Xisa: Extensible Inductive Shape Analysis

Bor-Yuh Evan Chang
U of Colorado, Boulder

Xavier Rival
INRIA/ENS Paris

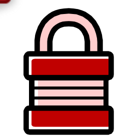
George C. Necula
U of California, Berkeley

Additional Contributors: Vincent Laviron, James Holley, Daniel Stuzman

Carnegie Mellon University – March 16, 2011

The promise of program analysis: Eliminate entire classes of bugs

For example,

- Reading from a closed file: read(); ✘
- Reacquiring a locked lock: acquire(); ✘

How?

- Systematically examine the program
- Simulate running program on "all inputs"
- "Automated code review"

Program analysis by example: Checking for double acquires

Simulate running program on “all inputs”

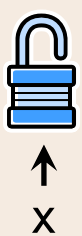
…code …

// x now points to an unlocked lock

acquire(x);

analysis state

… code …



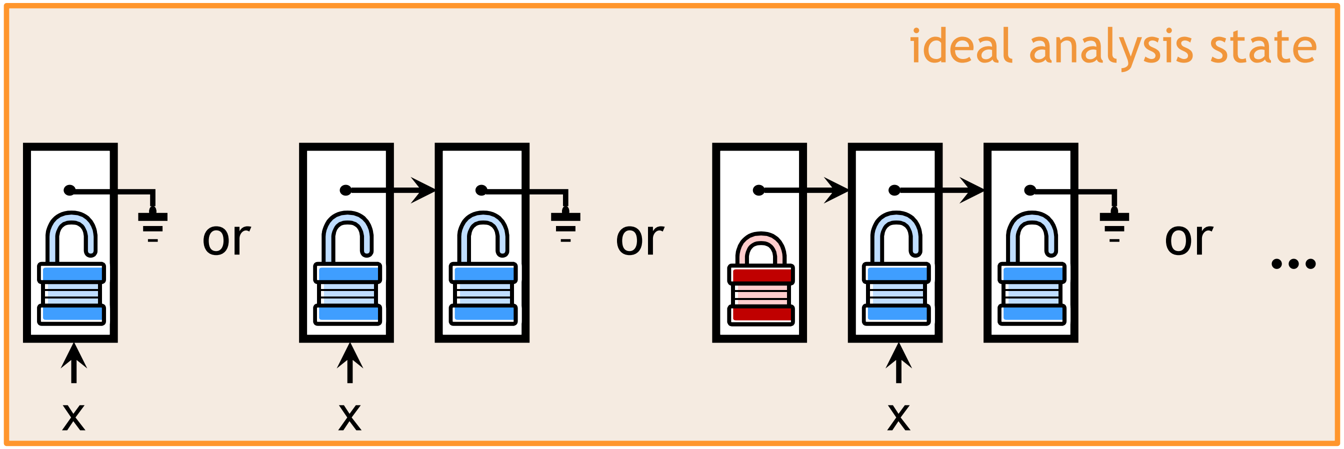
Program analysis by example: Checking for double acquires

Simulate running program on "all inputs"

undecidability

...code ...

// x now points to an unlocked lock ***in a linked list***



acquire(x);

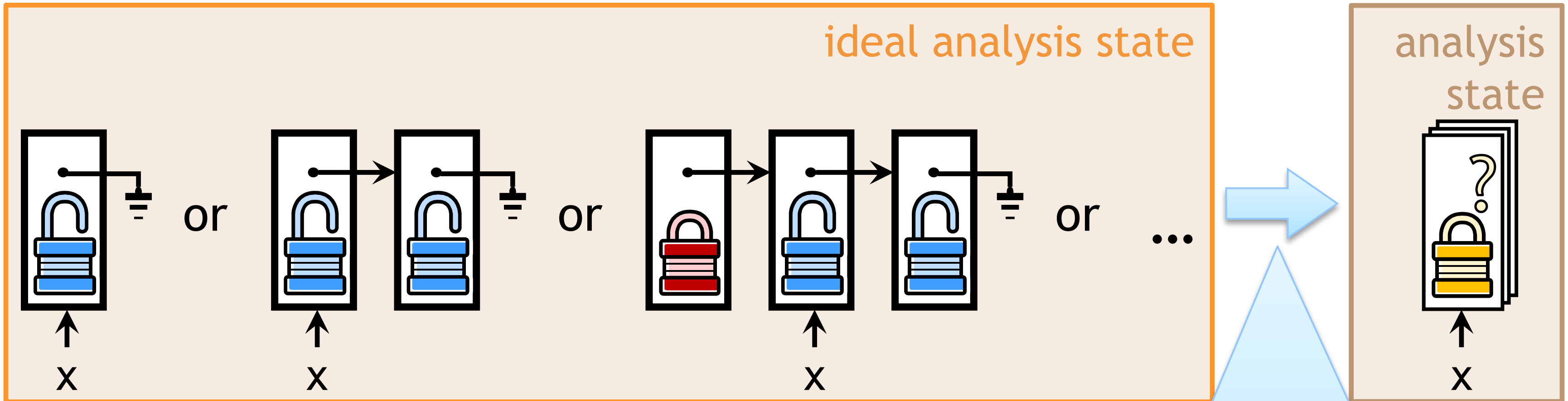
... code ...

Must abstract

Abstraction too coarse or not precise enough
(e.g., lost x is always unlocked)

...code ...

*// x now points to an unlocked lock **in a linked list***



acquire(x); ✗

mislabels good code as buggy

... code ...

For decidability, must **abstract**—"model all inputs" (e.g., merge objects)

To address the precision challenge

Traditional program analysis mentality:

“Why can’t developers write more specifications for our analysis? Then, we could verify so much more.”

“Since developers won’t write specifications, we will use default abstractions (perhaps coarse) that work hopefully most of the time.”

Cooperative approach:

“Can we design program analyses around the user? Developers write testing code. Can we adapt the analysis to use those as specifications?”

Summary of overview

Challenge in analysis: Finding a good abstraction

precise enough but not more than necessary

Powerful, generic abstractions

expensive, hard to use and understand

Built-in, default abstractions

often not precise enough (e.g., data structures)

Cooperative approach:

Must involve the user in abstraction

without expecting the user to be a program analysis expert

Overview of contributions

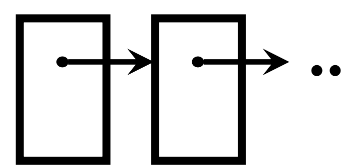
Extensible Inductive Shape Analysis (Xisa)

- Precise inference of data structure properties
  - Able to check, for instance, the locking example
- Targeted to software developers
  - Uses data structure checking code for guidance
    - Turns testing code into a specification for static analysis
- Efficient
    - Builds abstraction out of developer-supplied checking code

End-user approach

Extensible Inductive Shape Analysis

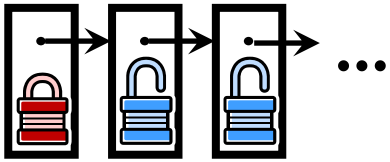
Precise inference of
data structure properties



Shape analysis is a fundamental analysis

Precise heap abstraction needed to analyze
- Traditional languages (C, Java)
- Web scripting languages

Improves verifiers that try to
- Eliminate resource usage bugs (locks, file handles)



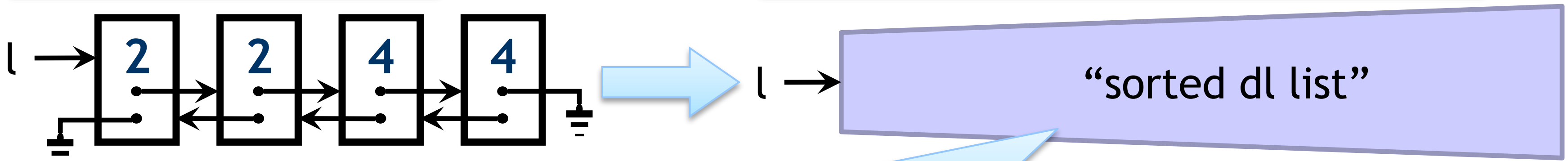
- Eliminate memory errors (leaks, dangling pointers)
- Eliminate concurrency errors (data races)
- Validate developer assertions

Enables program transformations
- Compile-time garbage collection
- Data structure refactorings

Shape analysis by example: Removing duplicates

<u>Example/Testing</u>

<u>Code Review/Static Analysis</u>



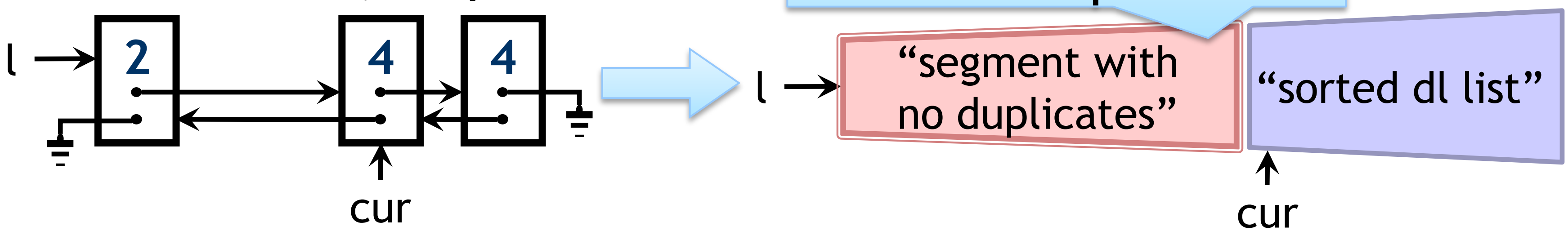
// l is a sorted doubly-linked li

program-specific

for each node cur in list l {

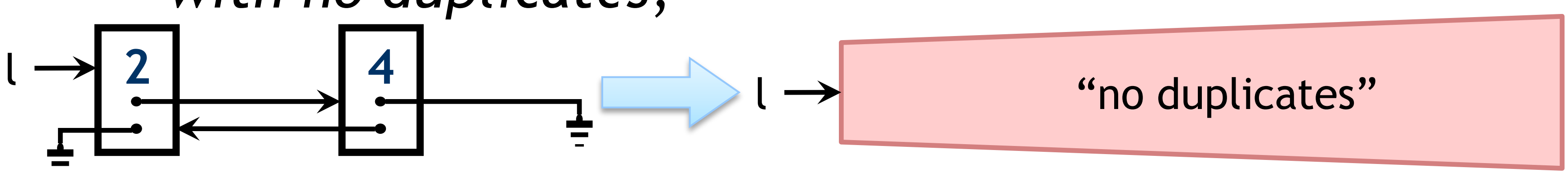
remove cur if duplicate;

intermediate state more complicated



}

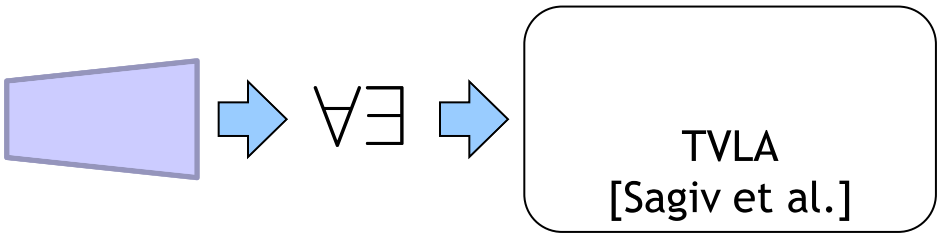
***assert** l is sorted, doubly-linked with no duplicates;*



Shape analysis is not yet practical

Choosing the heap abstraction difficult for precision

Some representative approaches:



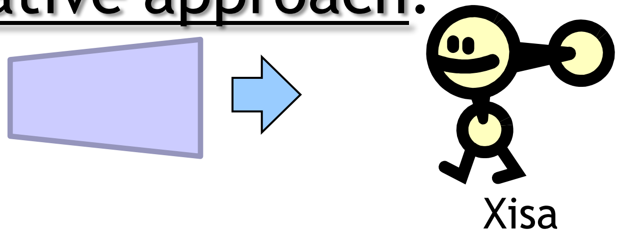
Parametric in low-level, analyzer-oriented predicates

+ Very general and expressive
- Harder for non-expert

Built-in high-level predicates

- Harder to extend
+ No additional user effort (if precise enough)

Cooperative approach:



Parametric in high-level, developer-oriented predicates

+ Extensible
+ Targeted at developers

Our approach: Executable specifications

Utilize “run-time **checking code**” as specification for static analysis.

$$
\begin{aligned}
&h.\mathrm{dll}(p) := \\
&\quad h = \mathbf{null} \wedge \mathbf{emp} \\
&\vee\ \exists n.\ h \neq \mathbf{null} \wedge \\
&\qquad h\cdot \mathrm{prev} \mapsto p\ * \\
&\qquad h\cdot \mathrm{next} \mapsto n\ * \\
&\qquad n.\mathrm{dll}(h)
\end{aligned}
$$

checker

- p specifies where prev should point

Contribution:
Build the abstraction for analysis out of developer-specified checking code

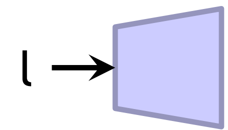
Contribution:
Generalize checkers for complicated intermediate states

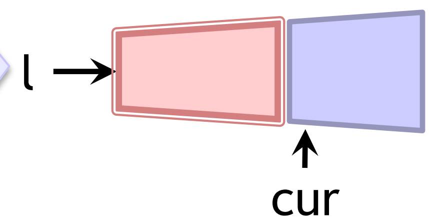
;

ist l {

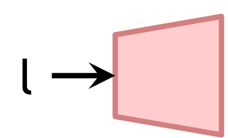
icate;

}





assert(l.sorted_dll_nodup(...));

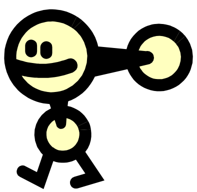


Xisa is …

An automated shape analysis with a precise memory abstraction based around invariant checkers.

```
h.dll(p) =
  if (h = null) then
    true
  else
    h→prev = prev  and
    h→next.dll(h)
```

checkers

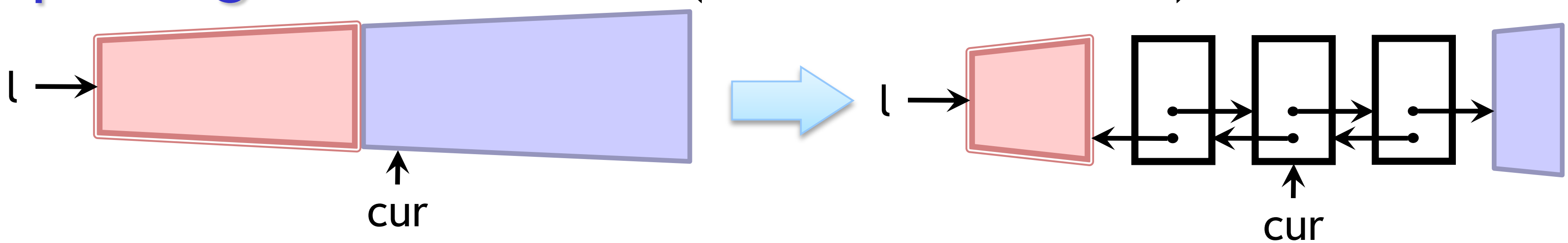


Xisa

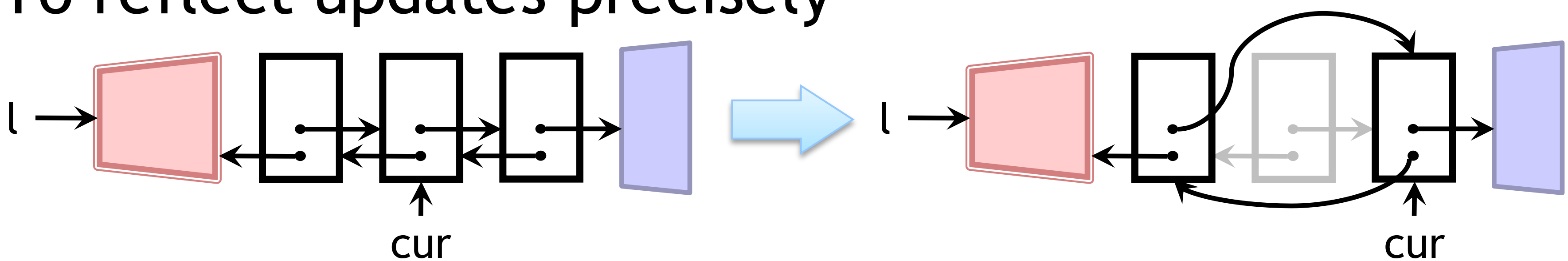
- Extensible and targeted for developers
  - Parametric in developer-supplied checkers—viewed as inductive definitions in separation logic
- Precise yet compact abstraction for efficiency
  - Data structure-specific based on properties of interest to the developer

Shape analysis is an abstract interpretation on abstract memory descriptions with …

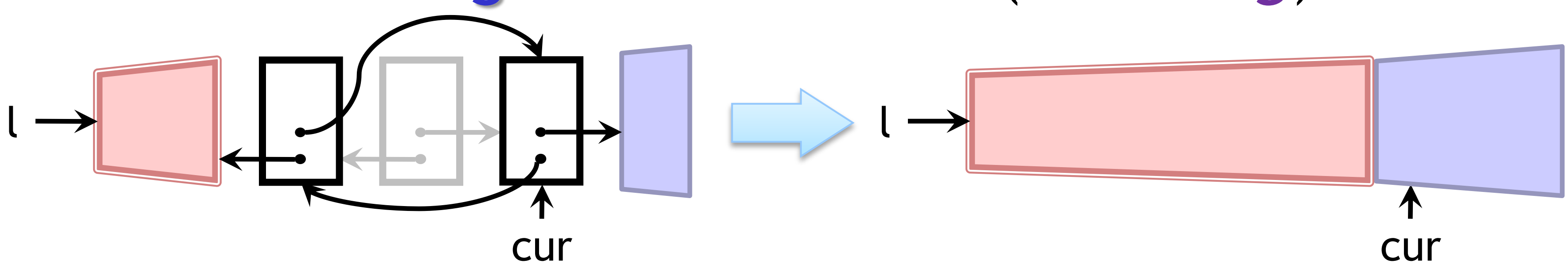
Splitting of summaries (*materialization*)



To reflect updates precisely



And summarizing for termination (*widening*)

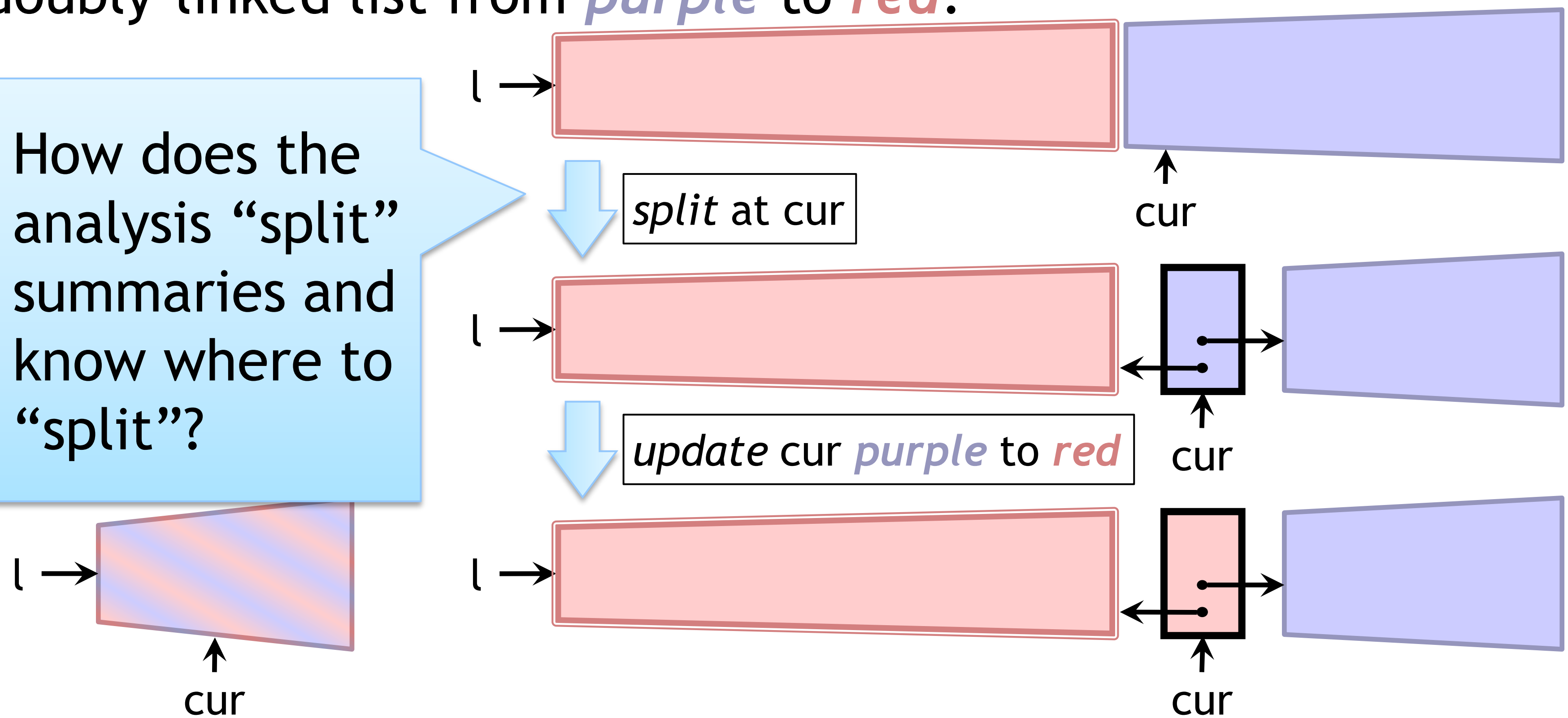


Must materialize summaries to interpret updates precisely

Want abstract update to be “exact”, that is, to update one “concrete memory cell”.

The example at a high-level: iterate using cur changing the doubly-linked list from *purple* to *red*.

How does the analysis “split” summaries and know where to “split”?



Roadmap: Components of Xisa

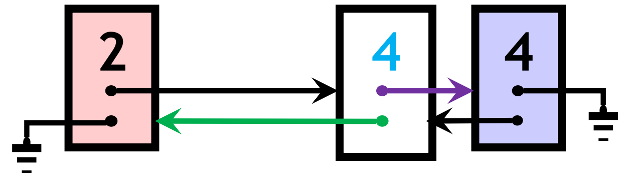
Defining a program analysis:

1. The abstraction (e.g., separation logic formulas with inductive definitions) and operations on the abstraction (e.g., unfolding, update)
2. How to effectively apply the operations (harder!)

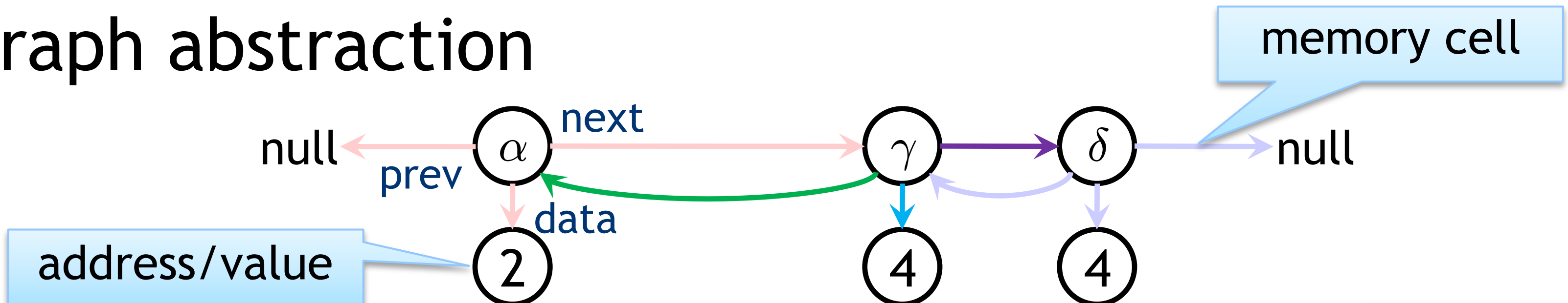
Challenge: Checkers are incomplete specs

Memory abstraction as separating shape graphs

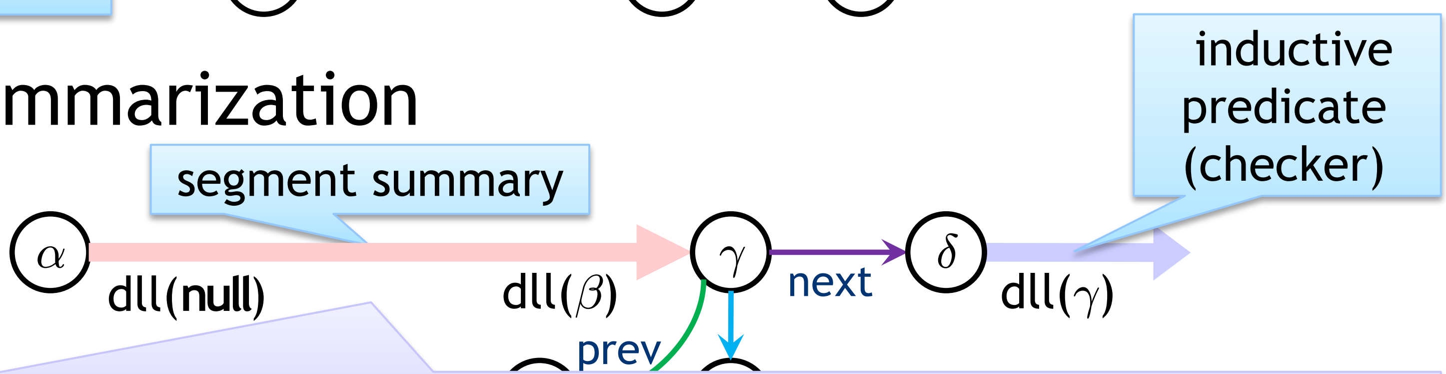
Memory partitioned into regions



Graph abstraction



Region summarization

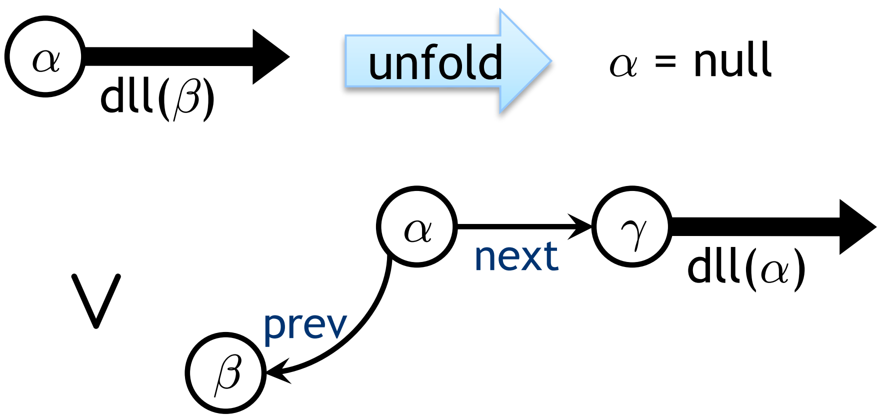


Segment generalization of a checker (α.dll(null) up to γ.dll(β))

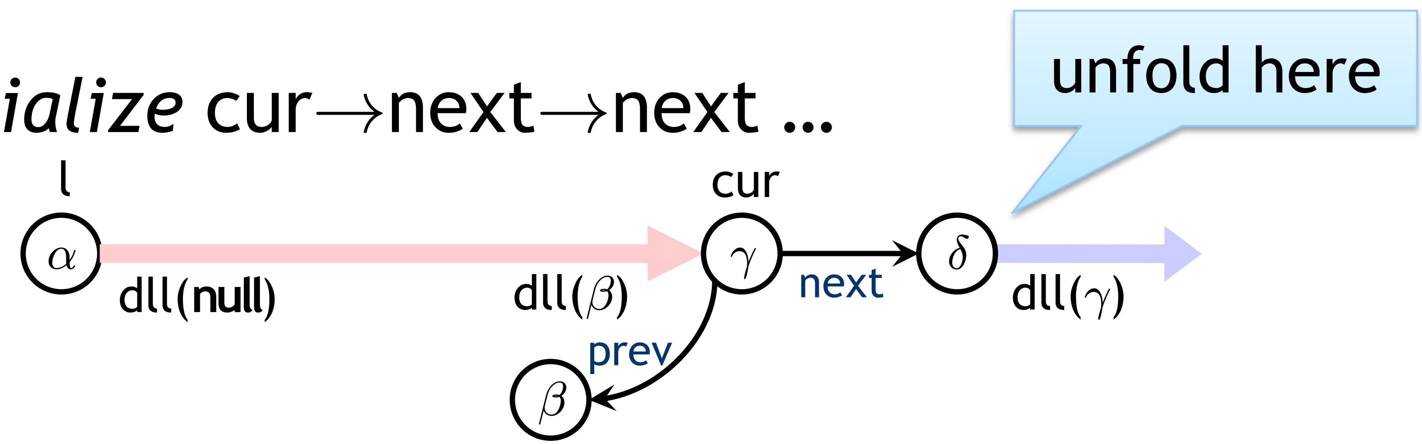
Unfold inductive definitions to split summaries

Definition yields graph unfolding rules

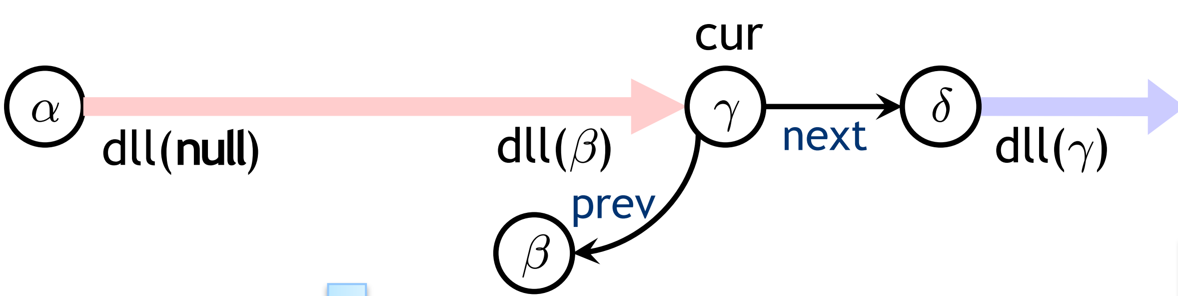
h.dll(p) :=
  h = null ∧ **emp**
∨ ∃n. h ≠ null ∧
    h·prev ↦ p ∗
    h·next ↦ n ∗
    n.dll(h)



To *materialize* cur→next→next …



Also need a “backwards” unfolding



cur→prev→next
= cur→next;

for each node cur in list l {
remove cur *if duplicate*;

Technical Details:

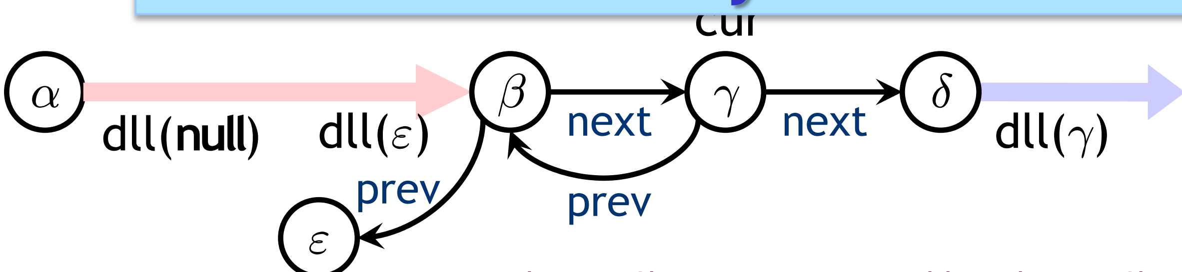
How does the analysis do this unfolding?

Why is this unfolding allowed?

(Key: Segments are also inductively defined)

[POPL’08]

How does the analysis know to do this unfolding?

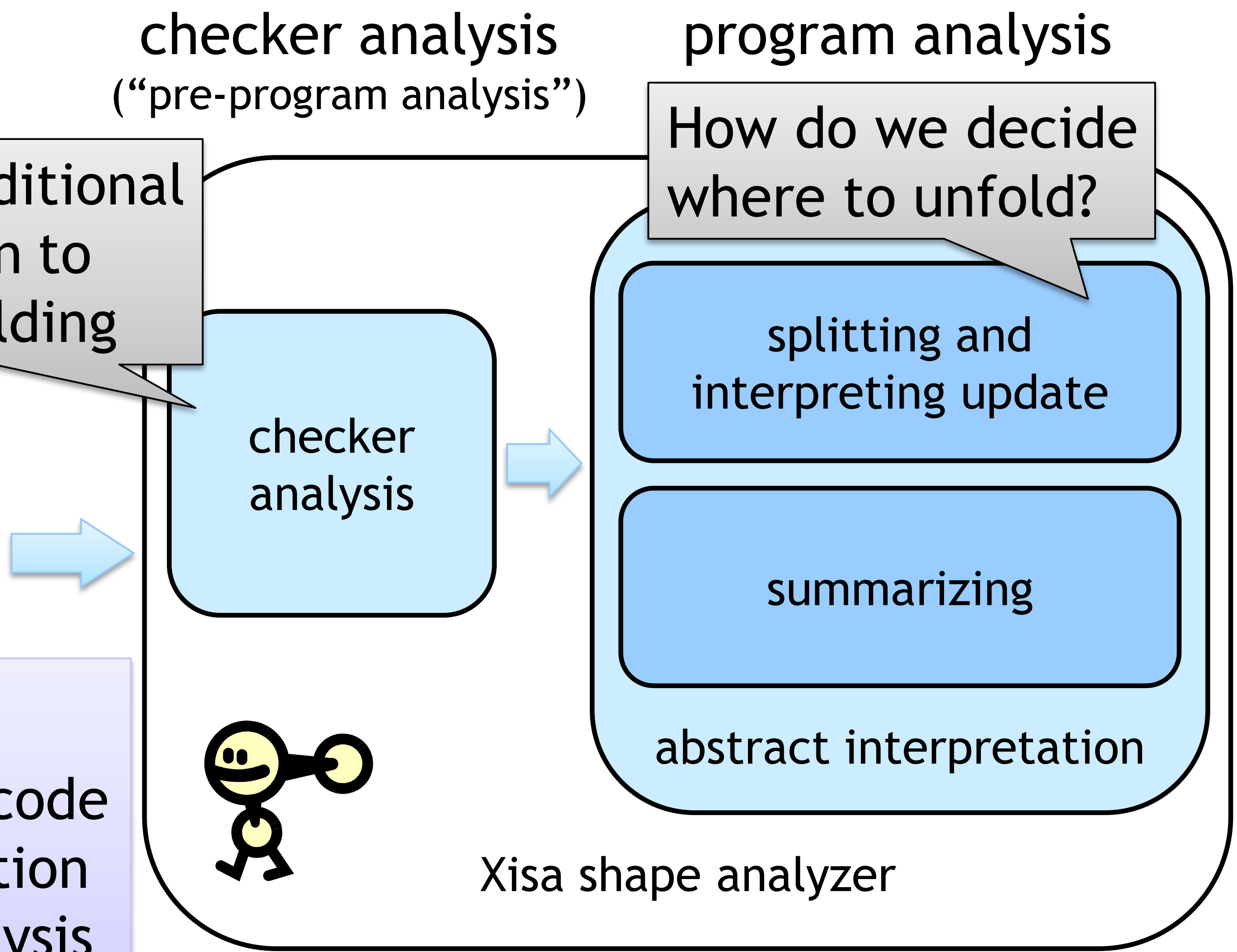


Roadmap: Components of Xisa

Derives additional information to guide unfolding

How do we decide where to unfold?

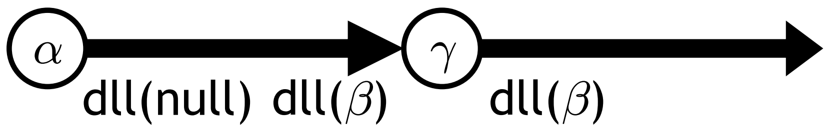
```
h.dll(p) =
  if (h = null) then
    true
  else
    h→prev = prev  and
    h→next.dll(h)
```



Contribution:
Turns testing code into specification for static analysis

Level types for deciding where to unfold

Summary

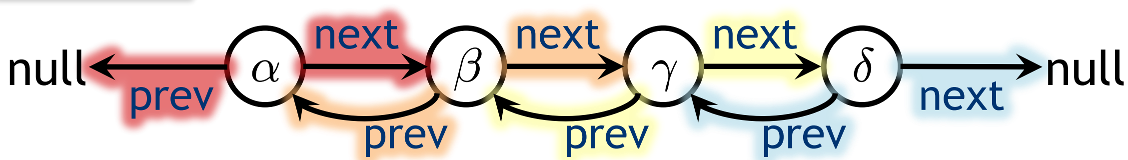


If it exists, where is:

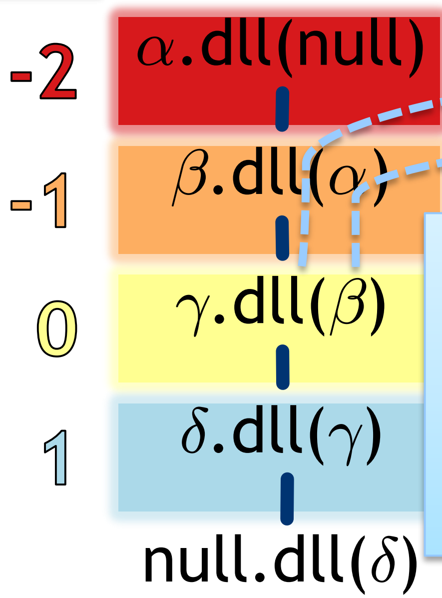
$\gamma \rightarrow$next ? 0

$\beta \rightarrow$next ? -1

Instance



Checker "Run" (call tree/derivation)



Says:

For h$\rightarrow$next/h$\rightarrow$prev, unfold **from** h

For p$\rightarrow$next/p$\rightarrow$prev, unfold **before** h

Checker Definition

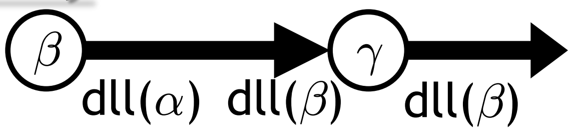
h:{next⟨0⟩, prev⟨0⟩ }
p:{next⟨-1⟩,prev⟨-1⟩}

h.dll(p) =
  if (h = null) **then**
    true
  else
    h$\rightarrow$prev = p **and**
    h$\rightarrow$next.dll(h)

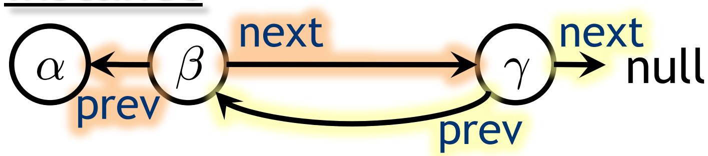
Level types make the analysis robust with respect to how checkers are written

Doubly-linked list checker (as before)

Summary



Instance

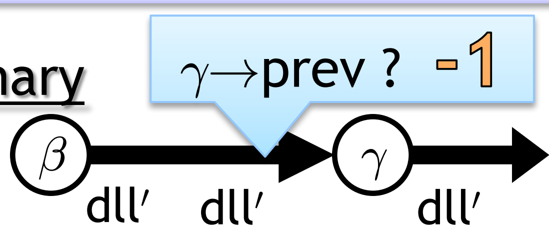


$h : \{next\langle 0\rangle,\ prev\langle 0\rangle\}$
$p : \{next\langle -1\rangle, prev\langle -1\rangle\}$

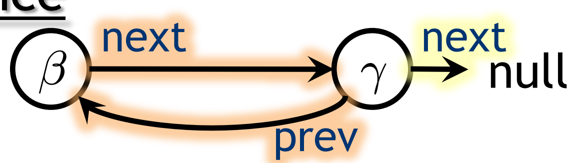
h.dll(p) =
  if (h $=$ **null**) **then**
    true
  else
    h$\to$prev $=$ p **and**
    h$\to$next.dll(h)

Alternative doubly-linked list checker

Summary



Instance



Different types for different unfolding

$h : \{next\langle 0\rangle,\ prev\langle -1\rangle\}$

h.dll′() =
  if (h$\to$next $=$ **null**) **then**
    true
  else
    h$\to$next$\to$prev $=$ h
    and h$\to$next.dll′()

Summary of checker parameter types

Tell where to unfold for which fields

Make analysis robust with respect to how checkers are written

- Learn where in summaries unfolding won't help

Can be inferred automatically with a fixed-point computation on the checker definitions

Summary of interpreting updates

Splitting of summaries needed for precision

Unfolding checkers is a natural way to do splitting

- When checker traversal matches code traversal

Checker parameter type analysis

- Useful for guiding unfolding in difficult cases, for example, “back pointer” traversals

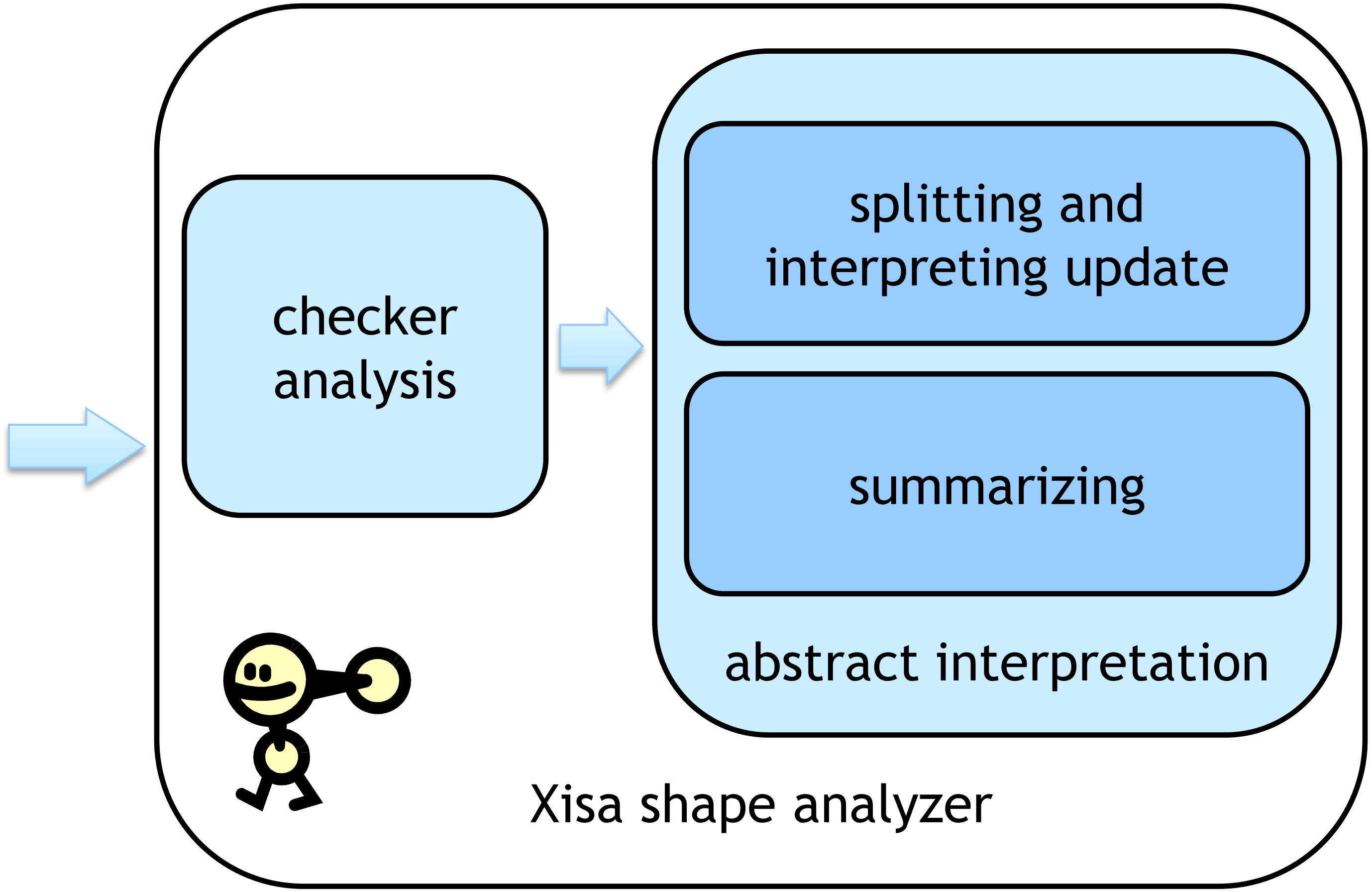
Roadmap: Components of Xisa

checker analysis
("pre-program analysis")

program analysis

```
h.dll(p) =
  if (h = null) then
    true
  else
    h→prev = prev  and
    h→next.dll(h)
```

checkers

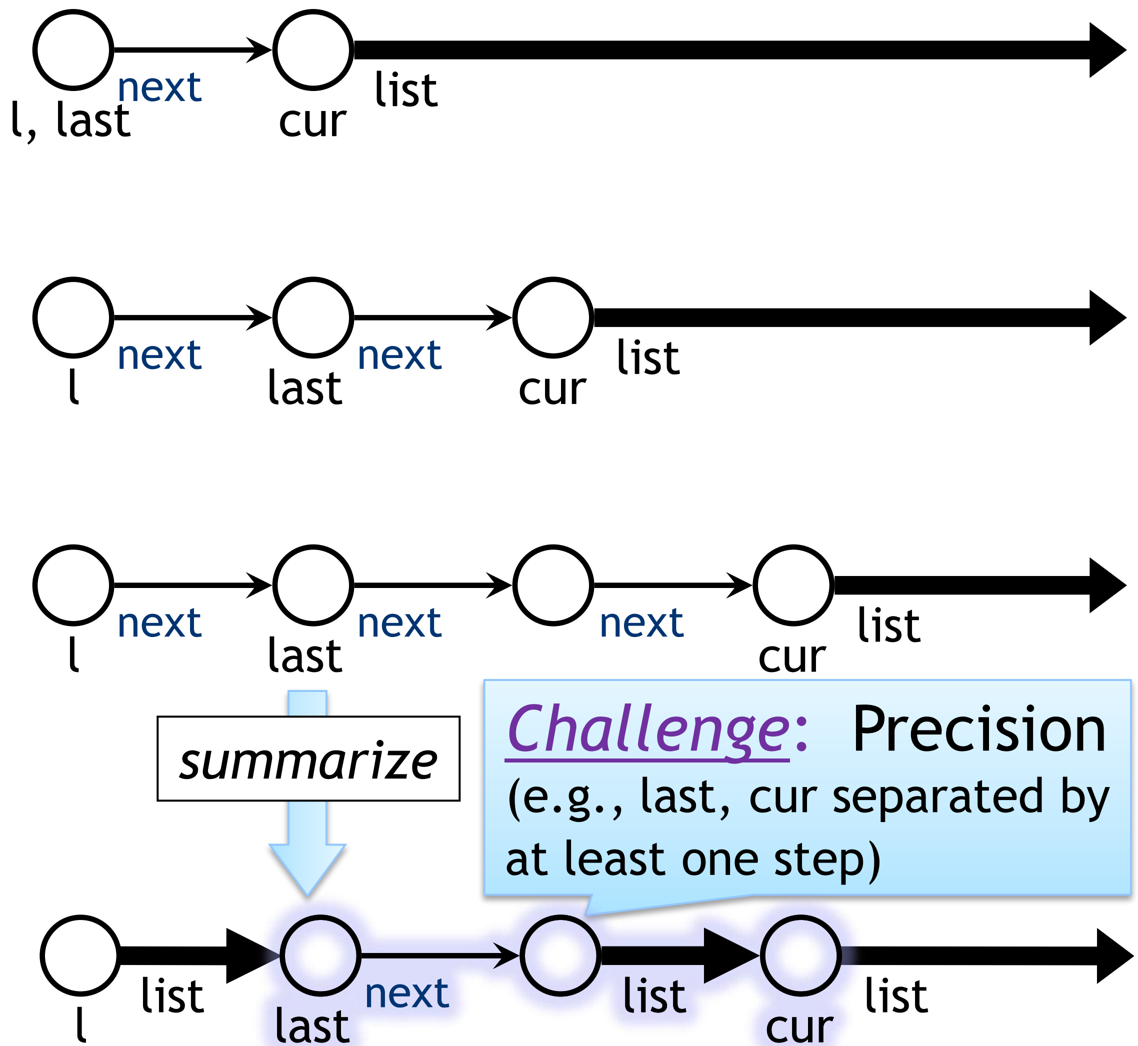


Summarize by folding into inductive predicates

```
last = l;
cur = l→next;
while (cur != null) {
    // … cur, last …
    if (…) last = cur;
    cur = cur→ next;
}
```

Previous approaches guess where to fold for each graph.

Contribution:
Determine where by comparing graphs across history

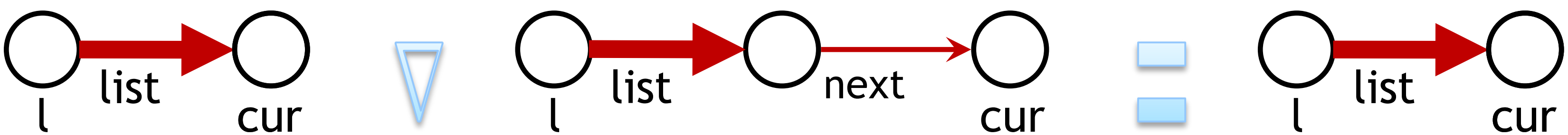


Challenge: Precision
(e.g., last, cur separated by at least one step)

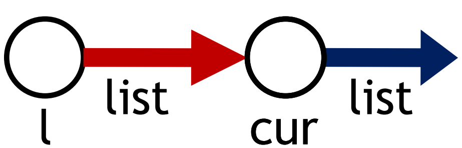
Use iteration history with a widening operator

Match regions

Apply local weakening rules on each region



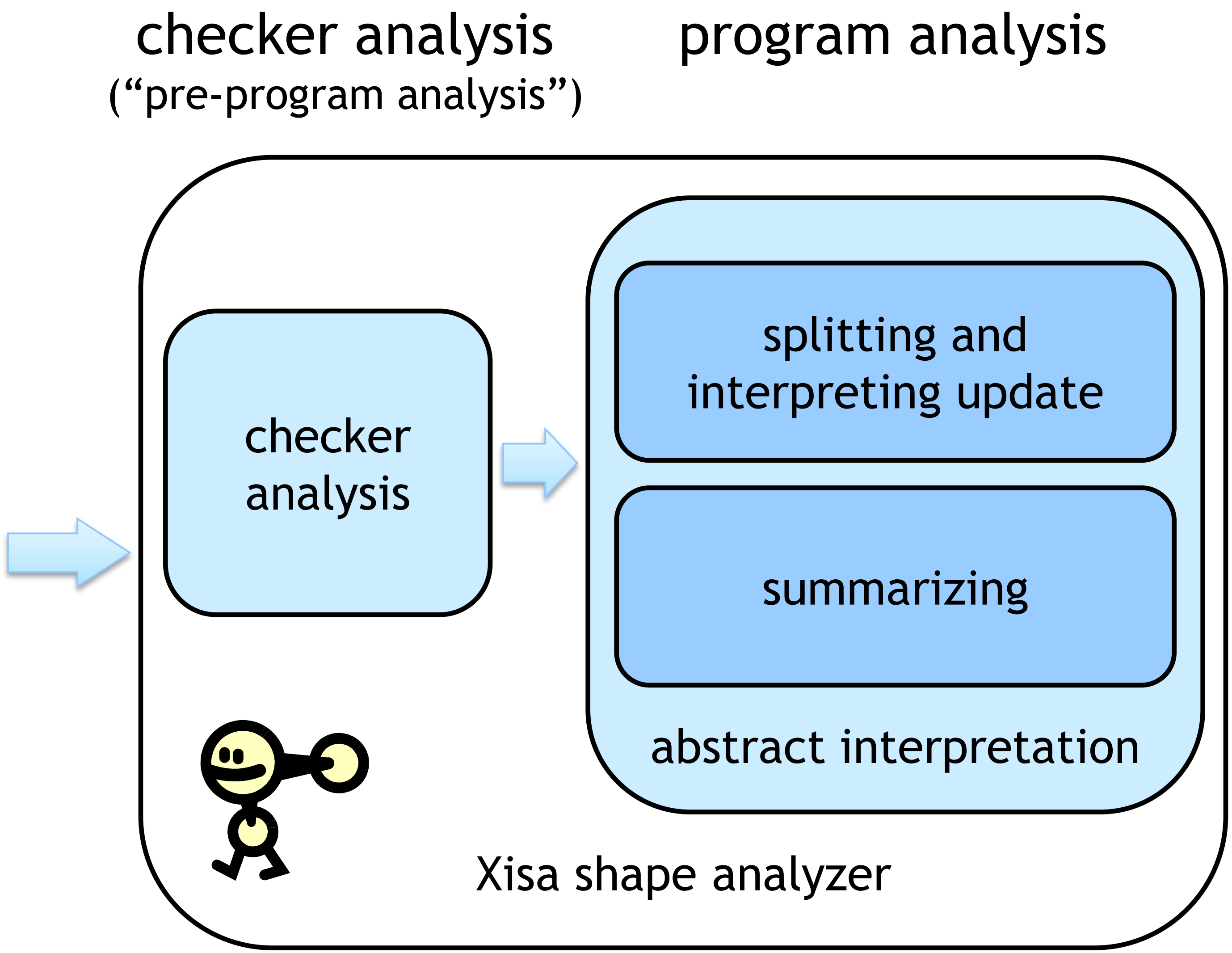
Widened result



Given checkers, everything is automatic

```
h.dll(p) =
  if (h = null) then
    true
  else
    h→prev = prev  and
    h→next.dll(h)
```

checkers



Results: Performance

Expressiveness: Different data structures

Times negligible for data structure operations (often in sec or $^{1}/_{10}$ sec)

Benchmark	Max. Num. Graphs at a Program Pt	Analysis Time (ms)
singly-linked list reverse	1	1.0
doubly-linked list reverse		1.5
doubly-linked list copy		5.4
doubly-linked list remove		17.9
doubly-linked list remove and back	5	18.1
search tree with parent insert	3	16.6
search tree with parent insertand back	5	64.7
two-level skip list rebalance	1	11.7
Linux `scull` driver (894 loc) (char arrays ignored, functions inlined)	4	3969.6

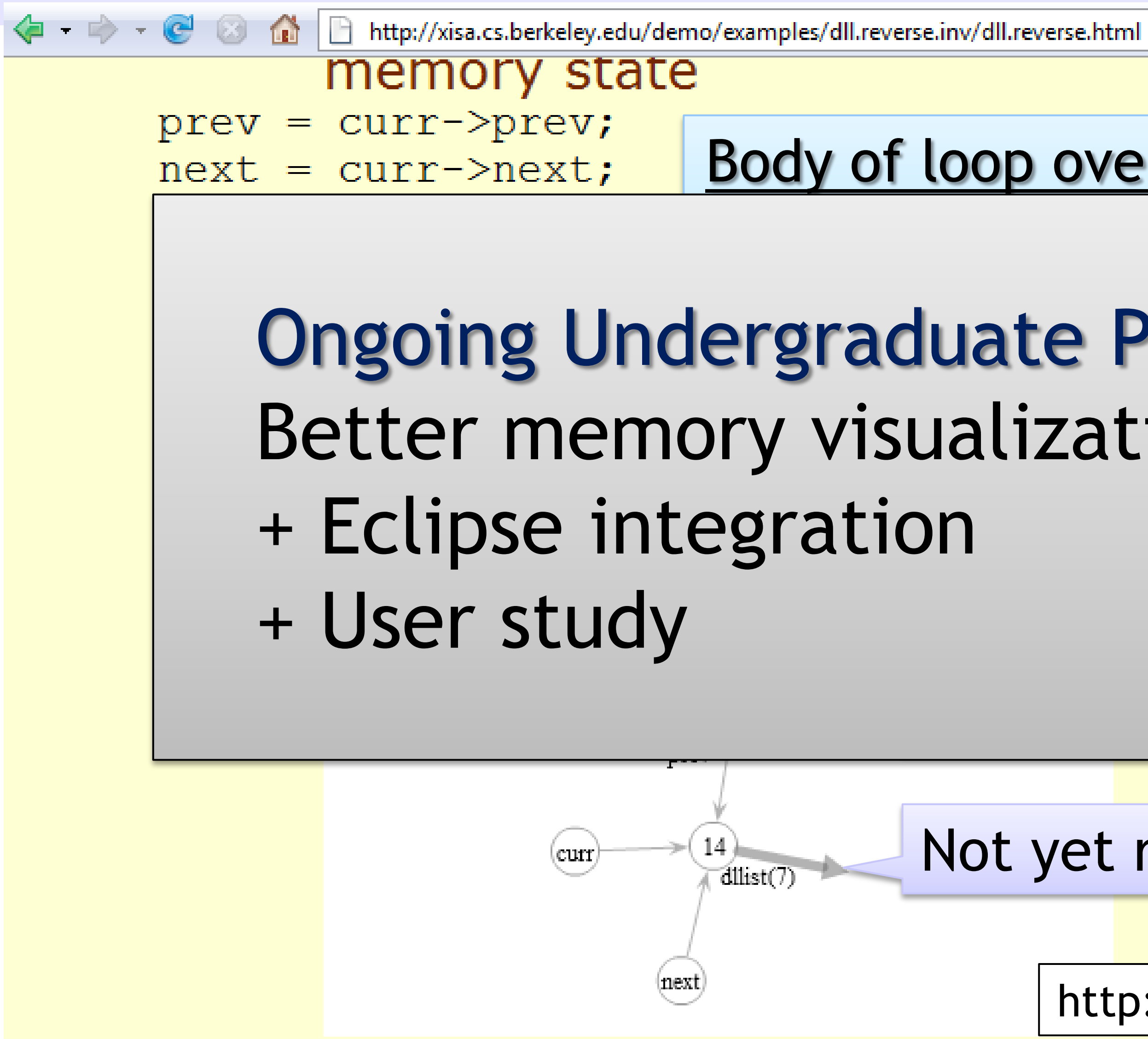
TVLA: 290 ms (singly-linked list reverse)

Space Invader only analyzes lists (built-in)

TVLA: 850 ms (search tree with parent insert)

Verified shape invariant as given by the checker is preserved across the operation.

Demo: Doubly-linked list reversal



Body of loop over the elements:

lds

Ongoing Undergraduate Project:
Better memory visualization
+ Eclipse integration
+ User study

ment

d
pped

Not yet reversed list

http://xisa.cs.colorado.edu/

Summary of Xisa: Extensible Inductive Shape Analysis

Key Insight: Checkers as specifications

Developer View: Global, Expressed in a familiar style

Analysis View: Capture developer intent,
Not arbitrary inductive definitions

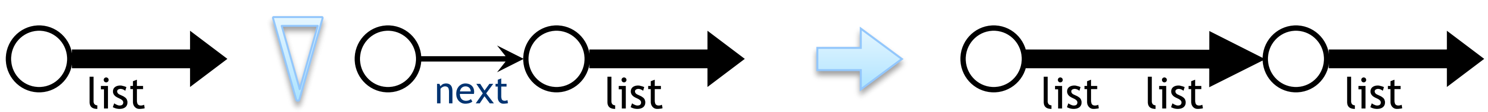
Constructing the program analysis

Intermediate states: **Generalized segment** predicates

Splitting: Checker parameter **types with levels**

$h : \{next\langle 0\rangle, prev\langle 0\rangle\}$ $p : \{next\langle -1\rangle, prev\langle -1\rangle\}$

Summarizing: **History-guided** approach with widening op



Subsequent Work

- C-Level Memory Abstraction [ESOP'10]
  - Separating shape graphs support mixing high-level (e.g., record fields) and low-level (e.g., union fields) memory abstractions
- "Very Context-Sensitive" Interprocedural Analysis [POPL'11]
  - Whole program, state-based interprocedural analysis using Xisa
  - Make call stack explicit and summarize using shape invariants

Future work: Exploiting common specification framework

Scenario: Code instrumented with lots of checker calls (perhaps automatically with object invariants)

```
assert( mychecker(x) );
// … operation on x …
assert( mychecker(x) );
```

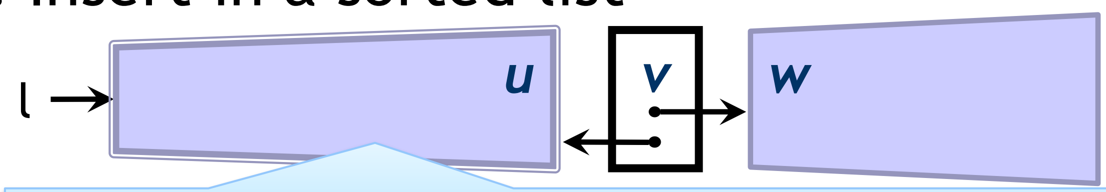
- Very slow to execute
- Hard to prove statically (in general)

Can we prove parts statically?

Static Analysis View: Hybrid checking

Testing View: Incrementalize invariant checking

Example: Insert in a sorted list



Preservation of sortedness shown statically

Emit run-time check for new element: $u \leq v \leq w$

Conclusion

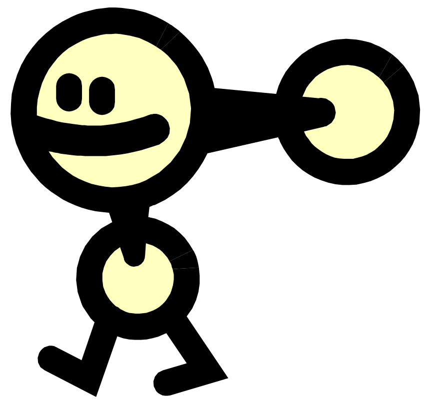
Extensible Inductive Shape Analysis

precision demanding program analysis
improved by novel user interaction

Developer: Gets results corresponding to intuition

Analysis: Focused on what's important to the developer

Practical precise tools for better software with a cooperative approach!



What can inductive shape analysis do for you?

http://xisa.cs.colorado.edu