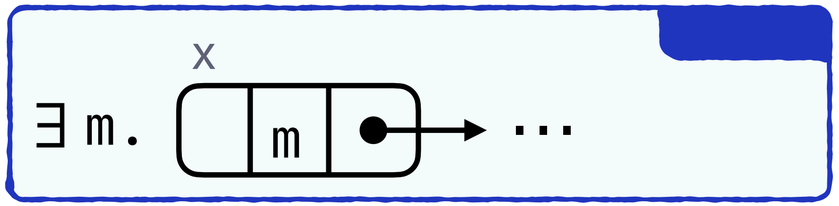


Solution

Introduce **past predicates** to keep track of the occurrence.
Past predicates are interference-free.



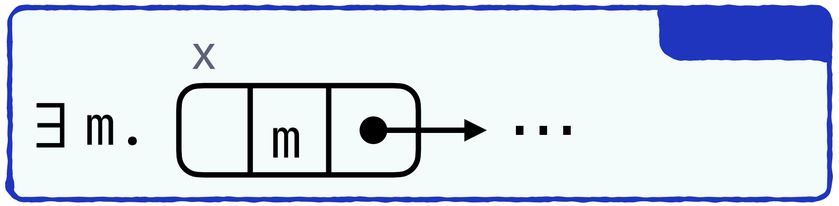
Example: `find(k)`



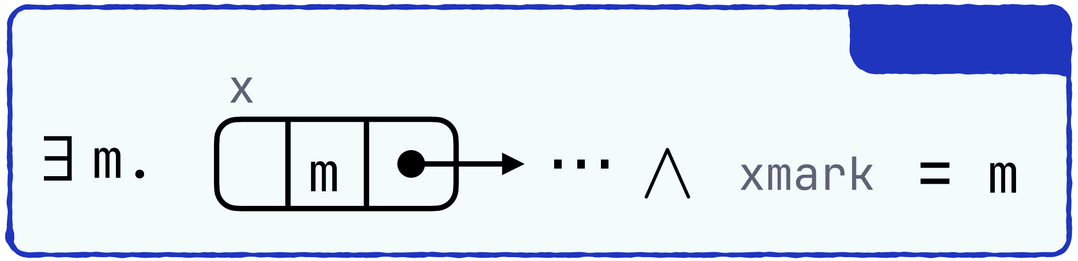

```
xmark = x→mark;
```

```
if (!xmark){
```

Example: `find(k)`

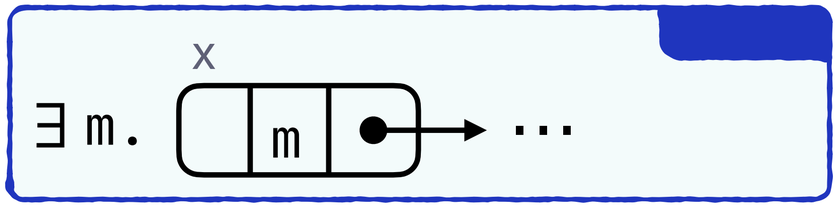



```
xmark = x→mark;
```

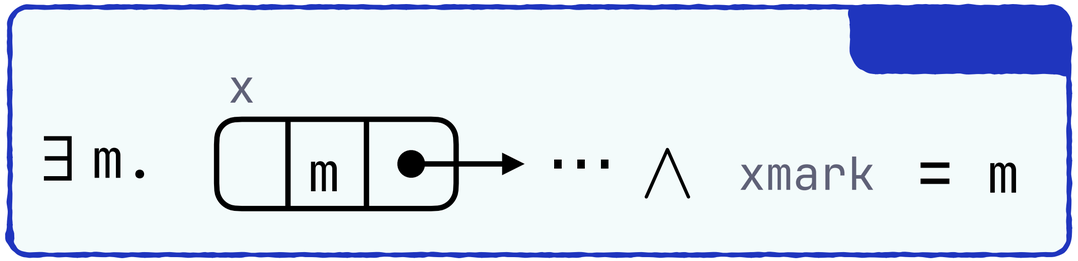


```
if (!xmark){
```

Example: `find(k)`



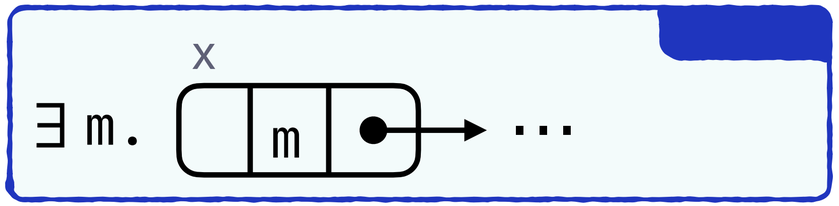

```
xmark = x→mark;
```



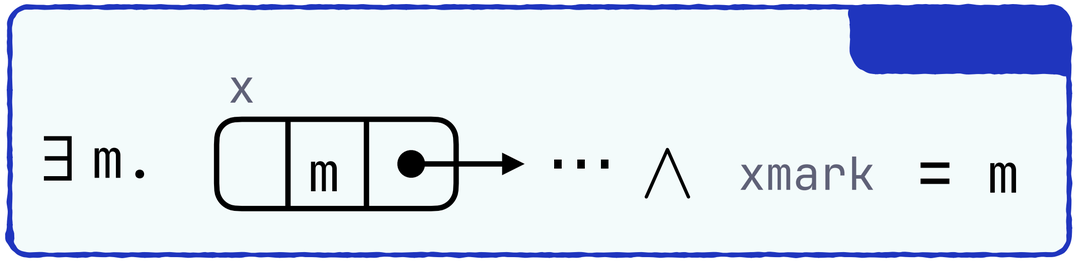
not interference-free

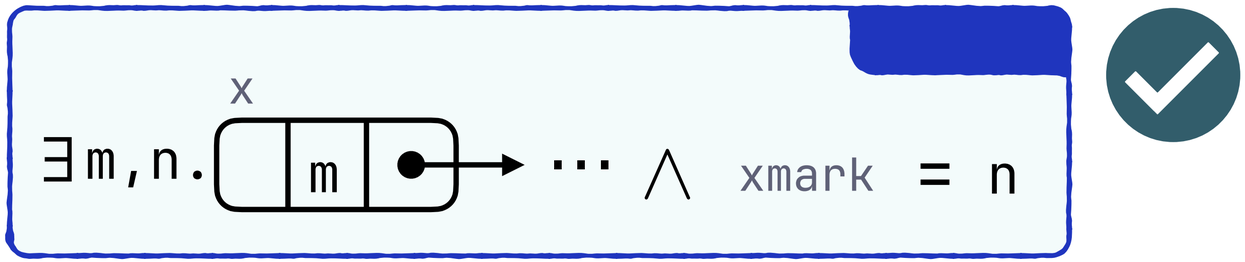
```
if (!xmark){
```

Example: `find(k)`



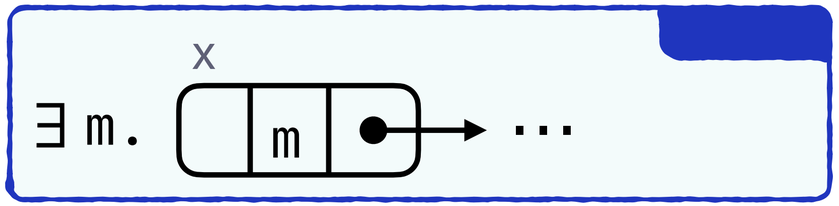

```
xmark = x→mark;
```



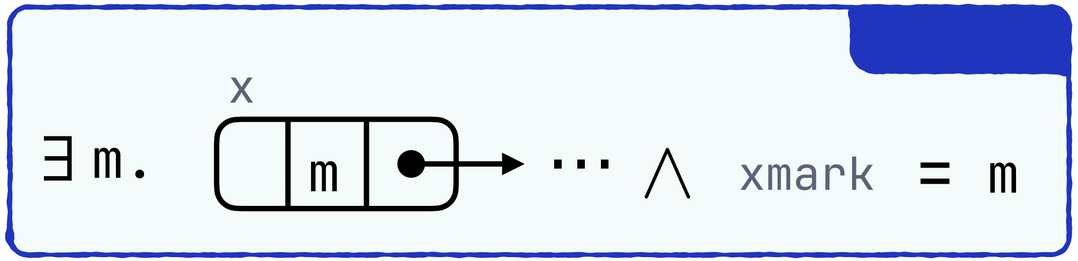



```
if (!xmark){
```

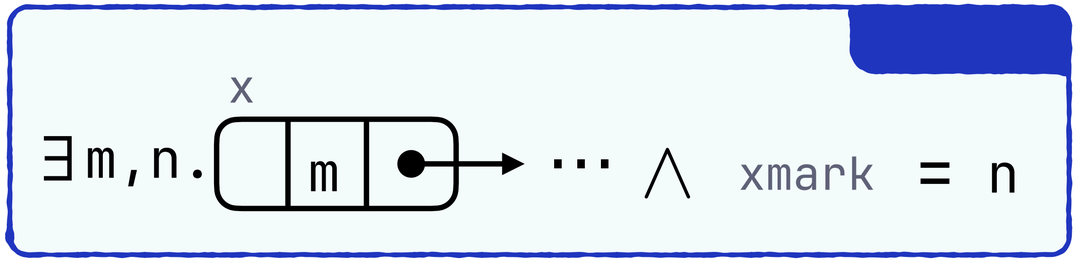
Example: `find(k)`




```
xmark = x→mark;
```



not interference-free



interference-free
but not helpful

```
if (!xmark){
```

State of the Art Solution

- **Prophecy variables**
  [Abadi and Lamport 1991; Vafeiadis 2008; Liang and Feng 2013; Jung et al. 2020]

  ➡ case split along all possible computations

  ➡ predict each computation **eagerly** upfront

  ➡ fixes the linearization point

State of the Art Solution

- **Prophecy variables**
  [Abadi and Lamport 1991; Vafeiadis 2008; Liang and Feng 2013; Jung et al. 2020]

  ➡ case split along all possible computations

  ➡ predict each computation **eagerly** upfront

  ➡ fixes the linearization point

difficult to automate
scalability

Separation Logic over Histories

- Let $(\Sigma, \star, \text{emp})$ be the separation algebra of states.

Separation Logic over Histories

- Let $(\Sigma, \star, \mathrm{emp})$ be the separation algebra of states.
- 

separation algebra of computations $(\Sigma^+, \star^+, \mathrm{emp}^+)$.

Separation Logic over Histories

- Let $(\Sigma, \star, \text{emp})$ be the separation algebra of states.
- 

separation algebra of computations $(\Sigma^+, \star^+, \text{emp}^+)$.
- State predicates are sets of states $Q \subseteq \Sigma$.

Separation Logic over Histories

- Let $(\Sigma, \star, \text{emp})$ be the separation algebra of states.
- 

separation algebra of computations $(\Sigma^+, \star^+, \text{emp}^+)$.
- State predicates are sets of states $Q \subseteq \Sigma$.
- 

computation predicates:

$$= \quad \Sigma^* . Q$$

Separation Logic over Histories

- Let $(\Sigma, \star, \text{emp})$ be the separation algebra of states.
- 

separation algebra of computations $(\Sigma^+, \star^+, \text{emp}^+)$.
- State predicates are sets of states $Q \subseteq \Sigma$.
- 

computation predicates:

$$= \quad \Sigma^* \,.\, Q$$

$$= \quad \Sigma^* \,.\, P \,.\, \Sigma^*$$

Separation Logic over Histories

- Let $(\Sigma, \star, \text{emp})$ be the separation algebra of states.
- NEW **separation algebra of computations** $(\Sigma^+, \star^+, \text{emp}^+)$.
- State predicates are sets of states $Q \subseteq \Sigma$.
- NEW **computation predicates**:

$$Q^{\text{Now}} = \Sigma^* . Q$$

$$P^{\text{Past}} = \Sigma^* . P . \Sigma^*$$

- Quantify on the level of computations (state-independently).

NEW

Separation Logic over Histories

- Let $(\Sigma, \star, \text{emp})$ be the separation algebra of states
- NEW **separation algebra of computations** $(\Sigma^+, \ldots$
- State predicates are sets of states $Q \subseteq \Sigma$.
- NEW **computation predicates**:

$$Q^{\textbf{Now}} = \Sigma^* . Q$$

$$P^{\textbf{Past}} = \Sigma^* . P . \Sigma^*$$

- Quantify on the level of computations (state-indepe

NEW

[OOPSLA'22]

A Concurrent Program Logic with a Future and History

ROLAND MEYER, TU Braunschweig, Germany
THOMAS WIES, New York University, USA
SEBASTIAN WOLFF, New York University, USA

Verifying fine-grained optimistic concurrent programs remains an open problem. Modern program logics provide abstraction mechanisms and compositional reasoning principles to deal with the inherent complexity. However, their use is mostly confined to pencil-and-paper or mechanized proofs. We devise a new separation logic geared towards the lacking automation. While local reasoning is known to be crucial for automation, we are the first to show how to retain this locality for (i) reasoning about inductive properties without the need for ghost code, and (ii) reasoning about computation histories in hindsight. We implemented our new logic in a tool and used it to automatically verify challenging concurrent search structures that require inductive properties and hindsight reasoning, such as the Harris set.

CCS Concepts: • **Theory of computation → Separation logic**; **Hoare logic**; **Automated reasoning**; *Program verification*; Programming logic.

Additional Key Words and Phrases: Linearizability, Non-blocking Data Structures, Harris Set

ACM Reference Format:
Roland Meyer, Thomas Wies, and Sebastian Wolff. 2022. A Concurrent Program Logic with a Future and History. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 174 (October 2022), 30 pages. https://doi.org/10.1145/3563337

1 INTRODUCTION

Concurrency comes at a cost, at least, in terms of increased effort when verifying program correctness. There has been a proliferation of concurrent program logics that provide an arsenal of reasoning techniques to address this challenge [Bell et al. 2010; Delbianco et al. 2017; Elmas et al. 2010; Fu et al. 2010; Gotsman et al. 2013; Gu et al. 2018; Hemed et al. 2015; Jung et al. 2018; Liang and Feng 2013; Manna and Pnueli 1995; Parkinson et al. 2007; Sergey et al. 2015; Vafeiadis and Parkinson 2007]. In addition, a number of general approaches have been developed to help structure the high-level proof argument [Feldman et al. 2018, 2020; Kragl et al. 2020; O'Hearn et al. 2010; Shasha and Goodman 1988]. However, the use of these techniques has been mostly confined to manual proofs done on paper, or mechanized proofs constructed in interactive proof assistants. We distill from these works a concurrent separation logic suitable for automating the construction of local correctness proofs for highly concurrent data structures. We focus on concurrent search structures (sets and maps indexed by keys), but the developed techniques apply more broadly. Our guiding principle is to perform all inductive reasoning, both in time and space, in lock-step with the program execution. The reasoning about inductive properties of graph structures and computation histories is relegated to the meta-theory of the logic by choosing appropriate semantic models.

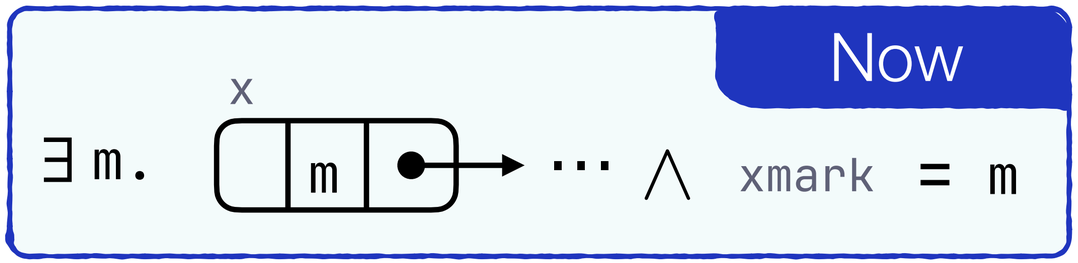
Running Example. We motivate our work using the Harris non-blocking set data structure [Harris 2001], which we will also use as a running example throughout the paper.

Authors' addresses: Roland Meyer, TU Braunschweig, Germany, roland.meyer@tu-bs.de; Thomas Wies, New York University, USA, wies@cs.nyu.edu; Sebastian Wolff, New York University, USA, sebastian.wolff@cs.nyu.edu.

2475-1421/2022/10-ART174
https://doi.org/10.1145/3563337

Example: `find(k)`

```
    ...
xmark = x→mark;
```

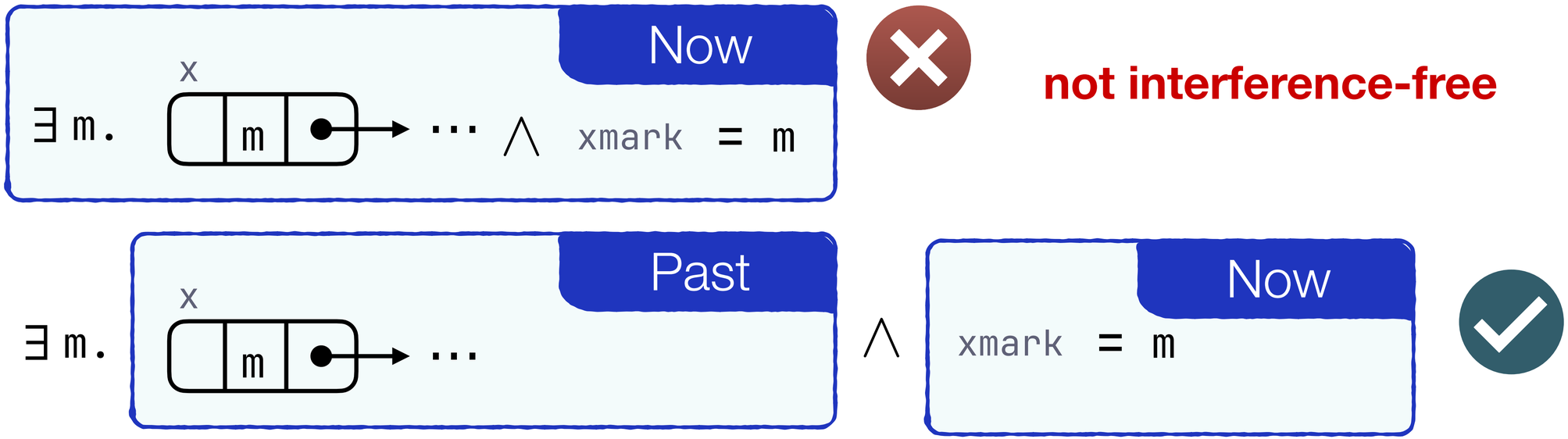


not interference-free

```
if (!xmark){
```

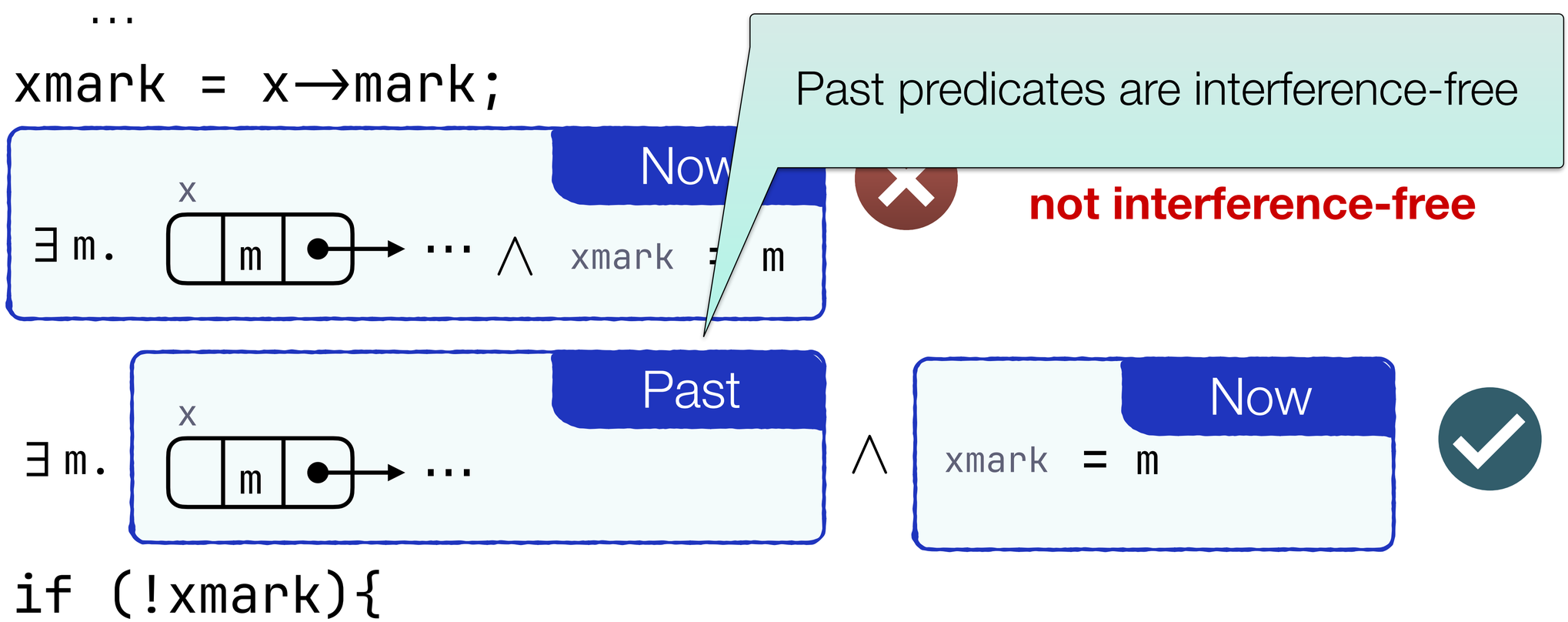
Example: `find(k)`

```
    ...
xmark = x→mark;
```



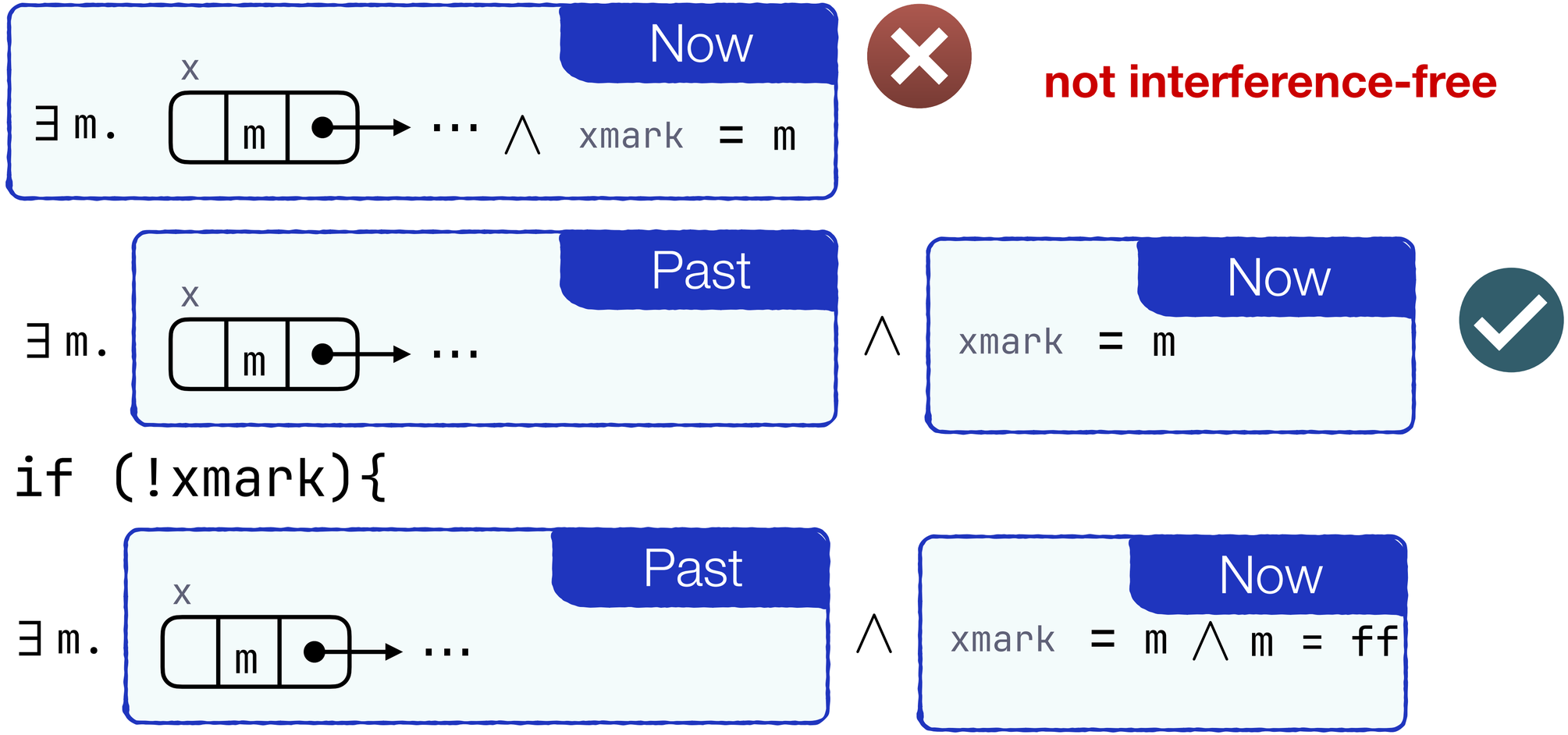

```
if (!xmark){
```

Example: `find(k)`



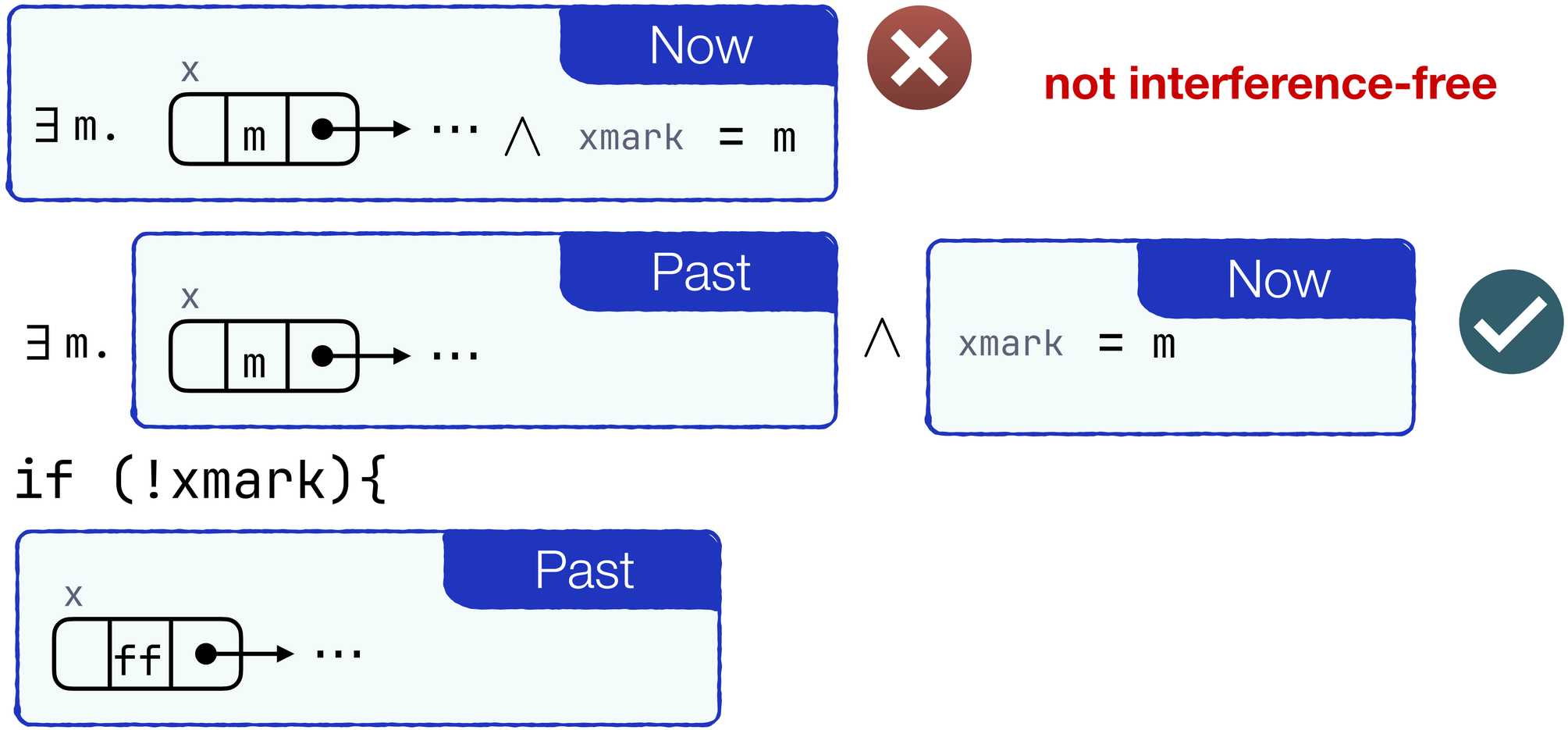
Example: `find(k)`

```
    ...
xmark = x→mark;
```



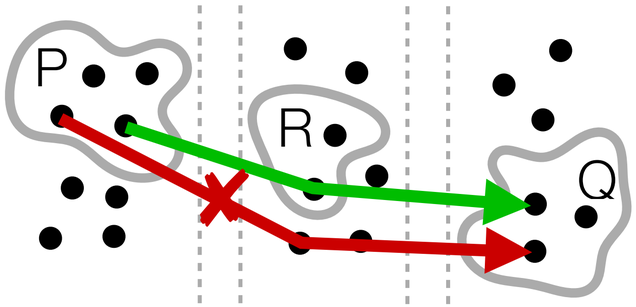
Example: `find(k)`

```
    ...
xmark = x→mark;
```



Goal

Show that a state (linearization point) has occurred.



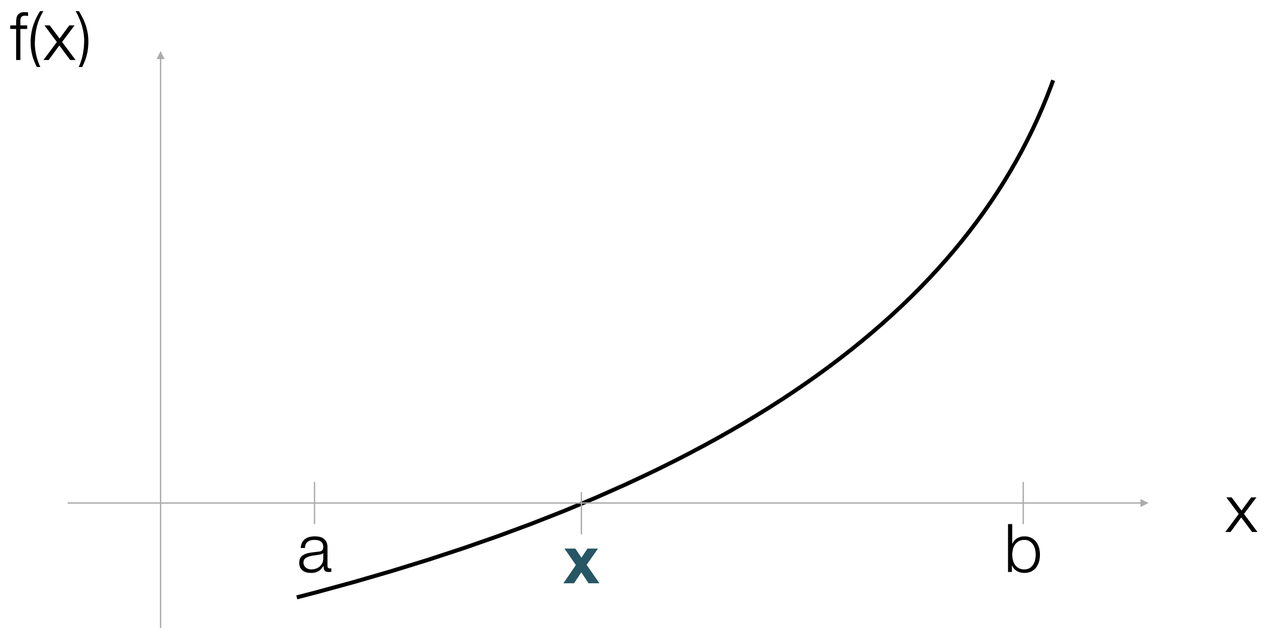
Problem

State-independent quantification not powerful enough.

Solution

Hindsight reasoning/temporal interpolation.

Hindsight Reasoning [O'Hearn et al. 2010; Lev-Ari et al. 2015Feldman et al. 2018, 2020]

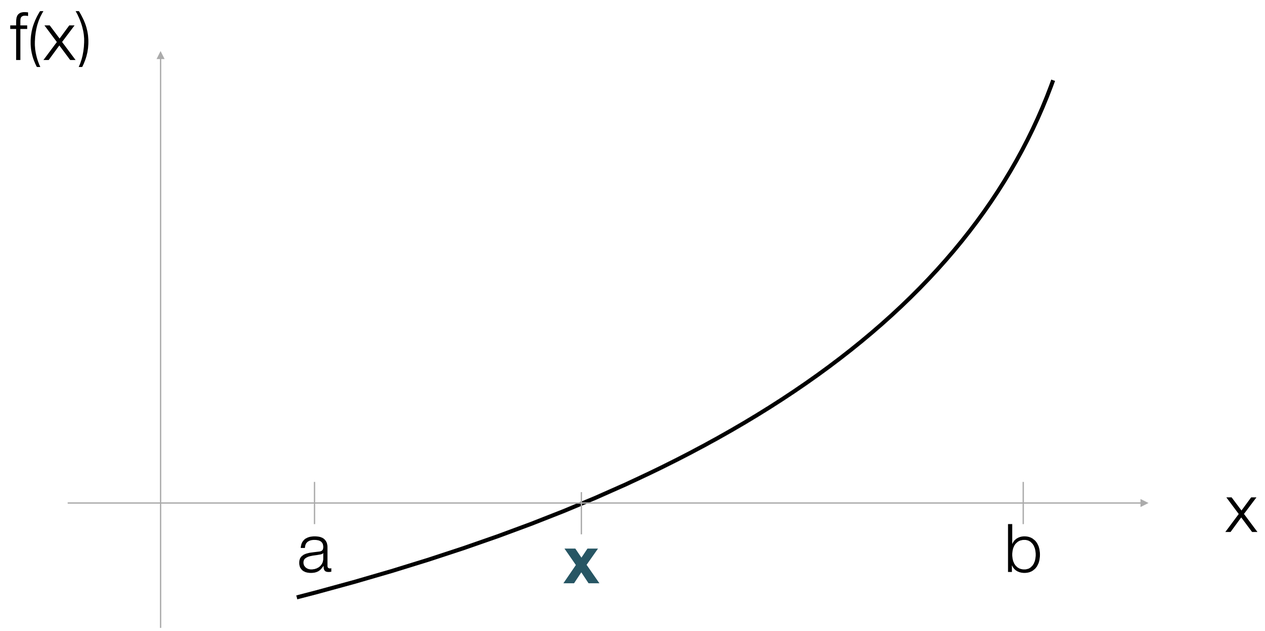


Nullstellensatz (Bolzano, 1817):

Let f be continuous on [a, b], f(a) < 0, and f(b) > 0.

Then **there is x** in [a, b] with f(x) = 0

Hindsight Reasoning [O'Hearn et al. 2010; Lev-Ari et al. 2015Feldman et al. 2018, 2020]



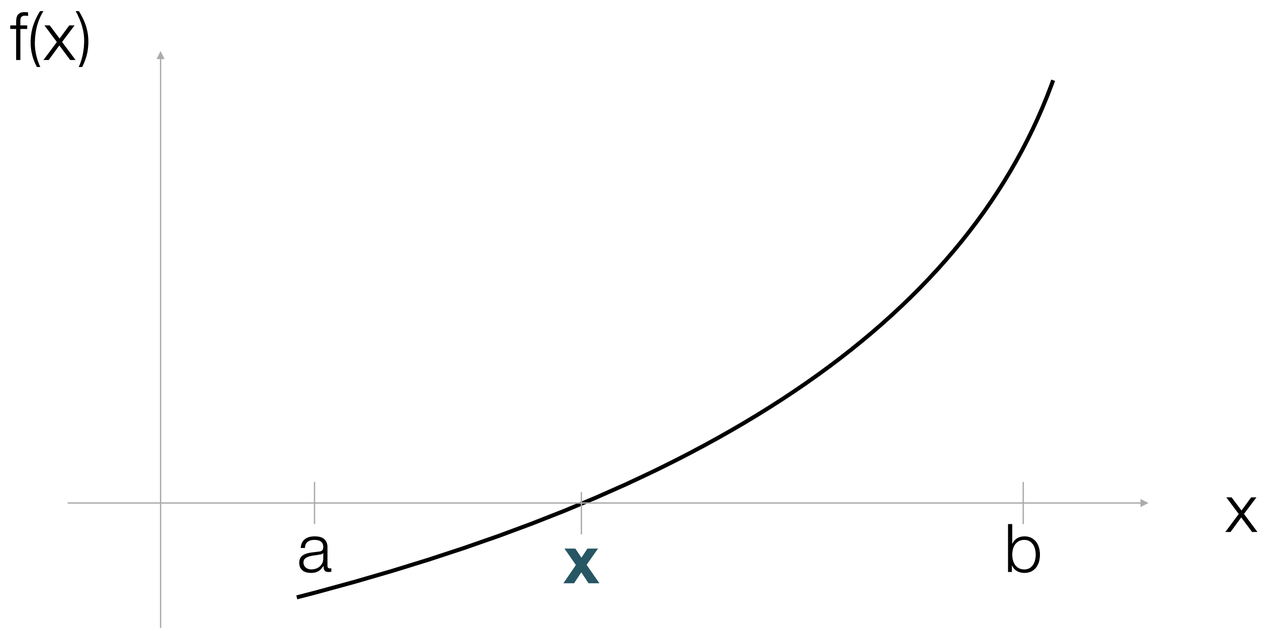
Nullstellensatz (Bolzano, 1817):

Let f be continuous on [a, b], f(a) < 0, and f(b) > 0.

Then **there is x** in [a, b] with f(x) = 0

— the **linearization point**.

Hindsight Reasoning [O'Hearn et al. 2010; Lev-Ari et al. 2015Feldman et al. 2018, 2020]



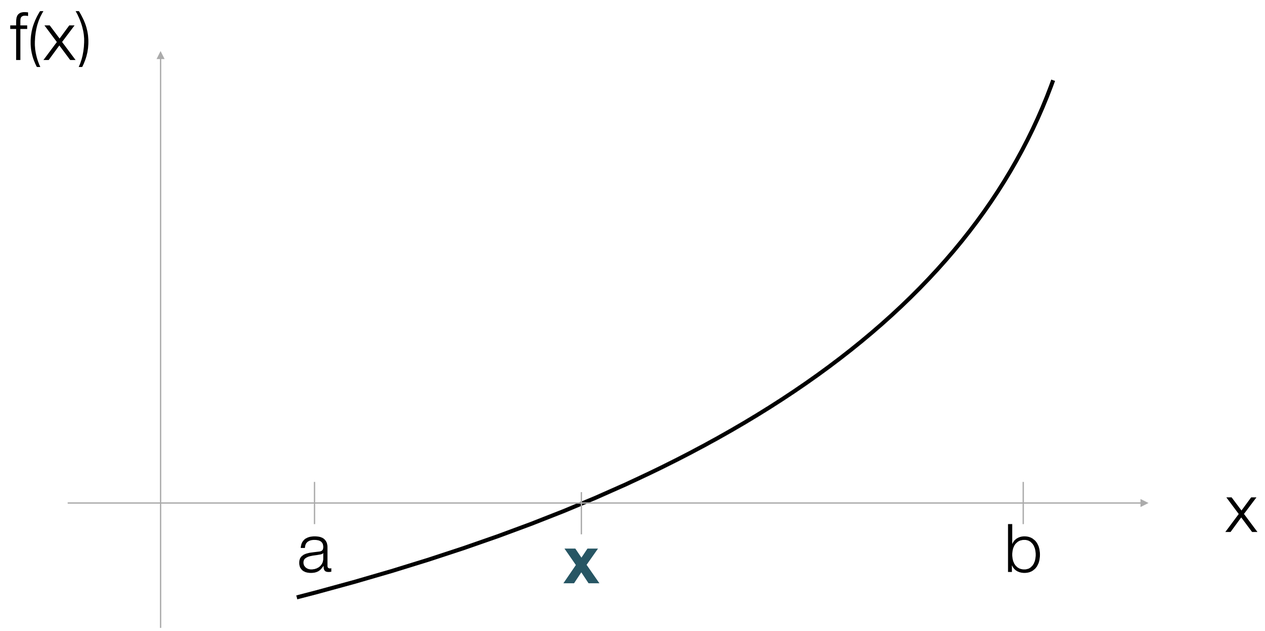
- ➡ record how computation unfolds
- ➡ derive what **must have happened**
- ➡ reflects programming practice

Nullstellensatz (Bolzano, 1817):

Let f be continuous on [a, b], f(a) < 0, and f(b) > 0.
Then **there is x** in [a, b] with f(x) = 0

— the **linearization point**.

Hindsight Reasoning [O'Hearn et al. 2010; Lev-Ari et al. 2015Feldman et al. 2018, 2020]



- ➡ record how computation unfolds
- ➡ derive what **must have happened**
- ➡ reflects programming practice

lazy approach,
easier to automate

Nullstellensatz (Bolzano, 1817):

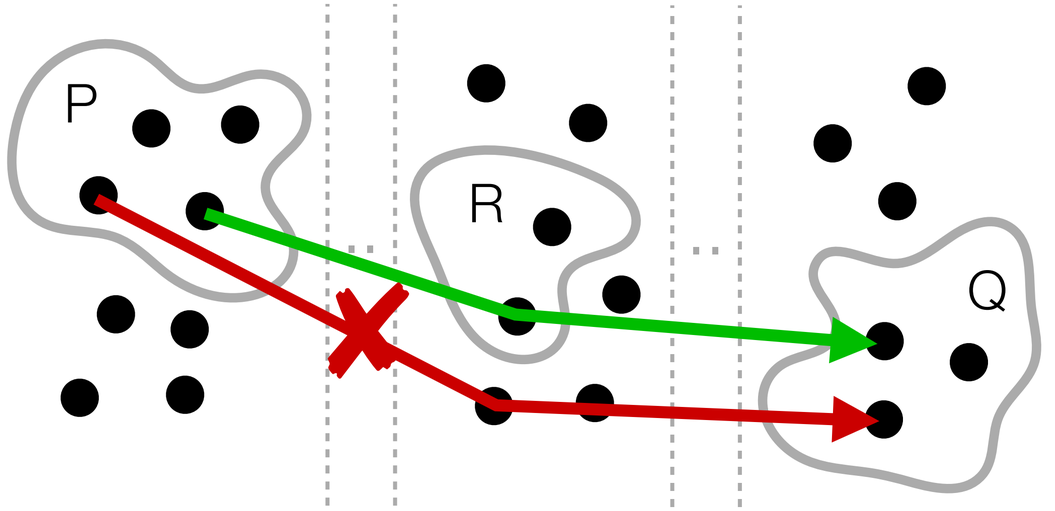
Let f be continuous on [a, b], f(a) < 0, and f(b) > 0.
Then **there is x** in [a, b] with f(x) = 0

— the **linearization point**.

Hindsight Reasoning

- Record execution history P **Past**

  not just current state Q **Now**



Hindsight Reasoning

- Record execution history P **Past**

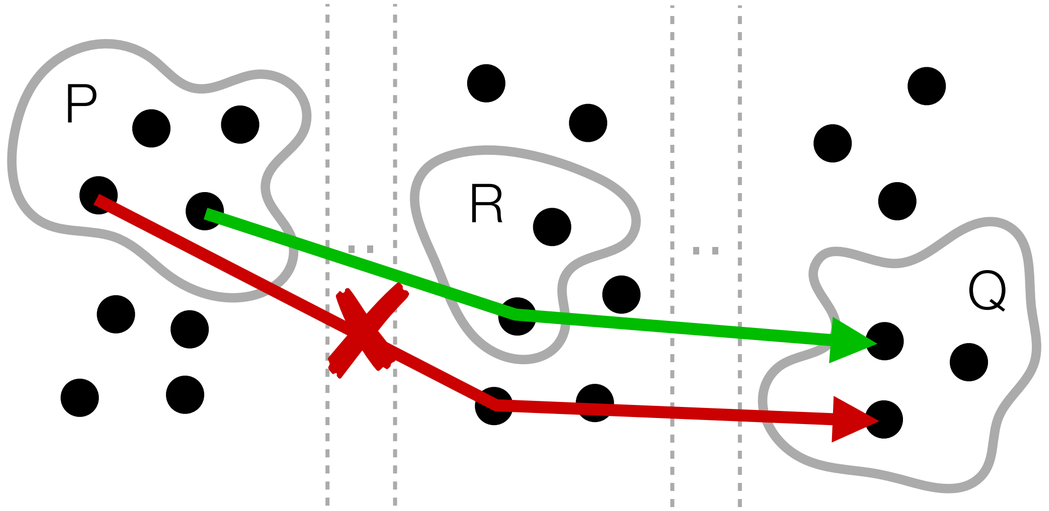
  not just current state Q **Now**

- Derive **inevitable** facts:

$$P \text{ Past} \wedge Q \text{ Now} \rightsquigarrow R \text{ Past}$$

if **all paths** from P to Q

pass through R.



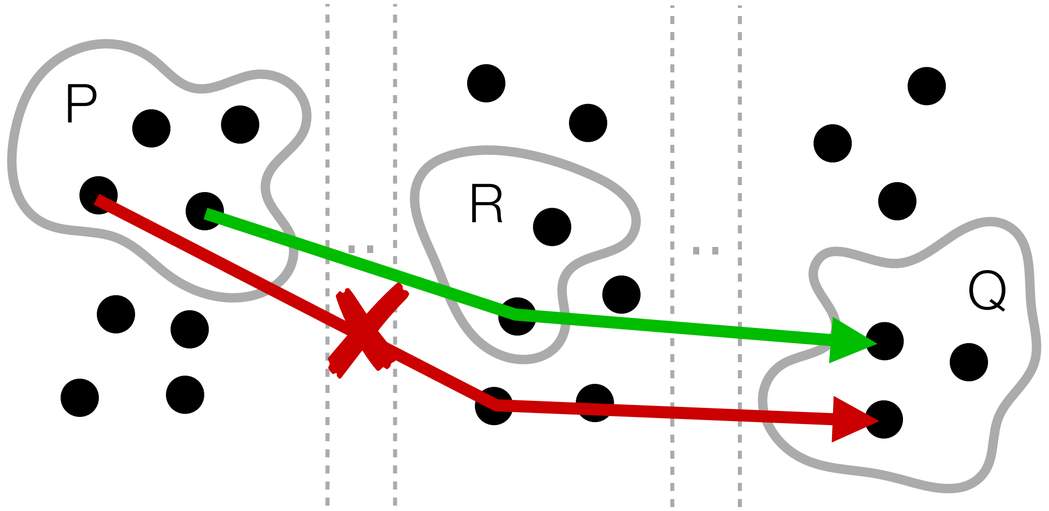
Hindsight Reasoning

- Record execution history P **Past**

  not just current state Q **Now**

- Derive **inevitable** facts:

$$P \text{ Past} \wedge Q \text{ Now} \rightsquigarrow R \text{ Past}$$



if **all paths** from P to Q

pass through R.

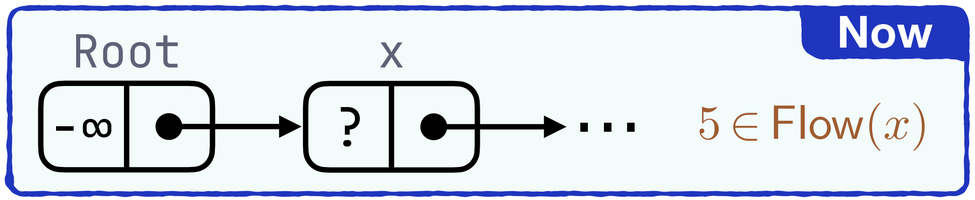
formed from **interferences**
in the Owicki-Gries proof

Example: `contains(5)`


```
x = Root→next;

while (x→key < 5):

    y = x→next;

    x = y;

return x→key == 5;
```

Example: `contains(5)`

```
x = Root→next;
```

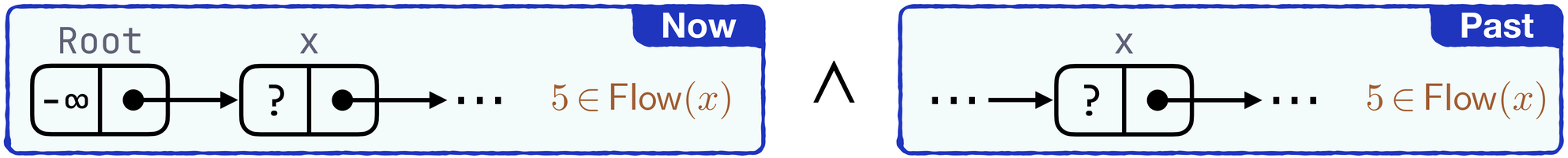


$\wedge$

```
while (x→key < 5):
    y = x→next;
    x = y;
return x→key == 5;
```

Example: `contains(5)`

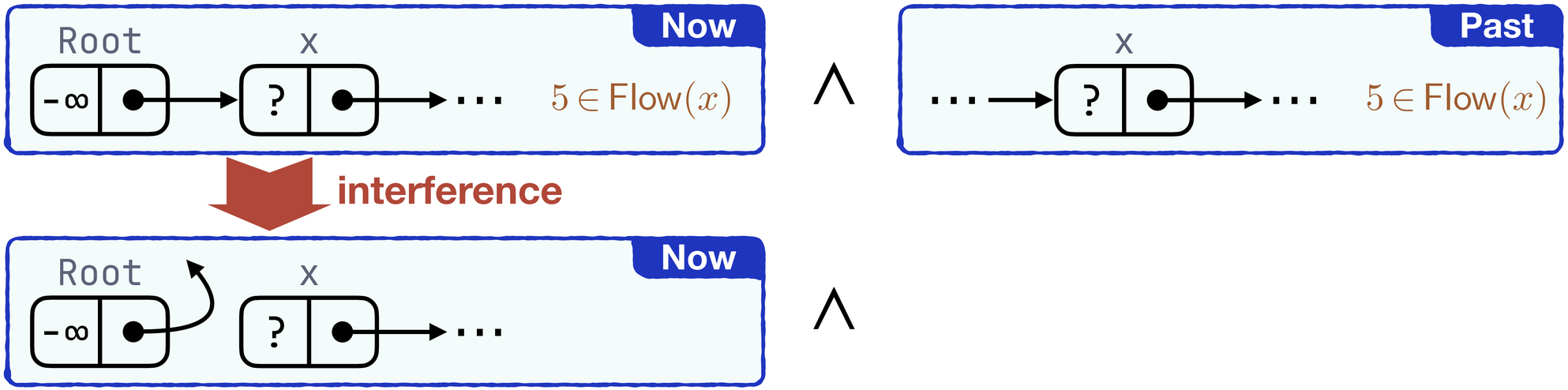
```
x = Root→next;
```




```
while (x→key < 5):
    y = x→next;
    x = y;
return x→key == 5;
```

Example: `contains(5)`

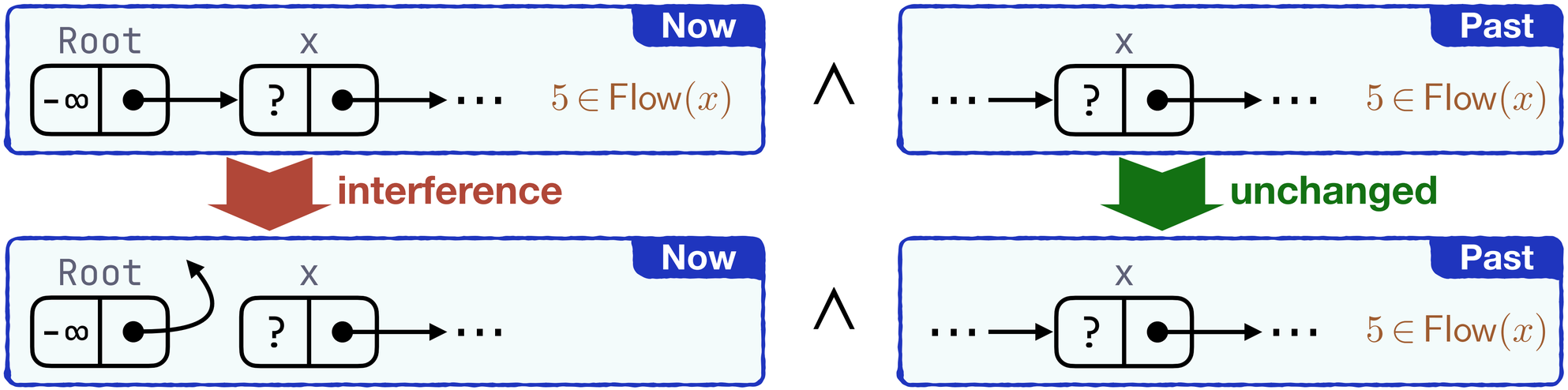
```
x = Root→next;
```




```
while (x→key < 5):
    y = x→next;
    x = y;
return x→key == 5;
```

Example: `contains(5)`

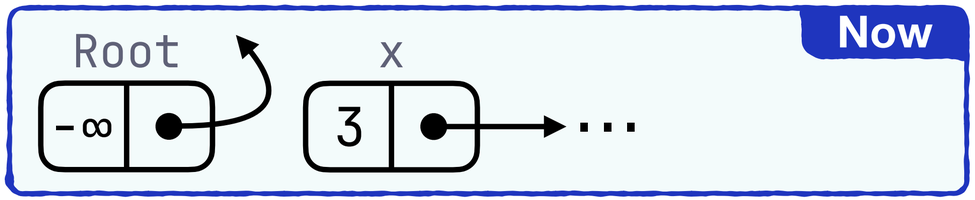
```
x = Root→next;
```



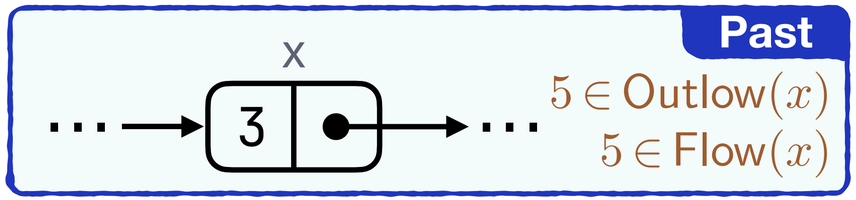

```
while (x→key < 5):
    y = x→next;
    x = y;
return x→key == 5;
```

Example: `contains(5)`

```
x = Root→next;

while (x→key < 5):
```



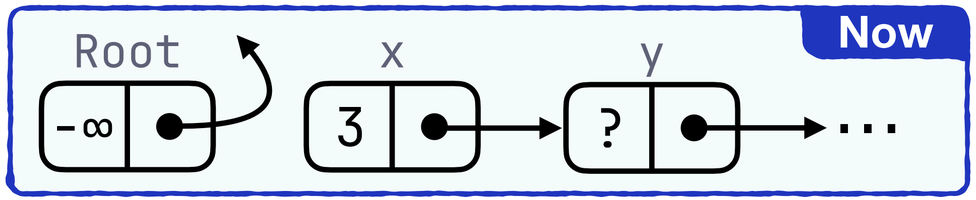
∧



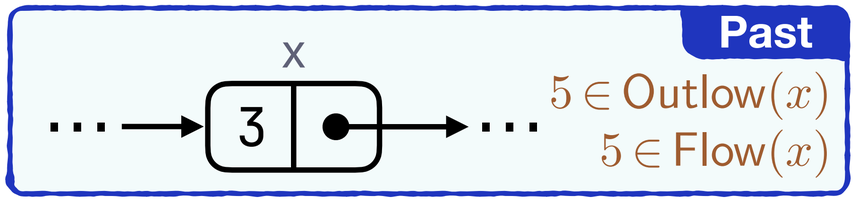

```
    y = x→next;

    x = y;

return x→key == 5;
```

Example: `contains(5)`

```
x = Root→next;
while (x→key < 5):
    y = x→next;
```



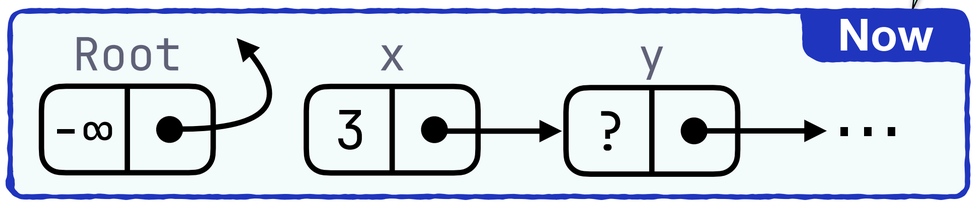
$\wedge$



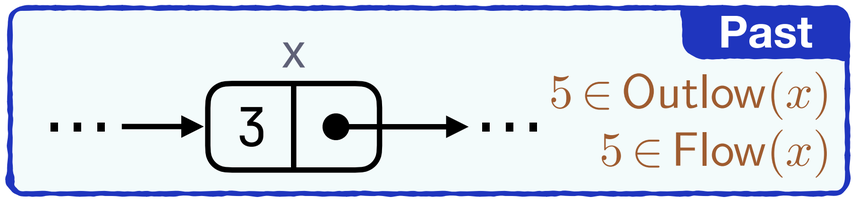

```
    x = y;
return x→key == 5;
```

Example: `contains(5)`

```
x = Root→next;
while (x→key < 5):
    y = x→next;
```



∧

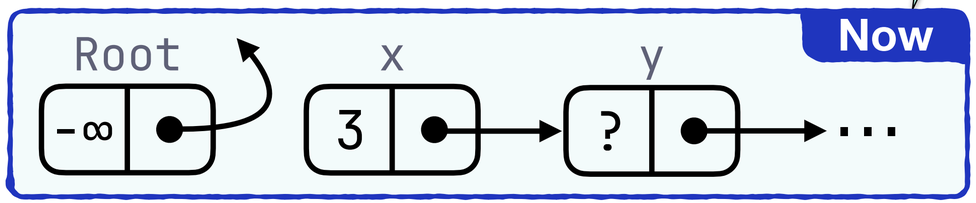


Traversal *learns* facts about **Now**, not the **Past**.

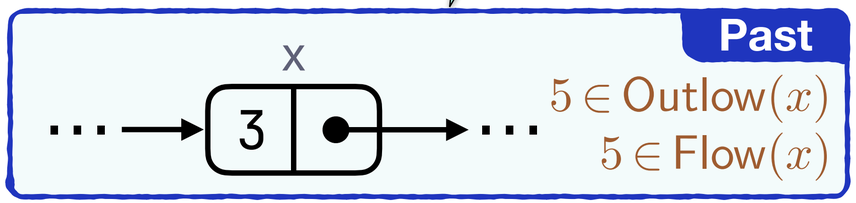
```
    x = y;
return x→key == 5;
```

Example: `contains(5)`

```
x = Root→next;
while (x→key < 5):
    y = x→next;
```



∧



Traversal *learns* facts about **Now**, not the **Past**.

Use **hindsight** to **propagate** **Now** facts into the **Past**.

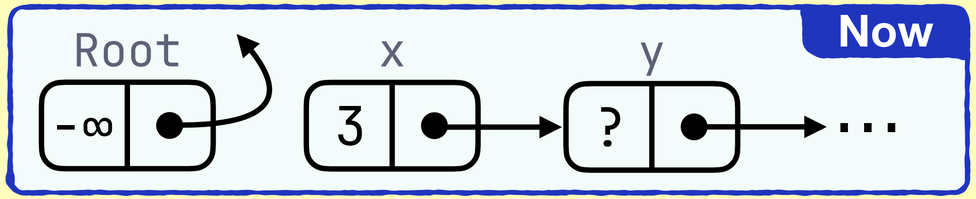
```
    x = y;
return x→key == 5;
```

Example: `contains(5)`

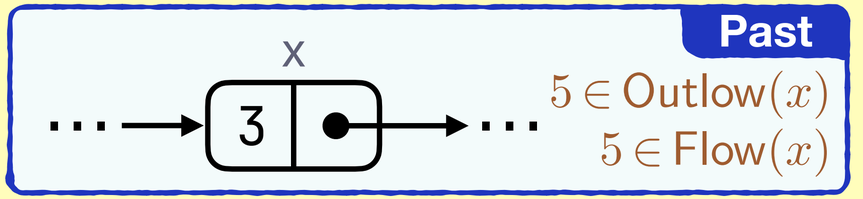
```
x = Root→next;
while (x→key < 5):
    y = x→next;
```

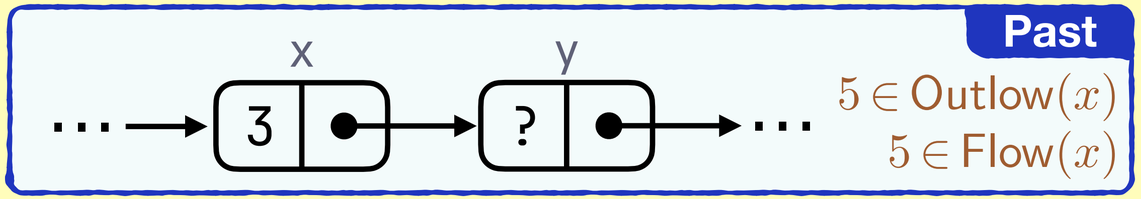
Traversal *learns* facts about **Now**, not the **Past**.

Use **hindsight** to **propagate** **Now** facts into the **Past**.



$\wedge$

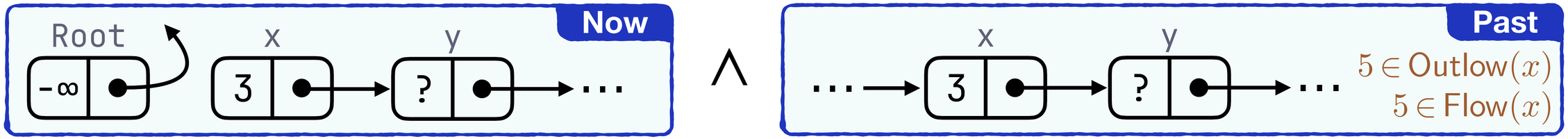




Temporal invariant:
unreachable nodes never change

Example: `contains(5)`

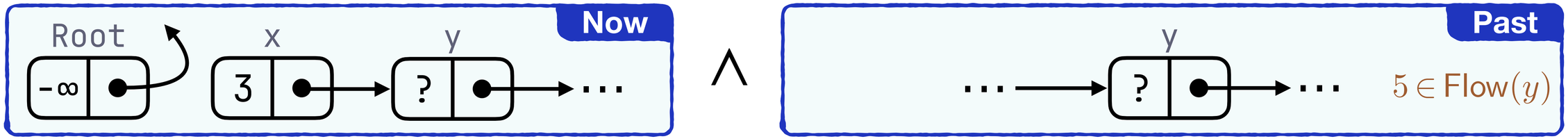
```
x = Root→next;
while (x→key < 5):
    y = x→next;
```




```
    x = y;
return x→key == 5;
```

Example: `contains(5)`

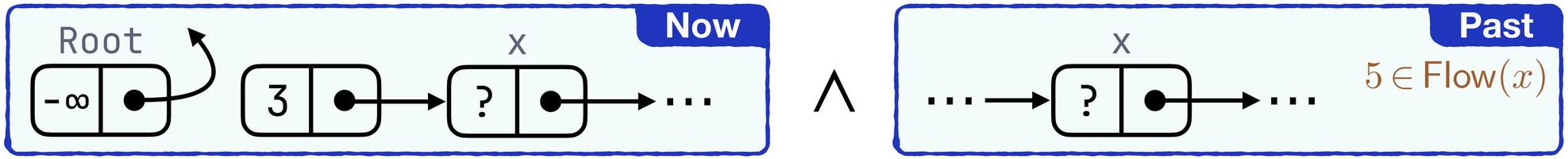
```
x = Root→next;
while (x→key < 5):
    y = x→next;
```




```
    x = y;
return x→key == 5;
```

Example: `contains(5)`

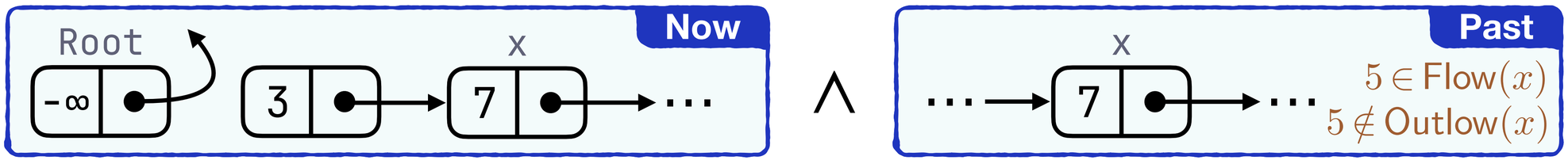
```
x = Root→next;
while (x→key < 5):
    y = x→next;
    x = y;
```




```
return x→key == 5;
```

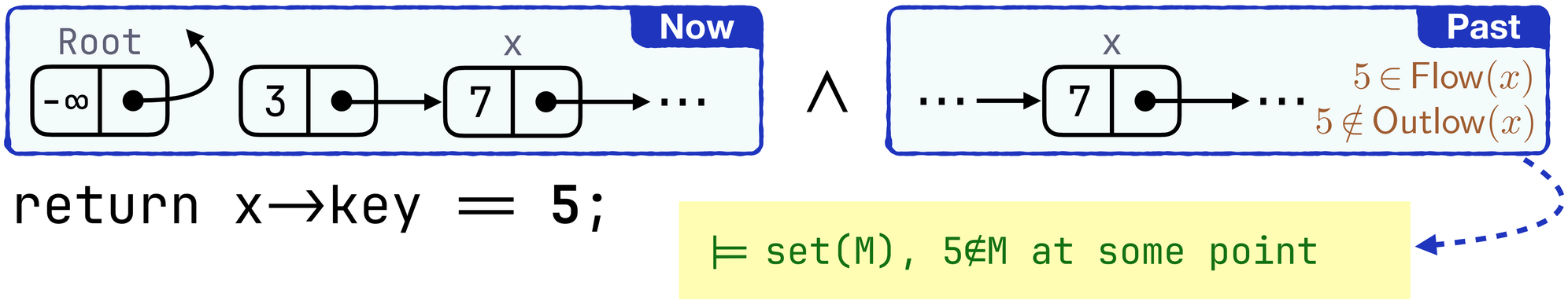
Example: `contains(5)`

```
x = Root→next;
while (x→key < 5):
    y = x→next;
    x = y;
return x→key == 5;
```



Example: `contains(5)`

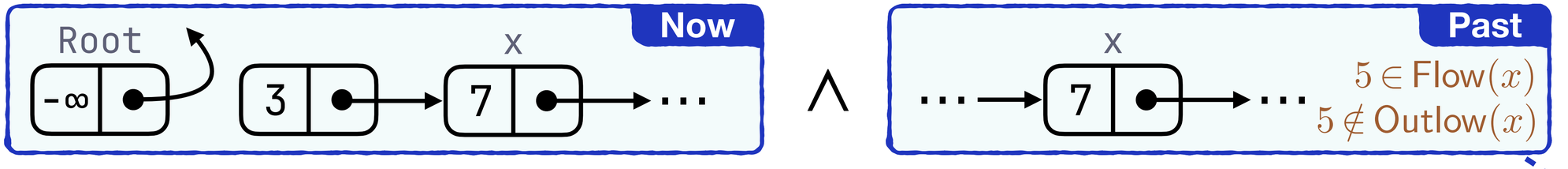
```
x = Root→next;
while (x→key < 5):
    y = x→next;
    x = y;
return x→key == 5;
```



⊨ set(M), 5∉M at some point

Example: `contains(5)`

```
x = Root→next;
while (x→key < 5):
    y = x→next;
    x = y;
return x→key == 5;
```



⊨ set(M), 5∉M at some point

establishes linearizability proof obligation

Technical Contribution

- **Temporal interpolation** proof rule

Technical Contribution

- **Temporal interpolation** proof rule

$$\frac{\mathsf{Interpolate}_{\mathbb{I}}(P,\, Q) \rightsquigarrow R}{\mathbb{I} \vdash \{S \wedge \mathsf{Past}(P) \wedge \mathsf{Now}(Q)\}\ \texttt{skip}\ \{S \wedge \mathsf{Past}(R)\}}$$

Technical Contribution

- **Temporal interpolation** proof rule

- Soundness

  ➡ **eliminate** frame rule

Technical Contribution

- **Temporal interpolation** proof rule

$$\frac{\mathsf{Interpolate}_{\mathbb{I}}(P,\, Q) \rightsquigarrow R}{\mathbb{I} \vdash \{S \wedge \mathsf{Past}(P) \wedge \mathsf{Now}(Q)\}\ \texttt{skip}\ \{S \wedge \mathsf{Past}(R)\}}$$

- Soundness

  ➡ **eliminate** frame rule

Technical Contribution

- **Temporal interpolation** proof rule

- Soundness
  - ➡ **eliminate** frame rule
  - ➡ **eliminate** temporal interpolation rule

Technical Contribution

- **Temporal interpolation** proof rule

$$\frac{\mathsf{Interpolate}_{\mathbb{I}}(P,\, Q) \rightsquigarrow R}{\mathbb{I} \vdash \{S \wedge \mathsf{Past}(P) \wedge \mathsf{Now}(Q)\}\ \texttt{skip}\ \{S \wedge \mathsf{Past}(R)\}}$$

- Soundness
  - ➡ **eliminate** frame rule
  - ➡ **eliminate** temporal interpolation rule

Technical Contribution

- **Temporal interpolation** proof rule

- Soundness
  - ➡ **eliminate** frame rule
  - ➡ **eliminate** temporal interpolation rule

Technical Contribution

- **Temporal interpolation** proof rule

$$\frac{\mathsf{Interpolate}_{\mathbb{I}}(P,\, Q) \rightsquigarrow R}{\mathbb{I} \vdash \{S \wedge \mathsf{Past}(P) \wedge \mathsf{Now}(Q)\}\ \texttt{skip}\ \{S \wedge \mathsf{Past}(R)\}}$$

- Soundness
  - ➡ **eliminate** frame rule
  - ➡ **eliminate** temporal interpolation rule

Technical Contribution

- **Temporal interpolation** proof rule

$$\frac{\mathsf{Interpolate}_{\mathbb{I}}(P,\, Q) \rightsquigarrow R}{\mathbb{I} \vdash \{S \wedge \mathsf{Past}(P) \wedge \mathsf{Now}(Q)\}\ \texttt{skip}\ \{S \wedge \mathsf{Past}(R)\}}$$

- Soundness
  - ➡ **eliminate** frame rule
  - ➡ **eliminate** temporal interpolation rule

Temporal interpolation structures proofs

Technical Contribution

- Interpolation requirements

$$\frac{\{\mathsf{Now}(P)\}\ \mathbb{I}^*\ \{\mathsf{Now}(Q) \Longrightarrow \mathsf{Past}(R)\}}{\mathsf{Interpolate}_{\mathbb{I}}(P,\, Q) \rightsquigarrow R}$$

Technical Contribution

- Interpolation requirements

$$\frac{\{\mathsf{Now}(P)\}\ \mathbb{I}^*\ \{\mathsf{Now}(Q) \implies \mathsf{Past}(R)\}}{\mathsf{Interpolate}_{\mathbb{I}}(P,\, Q) \rightsquigarrow R}$$

- Interpolation strategy

$$\frac{\exists I.\quad \mathsf{Now}(P) \implies I \quad I \text{ interference-free} \quad I \wedge \mathsf{Now}(Q) \implies \mathsf{Past}(R)}{\mathsf{Interpolate}_{\mathbb{I}}(P,\, Q) \rightsquigarrow R}$$

Technical Contribution

- Interpolation requirements

$$\frac{\{\mathsf{Now}(P)\}\ \mathbb{I}^*\ \{\mathsf{Now}(Q) \Longrightarrow \mathsf{Past}(R)\}}{\mathsf{Interpolate}_{\mathbb{I}}(P,\, Q) \rightsquigarrow R}$$

- Interpolation strategy

$$\frac{\exists I.\quad \mathsf{Now}(P) \Longrightarrow I \quad I \text{ interference-free} \quad I \wedge \mathsf{Now}(Q) \Longrightarrow \mathsf{Past}(R)}{\mathsf{Interpolate}_{\mathbb{I}}(P,\, Q) \rightsquigarrow R}$$

- Great choice: $R = P \wedge Q$

Technical Contribution

- Interpolation requirements

$$\frac{\{\mathsf{Now}(P)\}\ \mathbb{I}^*\ \{\mathsf{Now}(Q) \implies \mathsf{P}}{\mathsf{Interpolate}_{\mathbb{I}}(P, Q) \rightsquigarrow I}$$

- Interpolation strategy

$$\frac{\exists I.\quad \mathsf{Now}(P) \implies I \quad I \text{ interference-free} \quad I}{\mathsf{Interpolate}_{\mathbb{I}}(P, Q) \rightsquigarrow I}$$

- Great choice: $R = P \wedge Q$

[PLDI'23]

Embedding Hindsight Reasoning in Separation Logic

ROLAND MEYER, TU Braunschweig, Germany
THOMAS WIES, New York University, USA
SEBASTIAN WOLFF, New York University, USA

Automatically proving linearizability of concurrent data structures remains a key challenge for verification. We present temporal interpolation as a new proof principle to guide automated proof search using hindsight arguments within concurrent separation logic. Temporal interpolation offers an easy-to-automate alternative to prophecy variables and has the advantage of structuring proofs into easy-to-discharge hypotheses. Additionally, we advance hindsight theory by integrating it into a program logic, bringing formal rigor and complementary proof machinery. We substantiate the usefulness of temporal interpolation by implementing it in a tool and using it to automatically verify the Logical Ordering tree. The proof is challenging due to future-dependent linearization points and complex structure overlays. It is the first formal proof of this data structure. Interestingly, our formalization revealed an unknown bug and an existing informal proof as erroneous.

Conference Version:
Roland Meyer, Thomas Wies, and Sebastian Wolff. 2023. Embedding Hindsight Reasoning in Separation Logic. *Proc. ACM Program. Lang.* 7, PLDI, Article 182 (June 2023), 24 pages. https://doi.org/10.1145/3591296

1 INTRODUCTION

We are concerned with automatically proving linearizability, the standard correctness criterion for concurrent data structures [Herlihy and Wing 1990]. A concurrent data structure is linearizable subject to a sequential specification of its methods, if each method takes effect in a single atomic step of its concurrent execution, the method's *linearization point*, and satisfies the sequential specification in this step.

Concurrent separation logics [Bell et al. 2010; Delbianco et al. 2017; Elmas et al. 2010; Fu et al. 2010; Gotsman et al. 2013; Gu et al. 2018; Hemed et al. 2015; Parkinson et al. 2007; Sergey et al. 2015; Vafeiadis and Parkinson 2007] provide a powerful toolbox of deductive reasoning techniques to verify complex concurrent data structures. However, the proof construction heavily relies on the proof author's creativity and expertise in wielding the available tools effectively. For instance, in order to construct the inductive invariant of the data structure, the proof author may have to devise proof-specific resource algebras to express ghost state that captures the key aspects of the computation history. This hinders proof automation due to the vast complexity of the proof space that needs to be explored. Similarly, the proofs may make use of prophecy variables [Abadi and Lamport 1991] to predict future-dependent linearization points [Jung et al. 2020; Liang and Feng 2013; Vafeiadis 2008]. Constructing such proofs involves backward reasoning, which is difficult to automate [Bouajjani et al. 2017]. It stands to reason that there is a need for guiding principles that help to structure the proof and that provide effective strategies for automated tools to prune the search space.

Hindsight theory [Feldman et al. 2018, 2020; Lev-Ari et al. 2015; O'Hearn et al. 2010] provides such a guiding principle, which we refer to as *temporal interpolation*. One proves lemmas of the form: if there existed a past state that satisfied property p and the current state satisfies q, then there must have existed an intermediate state that satisfied o. Such lemmas can then be applied, e.g., to prove the

Implementations

- `plankton`

  - linearizability checker for concurrent search structures
  - input: edge functions + node invariant
  - hard-coded: search path flow monoid
  - automatically constructs proofs

Implementations

- `plankton` 

  - linearizability checker for concurrent search structures
  - input: edge functions + node invariant
  - hard-coded: search path flow monoid
  - automatically constructs proofs
- `nekton` 

  - additional input: custom flow monoid + proof outline
  - checks proof outline for validity

Implementations

- `plankton`

  - linearizability checker for concurrent search
  - input: edge functions + node invariant
  - hard-coded: search path flow monoid
  - automatically constructs proofs

- `nekton`

  - additional input: custom flow monoid + proc
  - checks proof outline for validity

[PLDI'23]

Embedding Hindsight Reasoning in Separation Logic

ROLAND MEYER, TU Braunschweig, Germany
THOMAS WIES, New York University, USA
SEBASTIAN WOLFF, New York University, USA

Automatically proving linearizability of concurrent data structures remains a key challenge for verification.
We present
arguments
tive to proj
Additionall
complemen
in a tool an
dependent
Interesting

Conferenc
Roland Mey
Proc. ACM

1 INTR
We are co
concurren
subject to
step of its
specificati
Concur
2010; Got
2015; Vafe
to verify c
the proof
in order t
devise pro
computati
that needs
Lamport 1
2013; Vafe
automate
help to str
search spa
Hindsig
such a gui
if there ex
have existe

[CAV'23]

nekton: a linearizability proof checker

Roland Meyer[1], Anton Opaterny[1(✉)], Thomas Wies[2], and Sebastian Wolff[2]

[1] TU Braunschweig, Braunschweig, Germany
{anton.opaterny,roland.meyer}@tu-bs.de
[2] New York University, New York, USA
{wies,sebastian.wolff}@cs.nyu.edu

Abstract nekton is a new tool for checking linearizability proofs of highly complex concurrent search structures. The tool's unique features are its parametric heap abstraction based on separation logic and the flow framework, and its support for hindsight arguments about future-dependent linearization points. We describe the tool, present a case study, and discuss implementation details.

Keywords: separation logic · proof checker · linearizability · flow framework

1 Introduction

We present nekton, a mostly automated deductive program verifier based on separation logic (SL) [23,27]. The tool is designed to aid the construction of linearizability proofs for complex concurrent search structures. Similar to many other SL-based tools [2,8,14,22,33,33], nekton uses an SMT solver to automate basic SL reasoning. Similar to the original implementation of CIVL [7], it uses non-interference reasoning à la Owicki-Gries [25] to automate thread modularity. What makes nekton stand out among these relatives is its inbuilt support for expressing complex inductive heap invariants using the flow framework [12,13,20] and the ability to (partially) automate complex linearizability arguments that require hindsight reasoning [4,5,15,18,19,24]. Together, these features enable nekton to verify challenging concurrent data structures such as the FEMRS tree [4] with little user guidance.

nekton [17] is derived from the tool plankton [18,19], which shares the same overall goals and features as nekton but strives for full proof automation at the expense of generality. In terms of the trade-off between automation and expressivity, nekton aims to occupy a sweet spot between plankton and general purpose program verifiers. In the following, we discuss nekton's unique features in more detail and explain how it deviates from plankton's design.

The flow framework can be used to express global properties of graph structures in a node-local manner, aiding compositional verification of recursive data structures. The framework is parametric in a *flow domain* which determines what global information about the graph is provided at each node. Various flow domains have been proposed that have shown to be useful in concurrency proofs [11,26]. To simplify proof automation, plankton uses a fixed flow domain that is geared towards verifying functional correctness of search structures. In contrast, nekton is parametric in the flow domain. For instance, it supports custom domains for reasoning about overlayed structures and

plankton

- Verifies future-dependent linearization points
- **Logical Ordering Tree**
  - found bug in implementation
  - found bug in previous proof
  - first proof of (fixed) version

Benchmark	Verification Time
Fine-Grained set	45*s* ✓
Lazy set	2*m* 13*s* ✓
FEMRS tree (no maintenance)	3*m* 50*s* ✓
Vechev&Yahav 2CAS set	1*m* 15*s* ✓
Vechev&Yahav CAS set	2*m* 20*s* ✓
ORVYY set	1*m* 36*s* ✓
Michael set	6*m* 53*s* ✓
Michael set (wait-free search)	6*m* 53*s* ✓
Harris set	57*m* 20*s* ✓
Harris set (wait-free search)	43*m* 00*s* ✓
LO-tree (generalized maintenance)	16*m* 43*s* ✓

Overview

Flow Framework

Data-flow-like ghost state for inductive heap invariants.

[POPL'18, ESOP'20, TACAS'23]

Hindsight

Non-fixed linearization points without prophecies.

[OOPSLA'22, PLDI'23]

Decomposing Updates

Unbounded footprints without induction.

[OOPSLA'22, *under submission*]

Goal

Localize the proof to a small number of nodes.

Problem

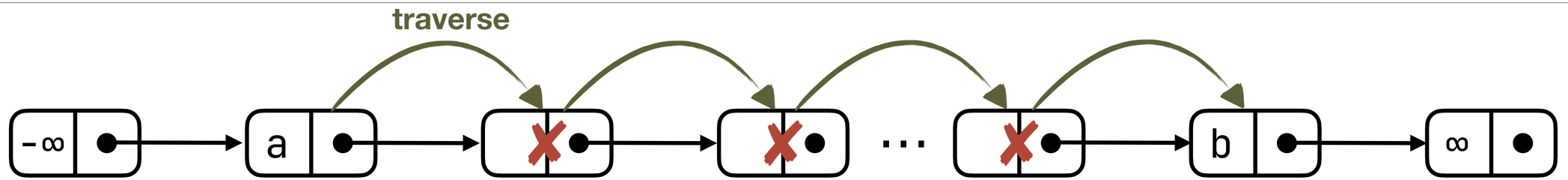
Complex updates affect an unbounded number of nodes.

Solution

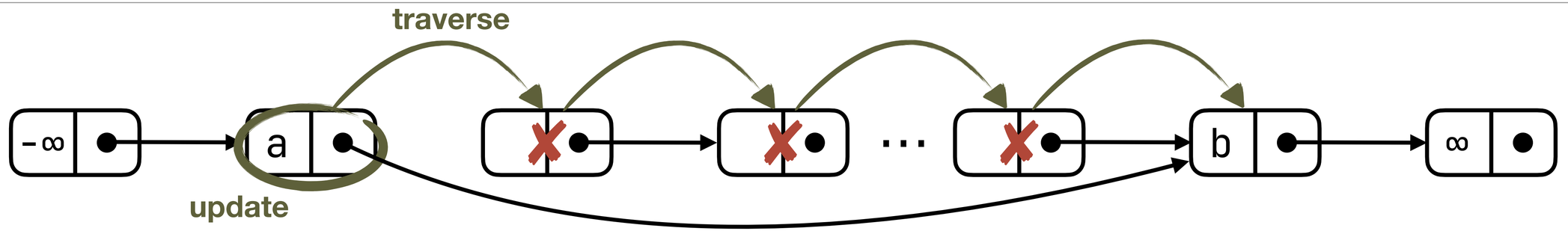
Keep track of ghost updates that are local.

Complex Updates

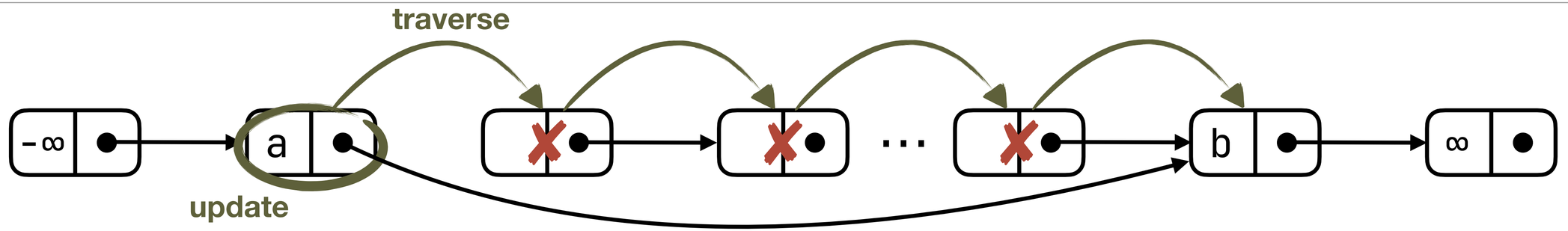
Complex Updates



Complex Updates



Complex Updates



State of the art: **recursive predicates + induction**

hard to automate

➡ custom predicate captures to-be-updated **region**

➡ **induction** over predicate upon actual update

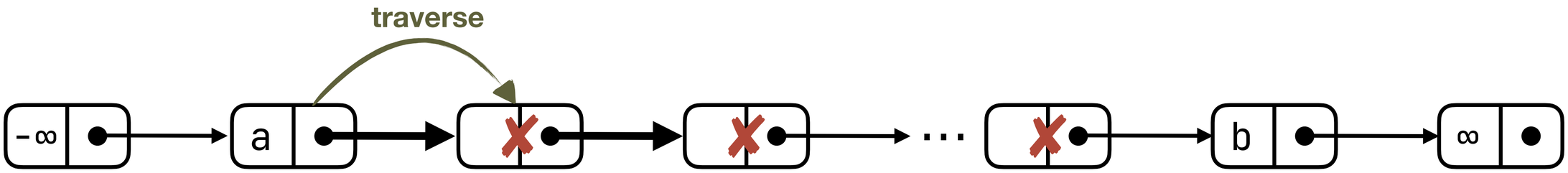
Lesson in Life

Computation is **local** and **continuous**

Complex Updates

Gost update chunks

- ➡ compose complex update from smaller chunks
- ➡ the chunks are hypothetical/ghost information for the proof
- ➡ can prove the composition **along the traversal**, no induction required

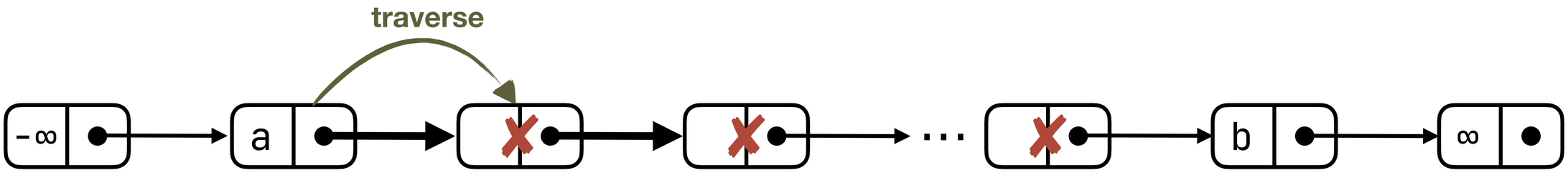


Complex Updates

Gost update chunks

- ➡ compose complex update from smaller chunks
- ➡ the chunks are hypothetical/ghost information for the proof
- ➡ can prove the composition **along the traversal**, no induction required

easier to automate

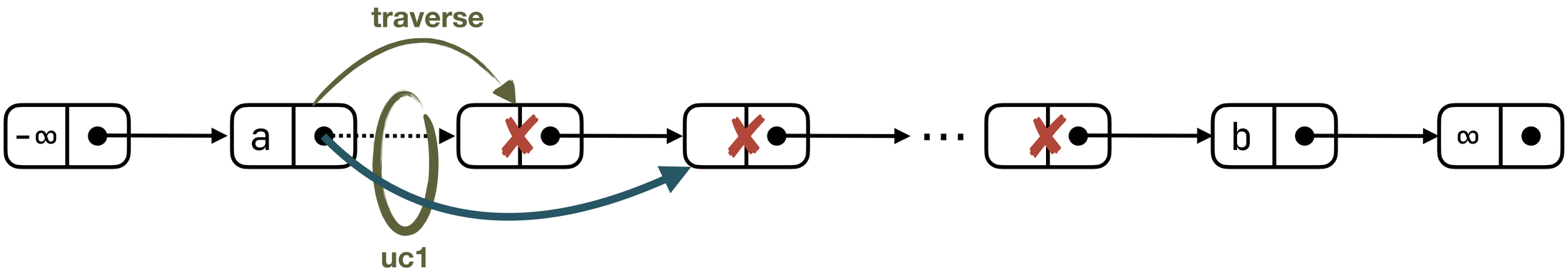


Complex Updates

Gost update chunks

- ➡ compose complex update from smaller chunks
- ➡ the chunks are hypothetical/ghost information for the proof
- ➡ can prove the composition **along the traversal**, no induction required

easier to automate

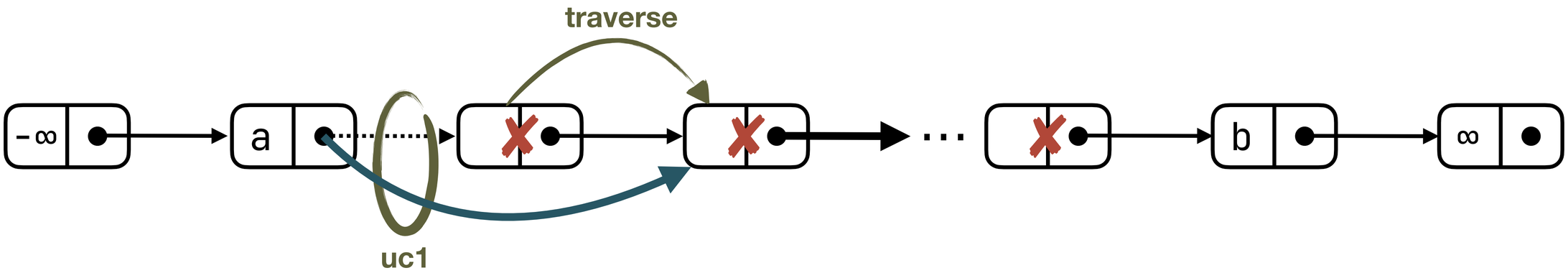


Complex Updates

Gost update chunks

- ➡ compose complex update from smaller chunks
- ➡ the chunks are hypothetical/ghost information for the proof
- ➡ can prove the composition **along the traversal**, no induction required

easier to automate

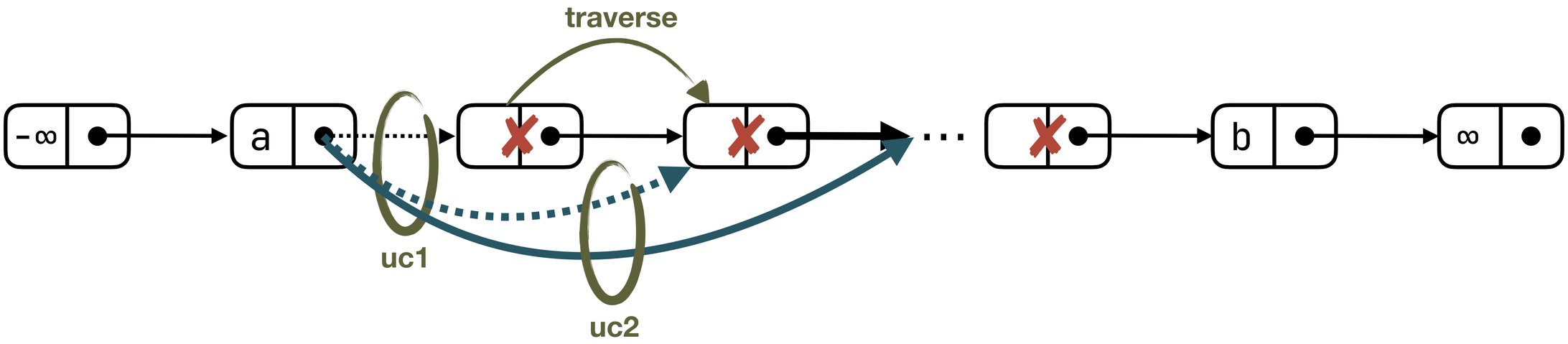


Complex Updates

Gost update chunks

- ➡ compose complex update from smaller chunks
- ➡ the chunks are hypothetical/ghost information for the proof
- ➡ can prove the composition **along the traversal**, no induction required

easier to automate

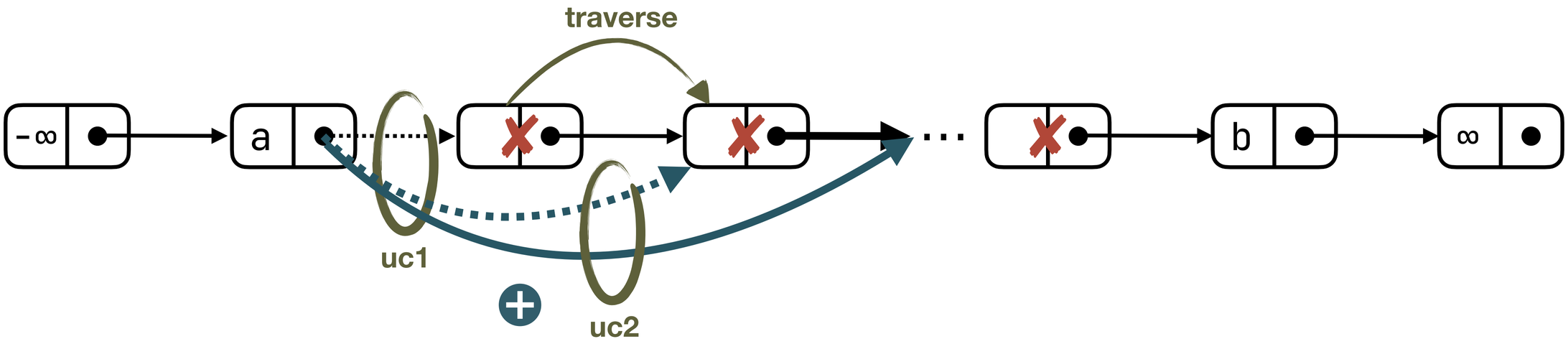


Complex Updates

Gost update chunks

- ➡ compose complex update from smaller chunks
- ➡ the chunks are hypothetical/ghost information for the proof
- ➡ can prove the composition **along the traversal**, no induction required

easier to automate

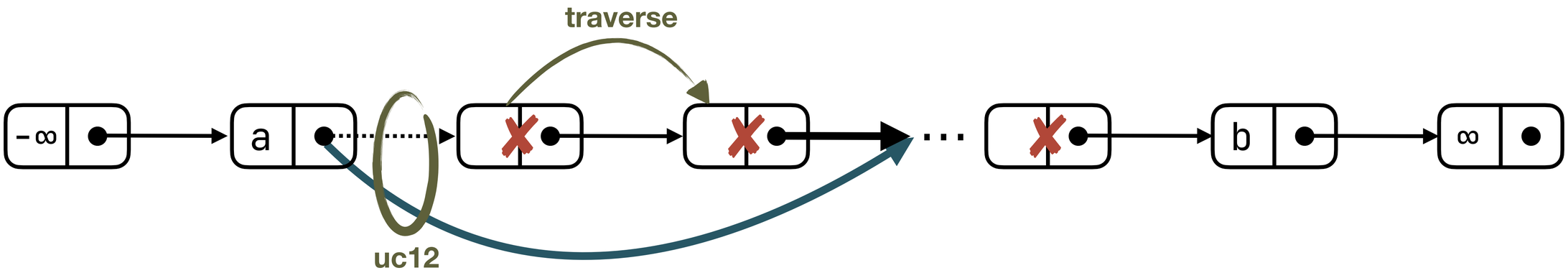


Complex Updates

Gost update chunks

- ➡ compose complex update from smaller chunks
- ➡ the chunks are hypothetical/ghost information for the proof
- ➡ can prove the composition **along the traversal**, no induction required

easier to automate

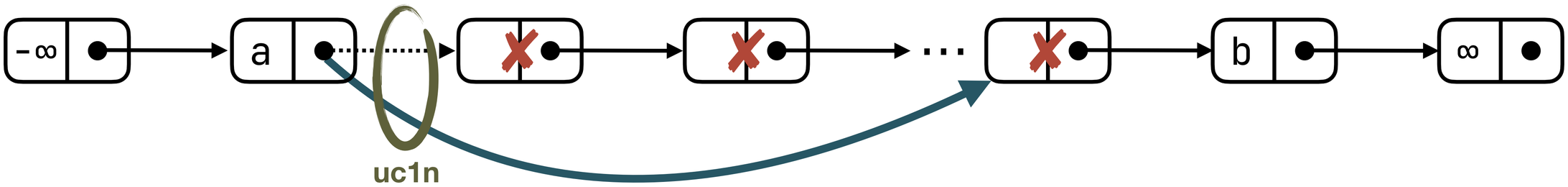


Complex Updates

Gost update chunks

- ➡ compose complex update from smaller chunks
- ➡ the chunks are hypothetical/ghost information for the proof
- ➡ can prove the composition **along the traversal**, no induction required

easier to automate

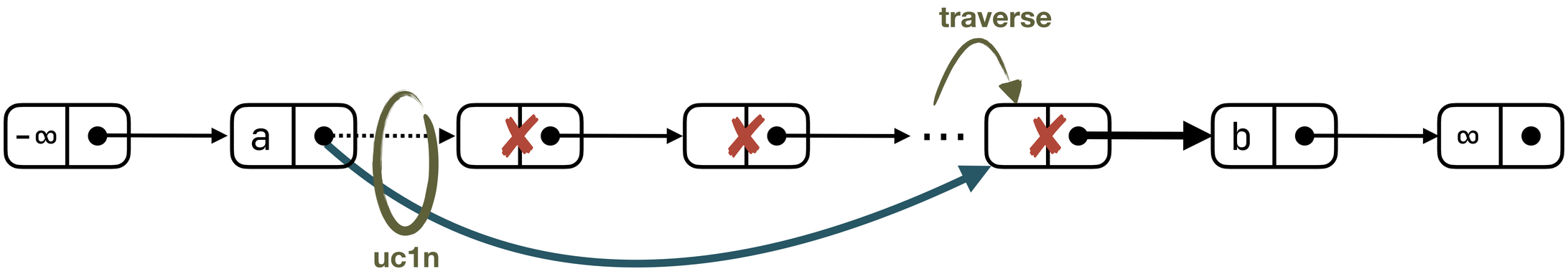


Complex Updates

Gost update chunks

- ➡ compose complex update from smaller chunks
- ➡ the chunks are hypothetical/ghost information for the proof
- ➡ can prove the composition **along the traversal**, no induction required

easier to automate

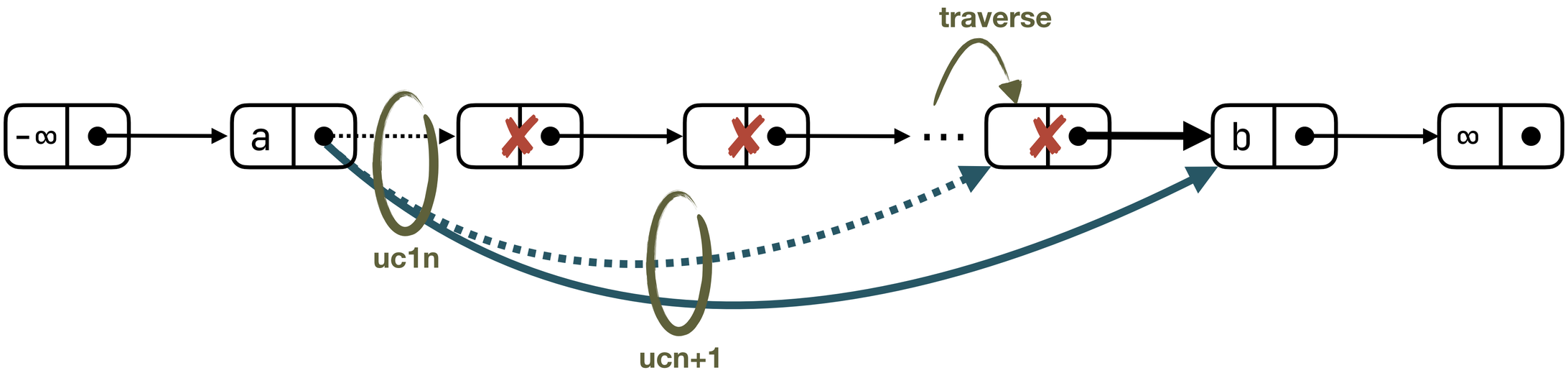


Complex Updates

Gost update chunks

- ➡ compose complex update from smaller chunks
- ➡ the chunks are hypothetical/ghost information for the proof
- ➡ can prove the composition **along the traversal**, no induction required

easier to automate

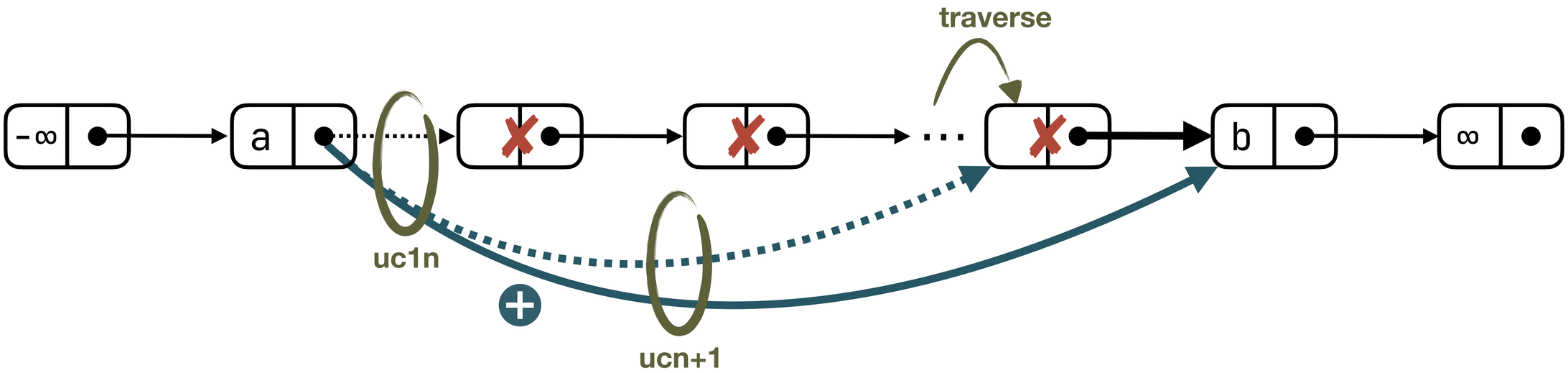


Complex Updates

Gost update chunks

- ➡ compose complex update from smaller chunks
- ➡ the chunks are hypothetical/ghost information for the proof
- ➡ can prove the composition **along the traversal**, no induction required

easier to automate

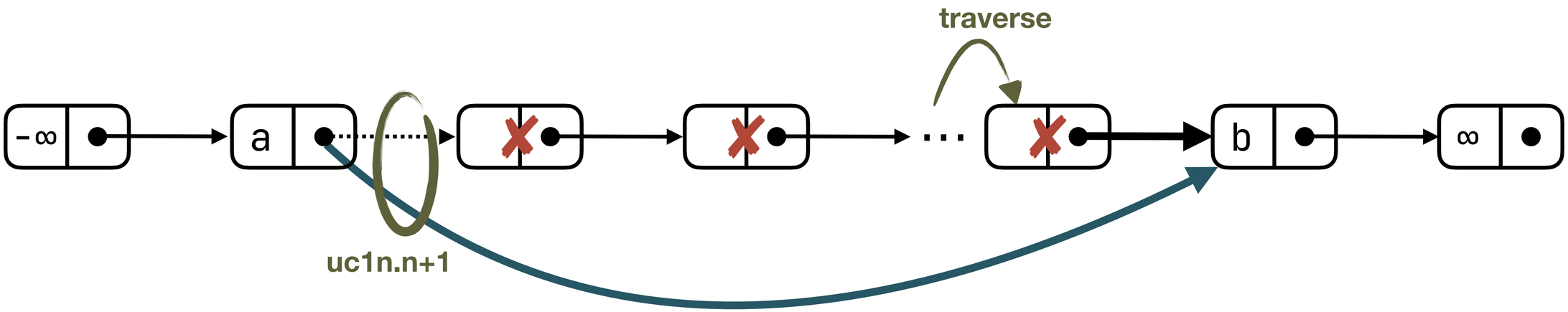


Complex Updates

Gost update chunks

- ➡ compose complex update from smaller chunks
- ➡ the chunks are hypothetical/ghost information for the proof
- ➡ can prove the composition **along the traversal**, no induction required

easier to automate



Complex Updates

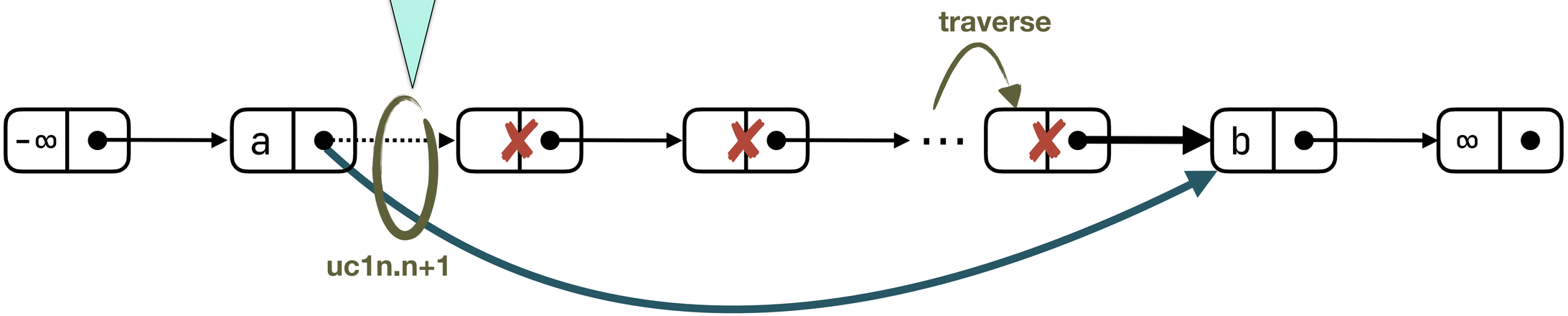
Gost update chunks

easier to automate

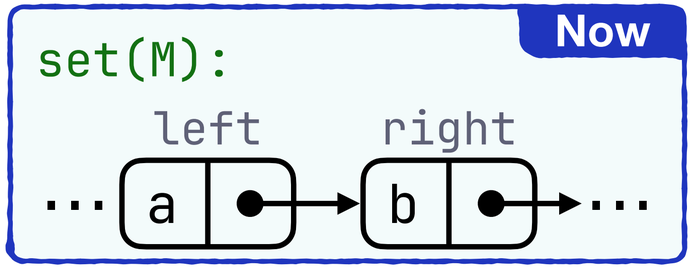
The update of interest,
constructed from **local** (three nodes) ghost update chunks
as the **limit of a computation**.

e proof

induction required



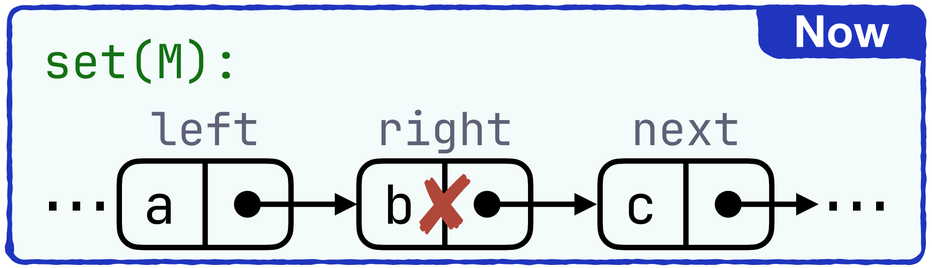
Example: removing marked nodes




```
while (right→mark):
    next = right→next;
    right = next;
left→next = right;
```

Example: removing marked nodes

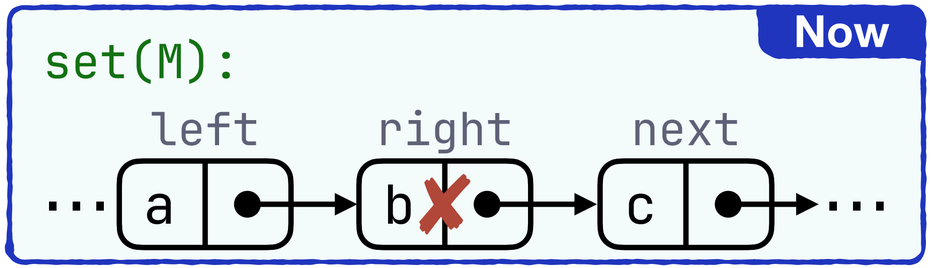
```
while (right→mark):
    next = right→next;
```

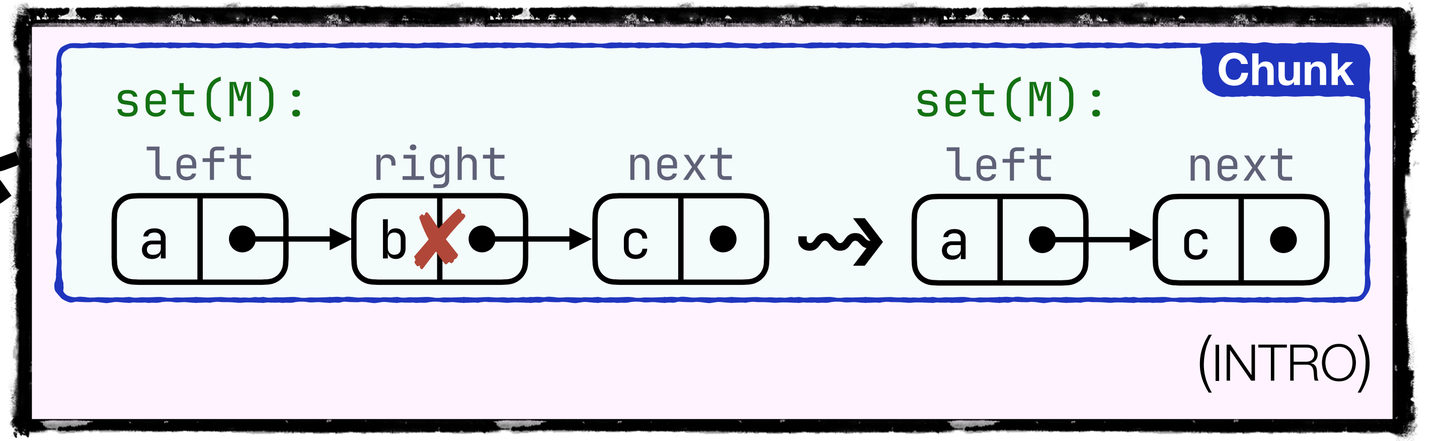



```
    right = next;
left→next = right;
```

Example: removing marked nodes

```
while (right→mark):
    next = right→next;
```

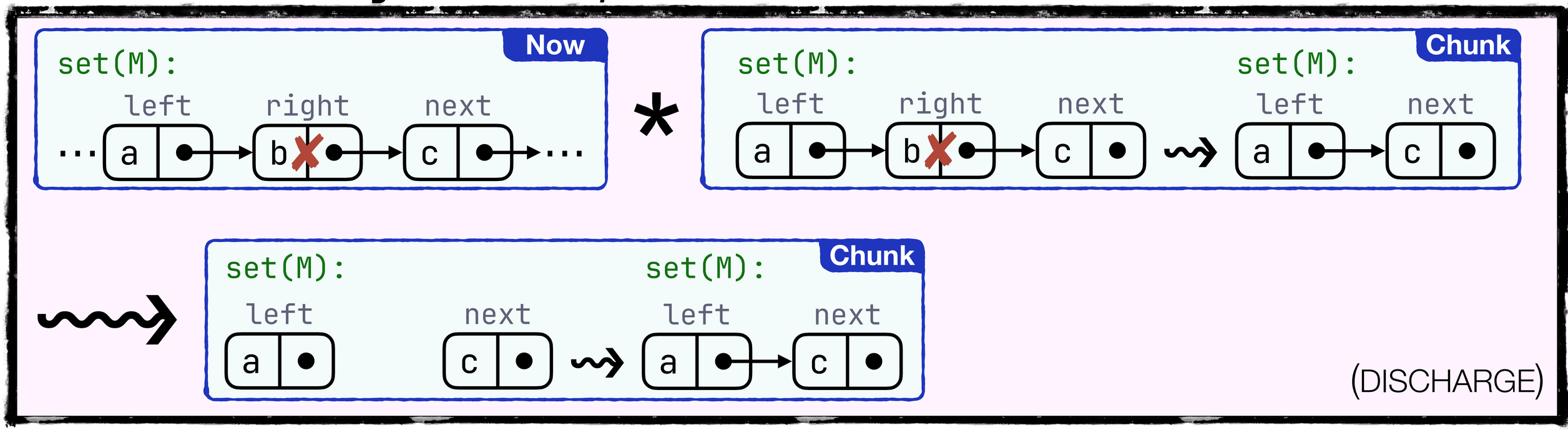





```
    right = next;

left→next = right;
```

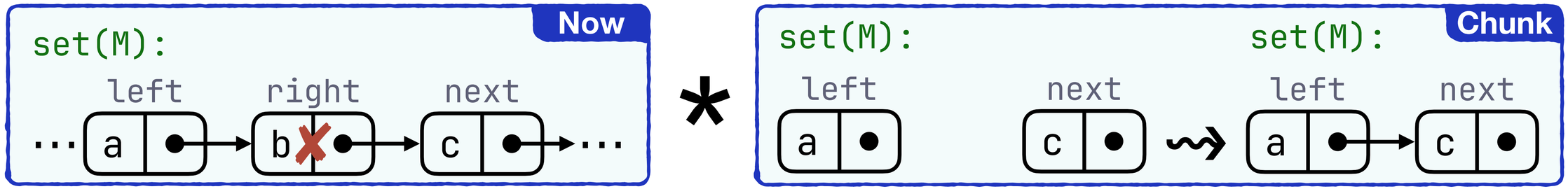
Example: removing marked nodes

```
while (right→mark):

    next = right→next;
```



Example: removing marked nodes

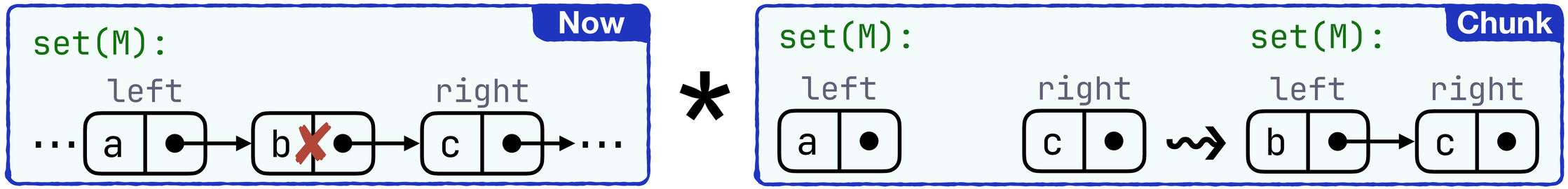
```
while (right→mark):
    next = right→next;
```




```
    right = next;

left→next = right;
```

Example: removing marked nodes

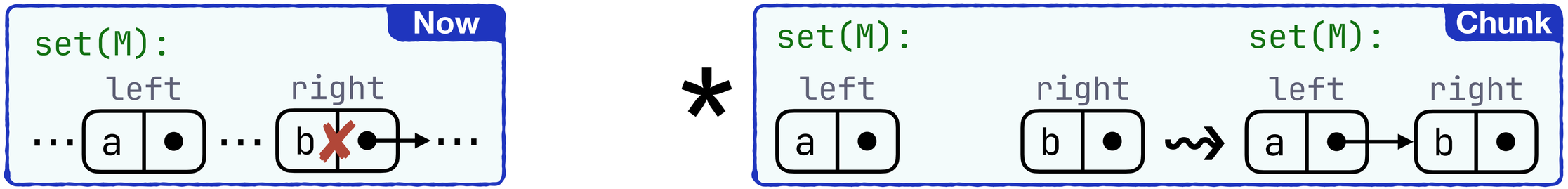
```
while (right→mark):
    next = right→next;
    right = next;
```




```
left→next = right;
```

Example: removing marked nodes

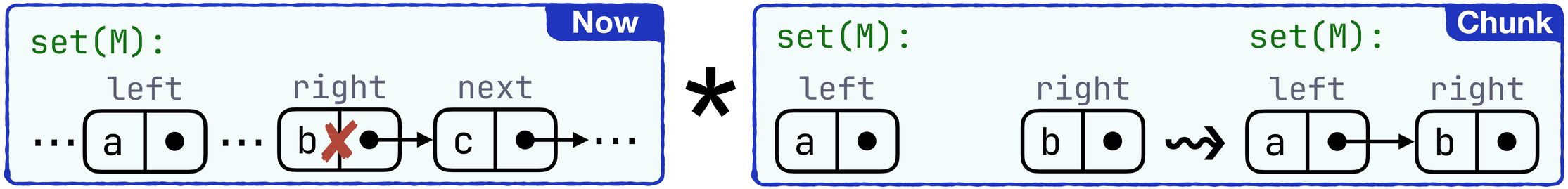
```
while (right→mark):
```




```
    next = right→next;

    right = next;

left→next = right;
```

Example: removing marked nodes

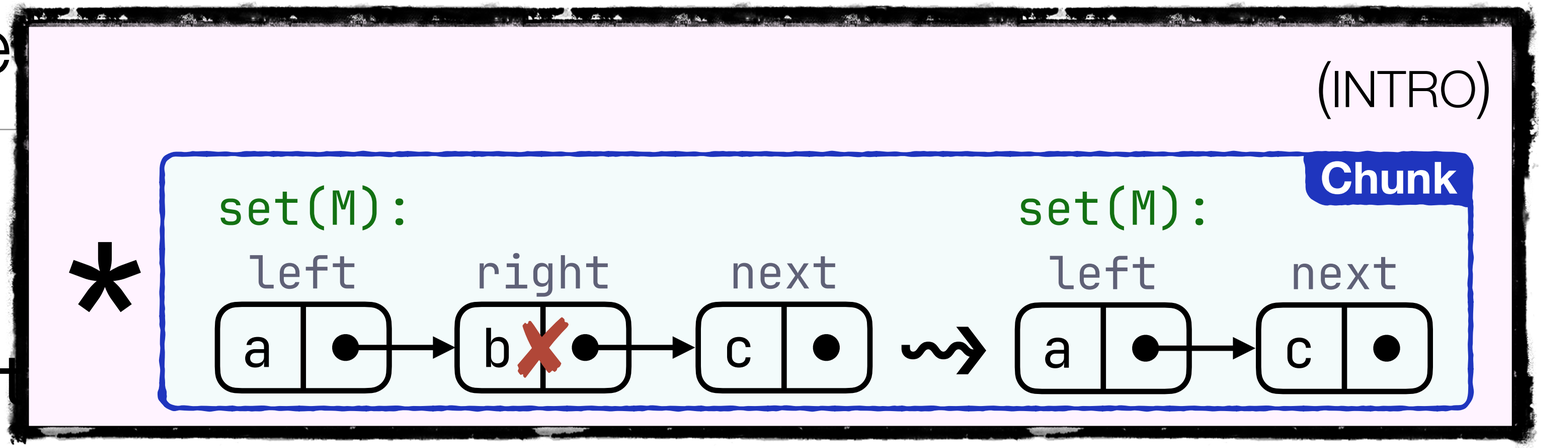
```
while (right→mark):
    next = right→next;
```

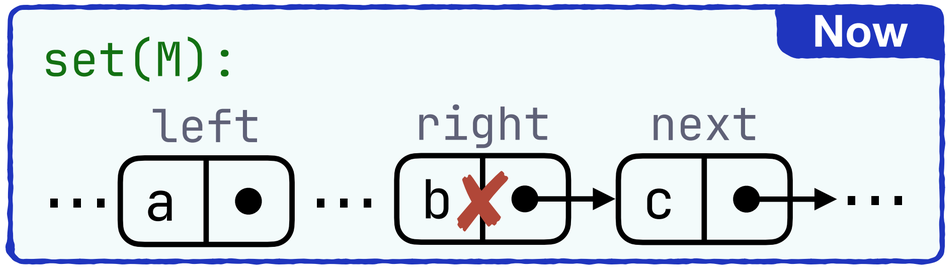


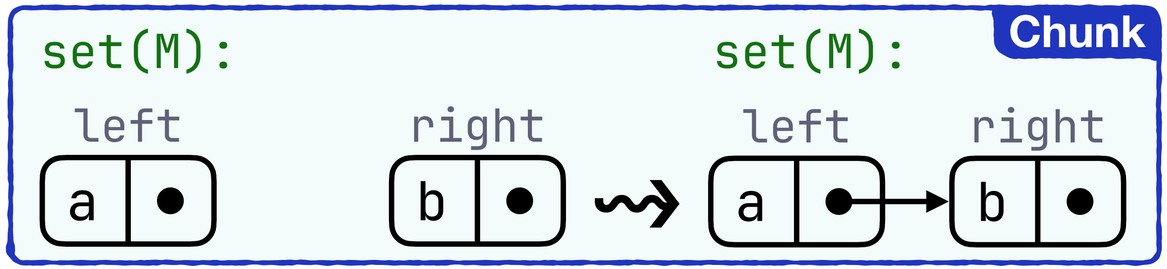

```
    right = next;

left→next = right;
```

Example: removing marke

```
while (right→mark):
    next = right→next
```



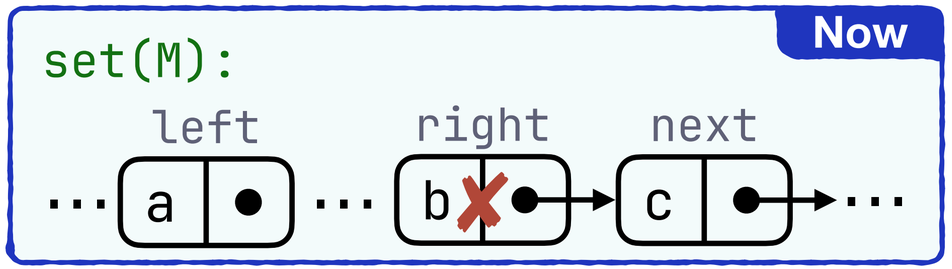




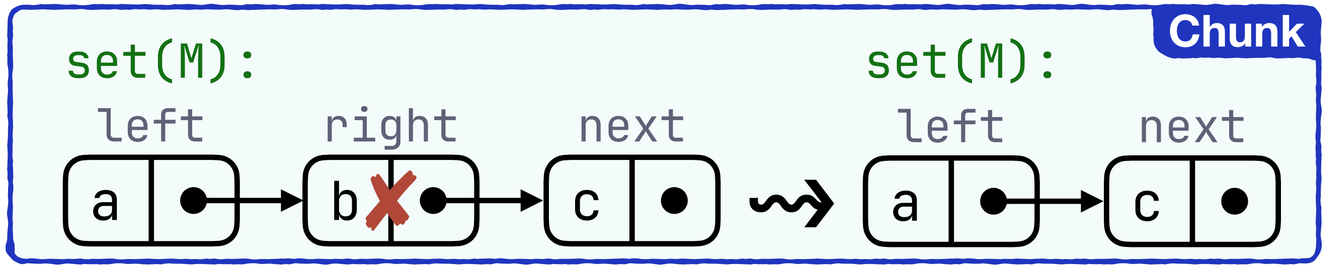

```
    right = next;

left→next = right;
```

Example: removing marked nodes

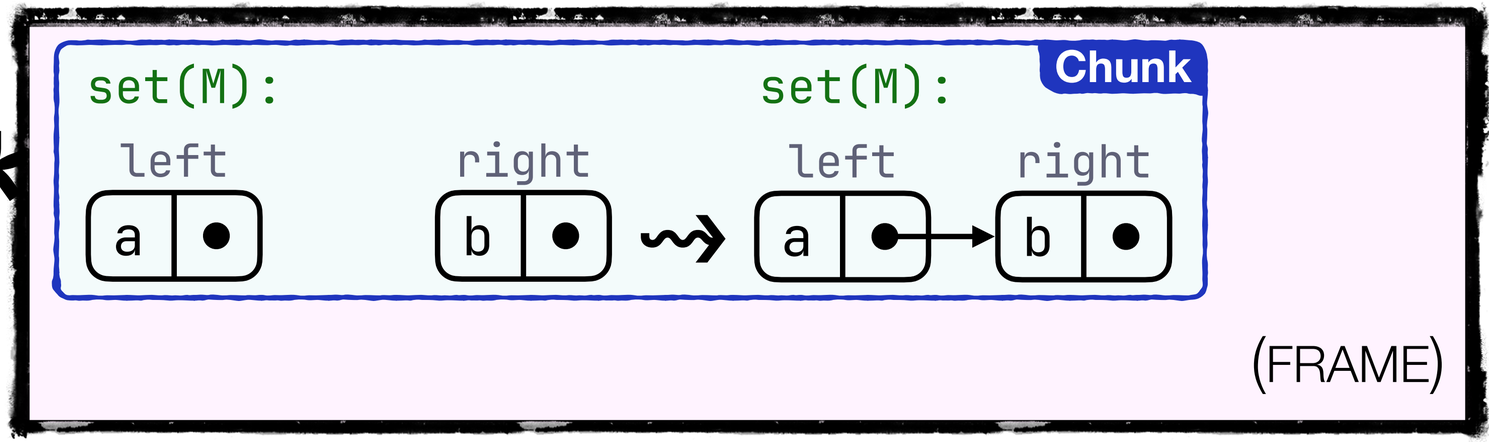
```
while (right→mark):
    next = right→next;
    right = next;
left→next = right;
```



*

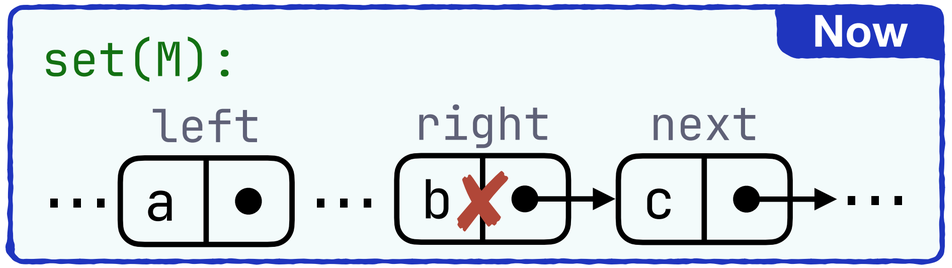


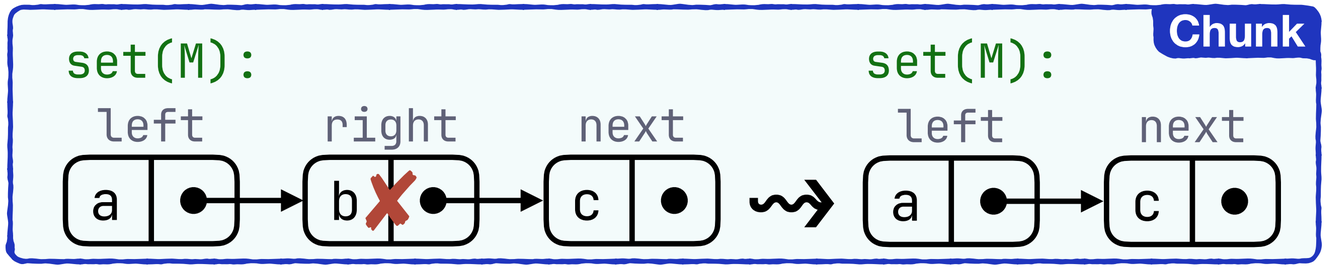
*

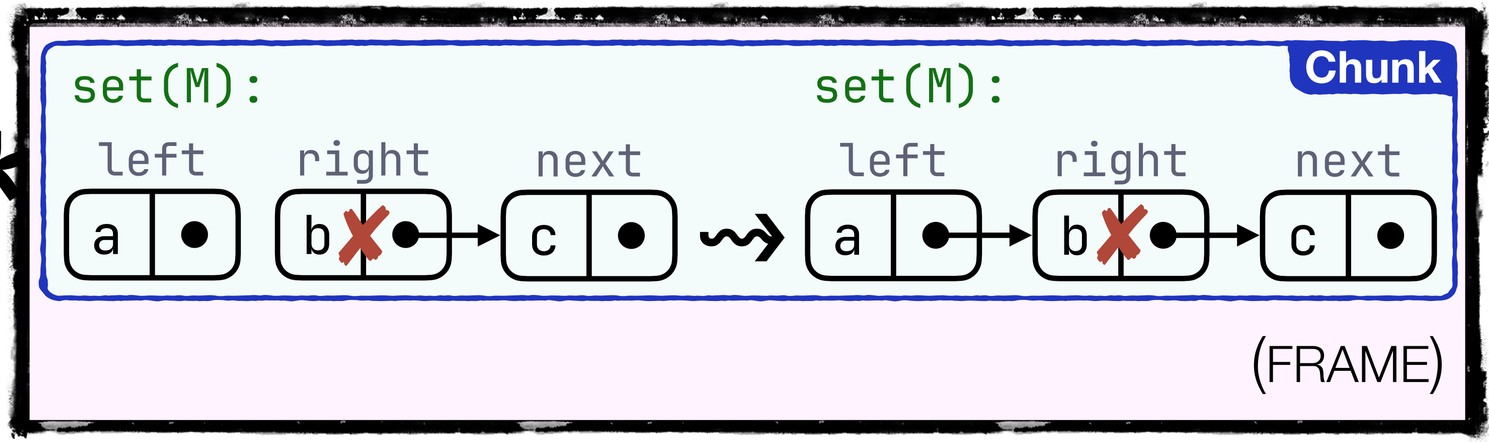


Example: removing marked nodes

```
while (right→mark):
    next = right→next;
    right = next;
left→next = right;
```

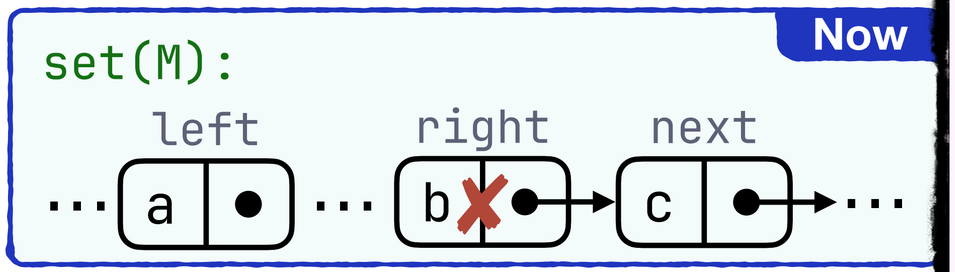




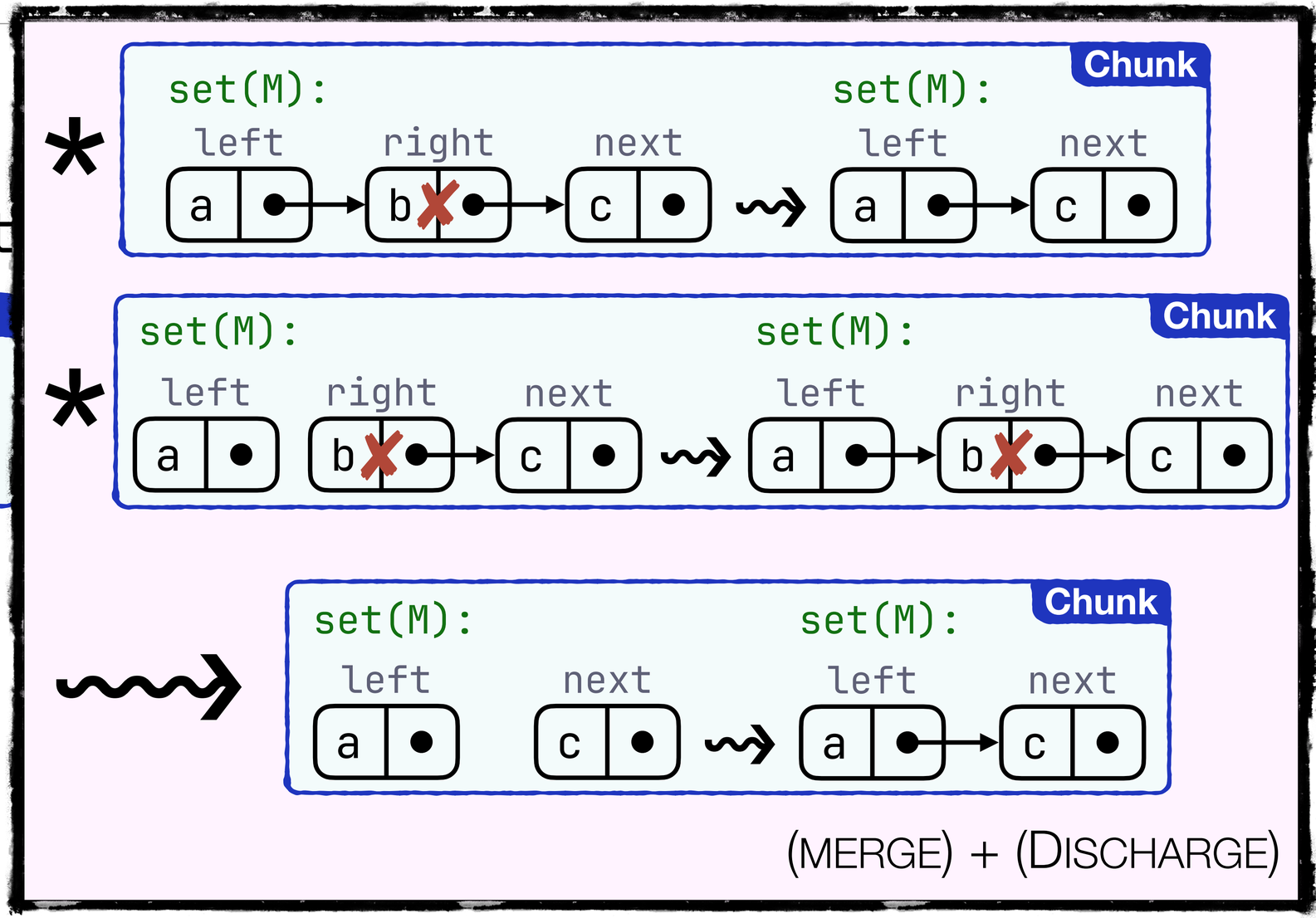


Example: removing marked nodes

```
while (right→mark):
    next = right→next
```

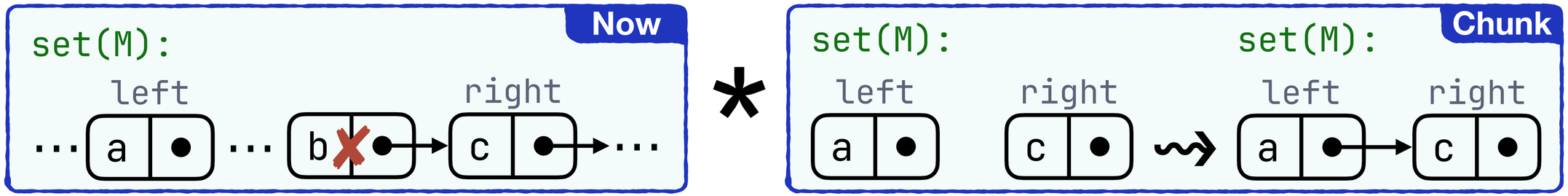



```
    right = next;
left→next = right;
```



Example: removing marked nodes

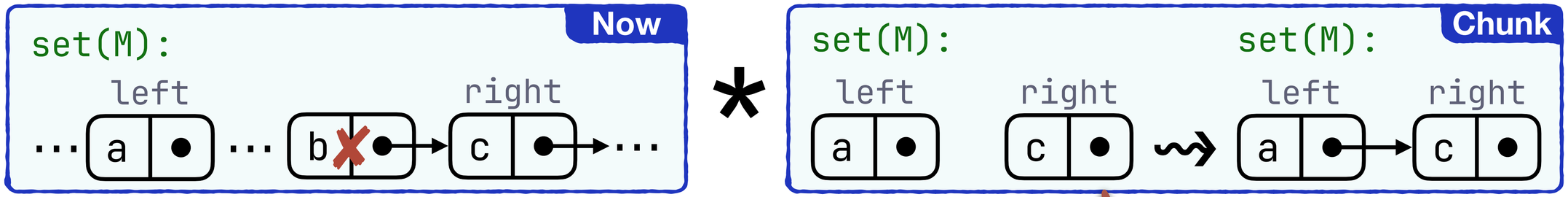
```
while (right→mark):
    next = right→next;
    right = next;
```



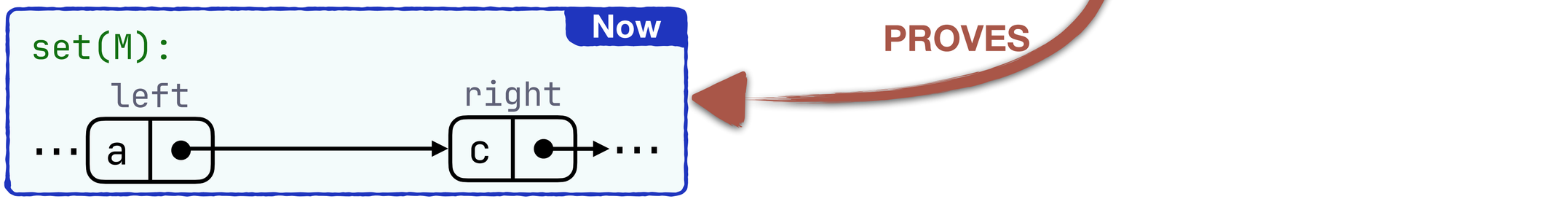

```
left→next = right;
```

Example: removing marked nodes

```
while (right→mark):
    next = right→next;
    right = next;
```




```
left→next = right;
```



Technical Contribution

- Abstract Owicki-Gries separation logic, with
  - update chunks: $\langle P \rangle \ \texttt{com} \ \langle Q \rangle$

Technical Contribution

- Abstract Owicki-Gries separation logic, with
  - update chunks: $\langle P \rangle\ \mathtt{com}\ \langle Q \rangle$
  - proof rules (excerpt):

$$\frac{}{R * \langle P * R \rangle\ \mathtt{com}\ \langle Q \rangle \implies \langle P \rangle\ \mathtt{com}\ \langle Q \rangle}\ \text{(DISCHARGE)}$$

$$\frac{\mathtt{com}_1;\mathtt{com}_2 \approx \mathtt{com}}{\langle P \rangle\ \mathtt{com}\ \langle Q \rangle * \langle Q \rangle\ \mathtt{com}\ \langle R \rangle \implies \langle P \rangle\ \mathtt{com}\ \langle R \rangle}\ \text{(MERGE)}$$

Technical Contribution

- Abstract Owicki-Gries separation logic, with
  - update chunks: $\langle P \rangle\ \mathtt{com}\ \langle Q \rangle$
  - proof rules (excerpt):

$$\frac{}{R * \langle P * R \rangle\ \mathtt{com}\ \langle Q \rangle \implies \langle P \rangle\ \mathtt{com}\ \langle Q \rangle} \text{(DISCHARGE)}$$

$$\frac{\mathtt{com}_1;\mathtt{com}_2 \approx \mathtt{com}}{\langle P \rangle\ \mathtt{com}\ \langle Q \rangle * \langle Q \rangle\ \mathtt{com}\ \langle R \rangle \implies \langle P \rangle\ \mathtt{com}\ \langle R \rangle} \text{(MERGE)}$$

- Soundness by elimination

Technical Contribution

- Abstract Owicki-Gries separation logic, with
  - update chunks: $\langle P \rangle \; \mathtt{com} \; \langle Q \rangle$
  - proof rules (excerpt):

$$\frac{}{R * \langle P * R \rangle \; \mathtt{com} \; \langle Q \rangle \implies}$$

$$\frac{\mathtt{com}_1;\mathtt{com}_2 \approx \mathtt{co}}{\langle P \rangle \; \mathtt{com} \; \langle Q \rangle * \langle Q \rangle \; \mathtt{com} \; \langle R \rangle \implies}$$

- Soundness by elimination

A Concurrent Program Logic with a Future and History

ROLAND MEYER, TU Braunschweig, Germany
THOMAS WIES, New York University, USA
SEBASTIAN WOLFF, New York University, USA

Verifying fine-grained optimistic concurrent programs remains an open problem. Modern program logics provide abstraction mechanisms and compositional reasoning principles to deal with the inherent complexity. However, their use is mostly confined to pencil-and-paper or mechanized proofs. We devise a new separation logic geared towards the lacking automation. While local reasoning is known to be crucial for automation, we are the first to show how to retain this locality for (i) reasoning about inductive properties without the need for ghost code, and (ii) reasoning about computation histories in hindsight. We implemented our new logic in a tool and used it to automatically verify challenging concurrent search structures that require inductive properties and hindsight reasoning, such as the Harris set.

CCS Concepts: • **Theory of computation → Separation logic**; **Hoare logic**; **Automated reasoning**; *Program verification*; Programming logic.

Additional Key Words and Phrases: Linearizability, Non-blocking Data Structures, Harris Set

ACM Reference Format:
Roland Meyer, Thomas Wies, and Sebastian Wolff. 2022. A Concurrent Program Logic with a Future and History. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 174 (October 2022), 30 pages. https://doi.org/10.1145/3563337

1 INTRODUCTION

Concurrency comes at a cost, at least, in terms of increased effort when verifying program correctness. There has been a proliferation of concurrent program logics that provide an arsenal of reasoning techniques to address this challenge [Bell et al. 2010; Delbianco et al. 2017; Elmas et al. 2010; Fu et al. 2010; Gotsman et al. 2013; Gu et al. 2018; Hemed et al. 2015; Jung et al. 2018; Liang and Feng 2013; Manna and Pnueli 1995; Parkinson et al. 2007; Sergey et al. 2015; Vafeiadis and Parkinson 2007]. In addition, a number of general approaches have been developed to help structure the high-level proof argument [Feldman et al. 2018, 2020; Kragl et al. 2020; O'Hearn et al. 2010; Shasha and Goodman 1988]. However, the use of these techniques has been mostly confined to manual proofs done on paper, or mechanized proofs constructed in interactive proof assistants. We distill from these works a concurrent separation logic suitable for automating the construction of local correctness proofs for highly concurrent data structures. We focus on concurrent search structures (sets and maps indexed by keys), but the developed techniques apply more broadly. Our guiding principle is to perform all inductive reasoning, both in time and space, in lock-step with the program execution. The reasoning about inductive properties of graph structures and computation histories is relegated to the meta-theory of the logic by choosing appropriate semantic models.

Running Example. We motivate our work using the Harris non-blocking set data structure [Harris 2001], which we will also use as a running example throughout the paper.

Authors' addresses: Roland Meyer, TU Braunschweig, Germany, roland.meyer@tu-bs.de; Thomas Wies, New York University, USA, wies@cs.nyu.edu; Sebastian Wolff, New York University, USA, sebastian.wolff@cs.nyu.edu.

2475-1421/2022/10-ART174
https://doi.org/10.1145/3563337

plankton

- Automatically decomposes complex updates

  ➡ eager heuristic

- First linearizability proof for Harris' set

  ➡ automated

Benchmark	Verification Time
Fine-Grained set	45*s* ✓
Lazy set	2*m* 13*s* ✓
FEMRS tree (no maintenance)	3*m* 50*s* ✓
Vechev&Yahav 2CAS set	1*m* 15*s* ✓
Vechev&Yahav CAS set	2*m* 20*s* ✓
ORVYY set	1*m* 36*s* ✓
Michael set	6*m* 53*s* ✓
Michael set (wait-free search)	6*m* 53*s* ✓
Harris set	57*m* 20*s* ✓
Harris set (wait-free search)	43*m* 00*s* ✓
LO-tree (generalized maintenance)	16*m* 43*s* ✓

Summary

Automated verification of concurrent search structures, using


the flow framework


hindsight reasoning


update chunks

Benchmark	Verification Time
Fine-Grained set	45*s* ✓
Lazy set	2*m* 13*s* ✓
FEMRS tree (no maintenance)	3*m* 50*s* ✓
Vechev&Yahav 2CAS set	1*m* 15*s* ✓
Vechev&Yahav CAS set	2*m* 20*s* ✓
ORVYY set	1*m* 36*s* ✓
Michael set	6*m* 53*s* ✓
Michael set (wait-free search)	6*m* 53*s* ✓
Harris set	57*m* 20*s* ✓
Harris set (wait-free search)	43*m* 00*s* ✓
LO-tree (generalized maintenance)	16*m* 43*s* ✓

Summary

Automated verification of concurrent search structures, using


the flow framework


hindsight reasoning


update chunks

Benchmark	Verification Time
Fine-Grained set	45*s* ✓
Lazy set	2*m* 13*s* ✓
FEMRS tree (no maintenance)	3*m* 50*s* ✓
Vechev&Yahav 2CAS set	1*m* 15*s* ✓
Vechev&Yahav CAS set	2*m* 20*s* ✓
ORVYY set	1*m* 36*s* ✓
Michael set	6*m* 53*s* ✓
Michael set (wait-free search)	6*m* 53*s* ✓
Harris set	57*m* 20*s* ✓
Harris set (wait-free search)	43*m* 00*s* ✓
LO-tree (generalized maintenance)	16*m* 43*s* ✓