Materialization in Shape Analysis with Structural Invariant Checkers

Bor-Yuh Evan Chang
Xavier Rival
George C. Necula

University of California, Berkeley

August 27, 2007
ITU Copenhagen

What's shape analysis? What's special?

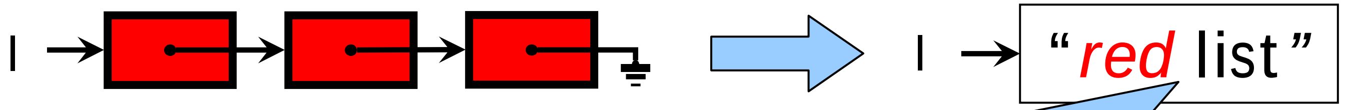
Shape analysis tracks memory manipulation in a flow-sensitive manner.

- Memory manipulation
  - Particularly important in systems code (in C)
- Flow-sensitive
  - Many important properties
    - E.g., Is an object freed? Is a file open?
  - Heap abstracted differently at different points
    - E.g., Not based on allocation site

Example: Typestate with shape analysis

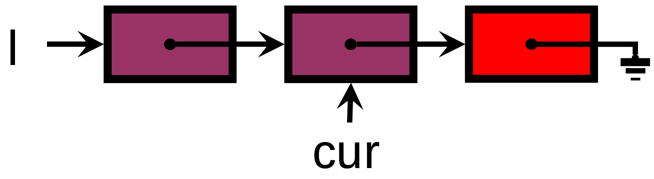
Concrete Example

Abstraction



program-specific predicate

```
cur = l;
while (cur != null) {
   assert(cur is red);
   make_purple(cur);
```



```
   cur = cur→next;
}
```

flow-sensitive heap abstraction

make_purple(·) could be

- lock(·)
- free(·)
- open(·)
- …

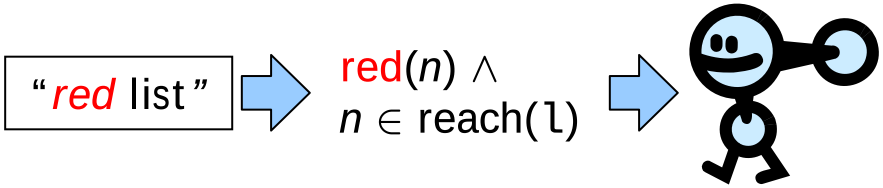
Shape analysis is not yet practical

Usability: Choosing the heap abstraction difficult

Space Invader
[Distefano et al.]

Built-in high-level predicates

- - Hard to extend
- + No additional user effort

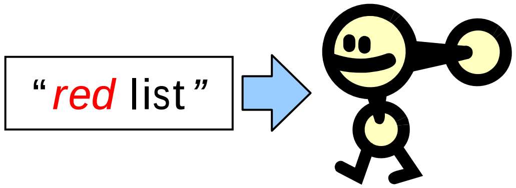


$\text{red}(n) \land$
$n \in \text{reach}(\texttt{l})$

TVLA
[Sagiv et al.]

Parametric in low-level, analyzer-oriented predicates

- + Very general and expressive
- - Hard for non-expert



Our Proposal

Parametric in high-level, developer-oriented predicates

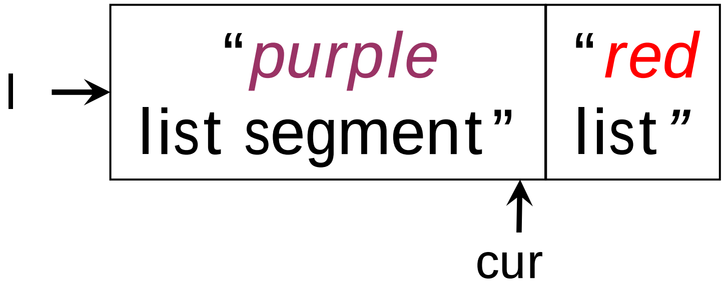
- + Extensible
- + Easier for developers

Shape analysis is not yet practical

Scalability: Finding right level of abstraction difficult

➡ Over-reliance on disjunction for precision

developer

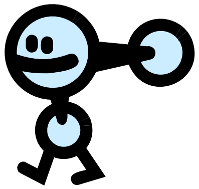


l

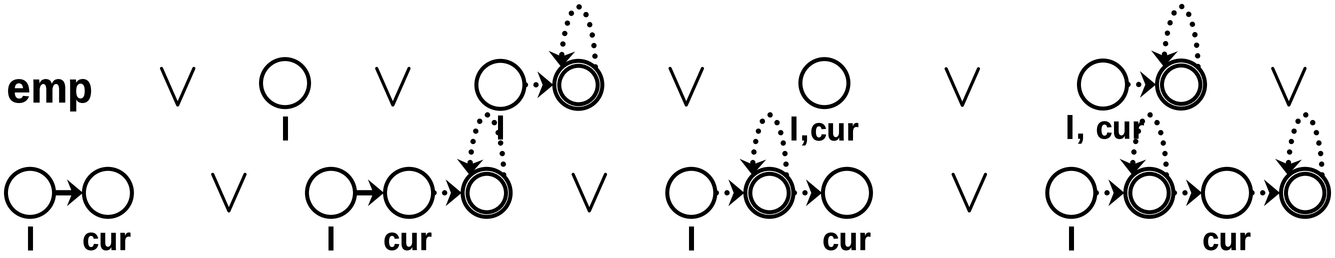
“*purple* list segment”

“*red* list”

cur



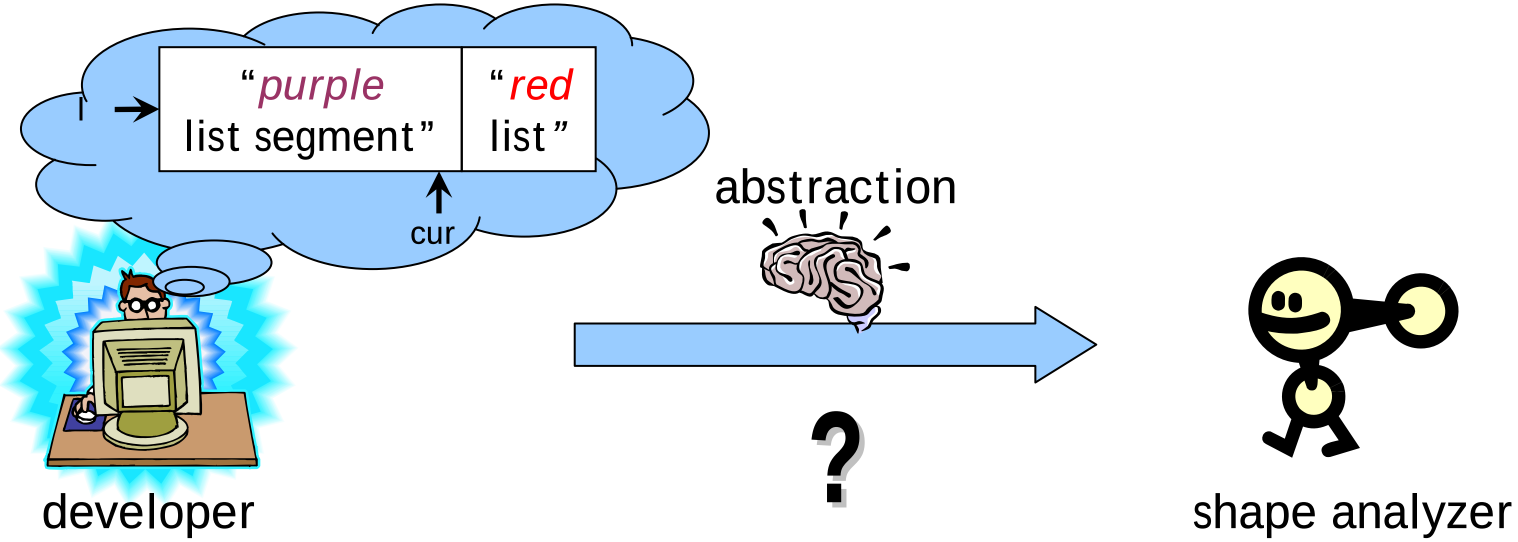
shape analyzer



Hypothesis

The developer can describe the memory in a compact manner at an abstraction level sufficient for the properties of interest (at least informally).

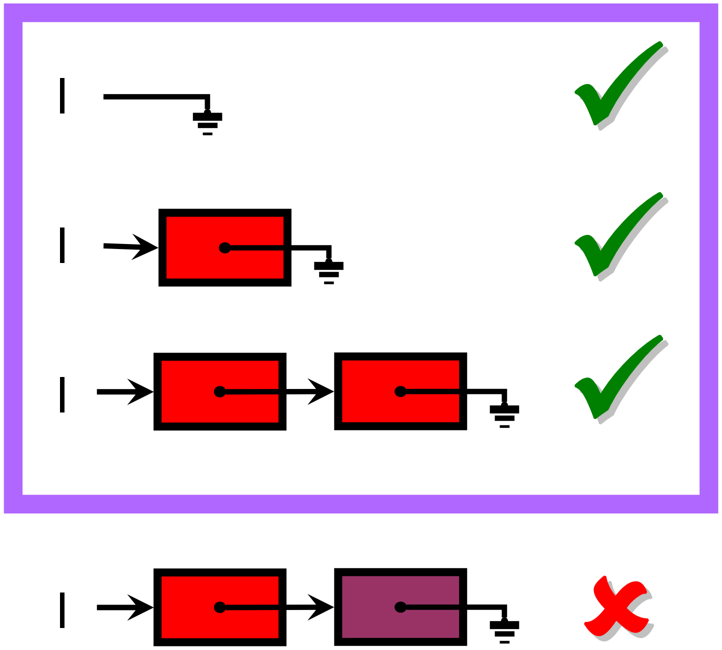
- Good abstraction is program-specific



Observation

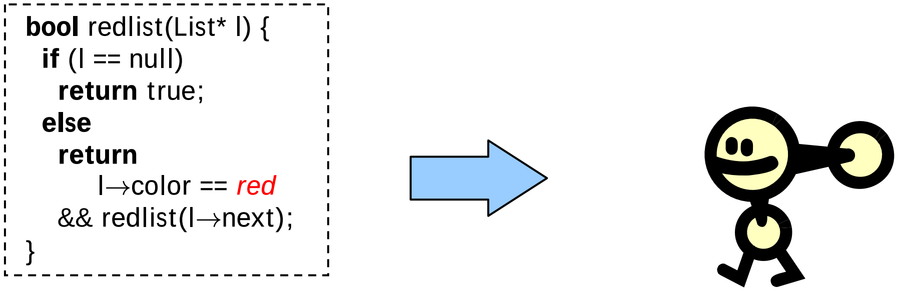
Checking code expresses a shape invariant and an intended usage pattern.

```
bool redlist(List* l) {
   if (l == null)
      return true;
   else
      return
            l→color == red
         &&  redlist(l→next);
}
```



Proposal

An automated shape analysis with a memory abstraction parameterized by invariant checkers.



```
bool redlist(List* l) {
  if (l == null)
    return true;
  else
    return
         l→color == red
    && redlist(l→next);
}
```

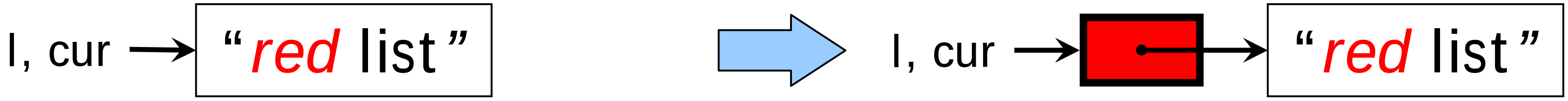
checkers

shape analyzer

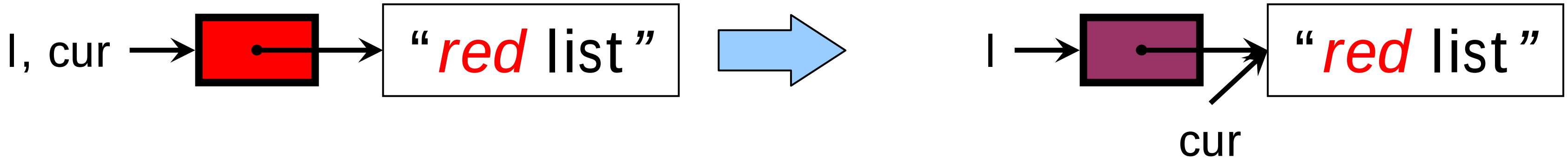
- Extensible
  - Abstraction based on the developer-supplied checkers
- Targeted for Usability
  - Global data structure specification, local invariant inference
- Targeted for Scalability
  - Based on the hypothesis

Shape analysis is an abstract interpretation on memory states with …

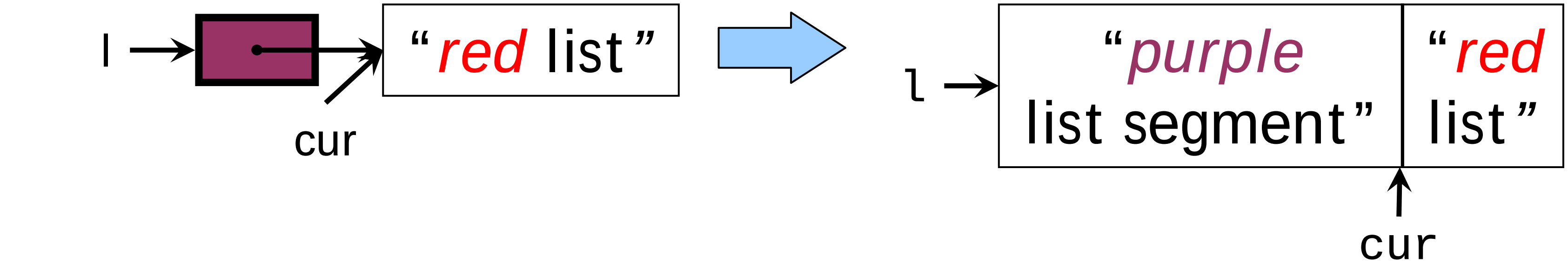
- Materialization (partial concretization)



- To perform strong updates



- And widening for termination



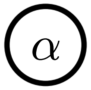
Outline

- **Memory abstraction**
  - **Restrictions on checkers**
  - **Challenge: Intermediate invariants**
- Materialization by forward unfolding
  - Where and how
  - Challenge: Unfolding segments
- Materialization by backward unfolding
  - Challenge: Back pointers
- Deciding where to unfold generically

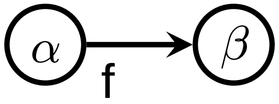
Abstract memory using ch

"Some number of points-to edges that satisfies checker c"

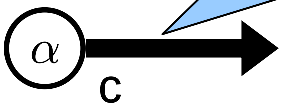
Graphs



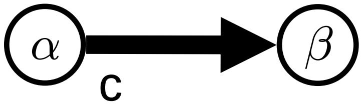
values
(address or null)



points-to relation
α@f $\mapsto$ β



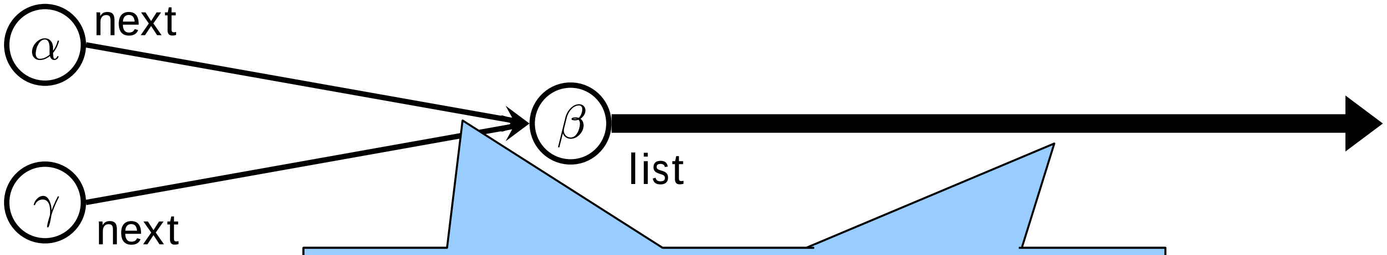
checker run
c(α)



partial run
?

Example

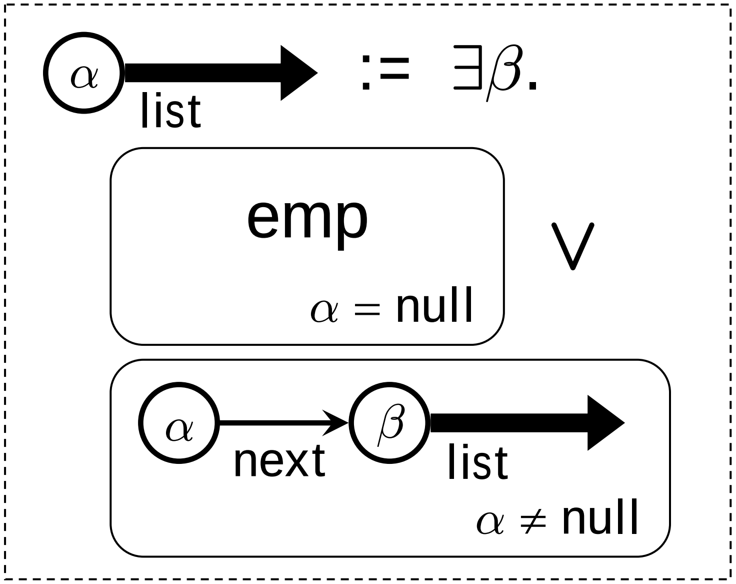
"Disjointly, $\alpha \rightarrow$next = β, $\gamma \rightarrow$next = β, and β is a list."



disjoint memory regions (∗)

Checkers as inductive definitions

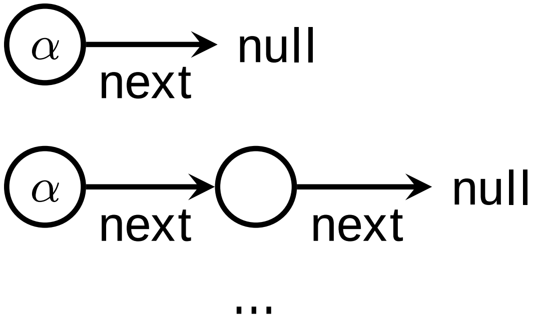
```
bool list(List* l) {
  if (l == null)
    return true;
  else
    return list(l→next);
}
```



Disjointness

Checker run can dereference any object field only once

emp $(\alpha = \text{null})$



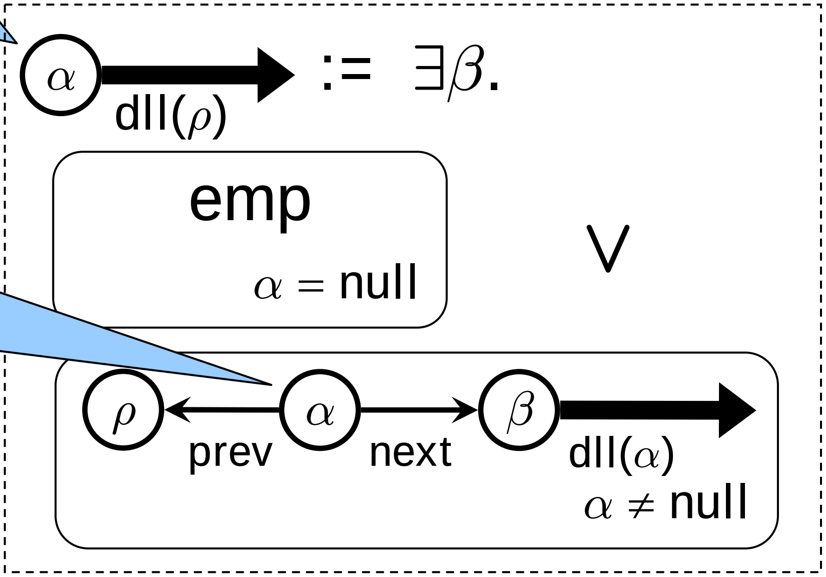
What can a checker do?

- In this talk, a *checker* …
  - is a pure, recursive function
  - dereferences any object field only once during a run
  - only one argument can be dereferenced (traversal arg)
  - has only additional pointer parameters

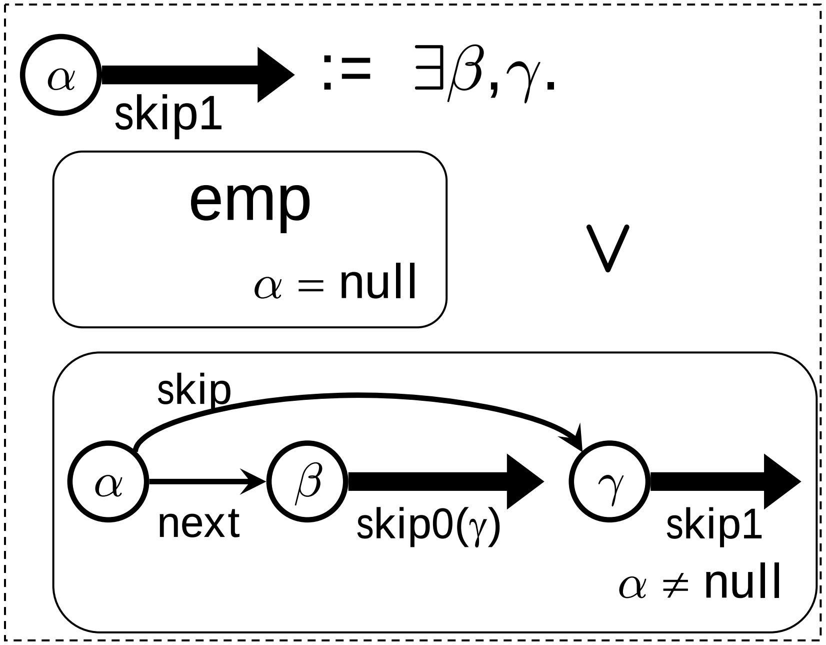
```
bool dll(Dll* l, Dll* prev) {
  if (l == null) return true;
  else
    return l→pr
         && dll(l→next);
}
```

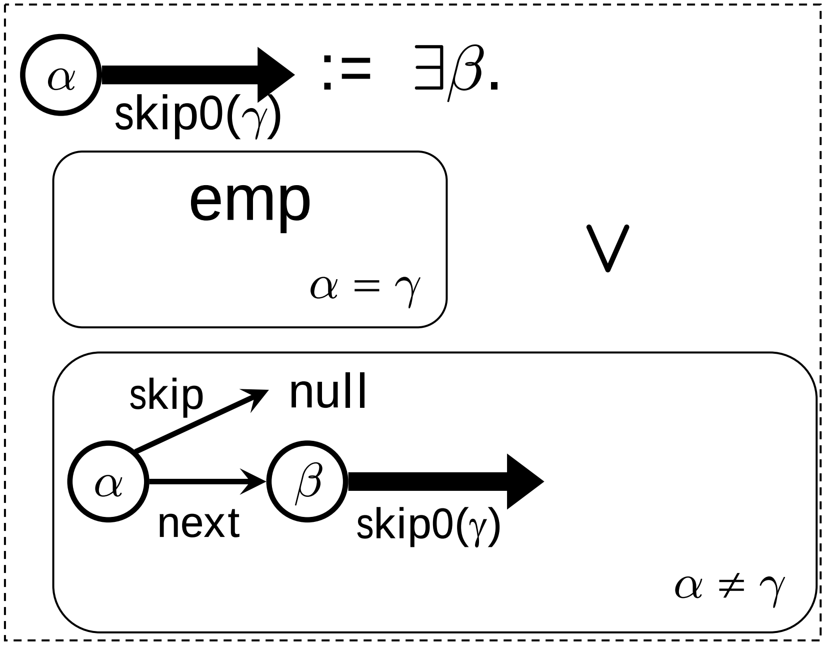
Traversal argument

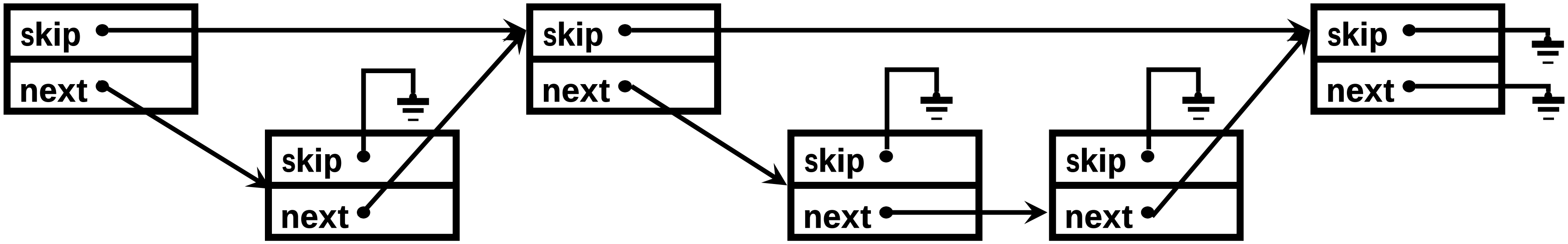
Only fields from traversal argument



Example checker: Two-level skip list



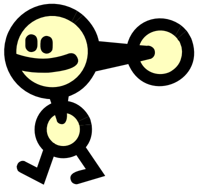




back to the abstract domain ...

```
bool redlist(List* l) {
  if (l == null)
    return true;
  else
    return
         l→color == red
    && redlist(l→next);
}
```

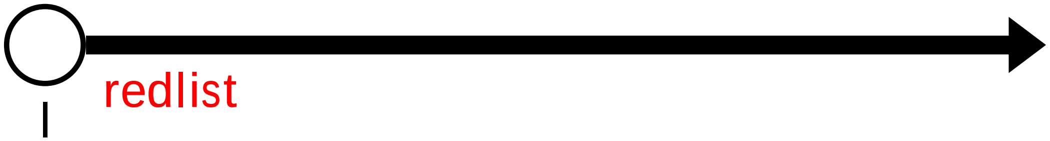
checkers

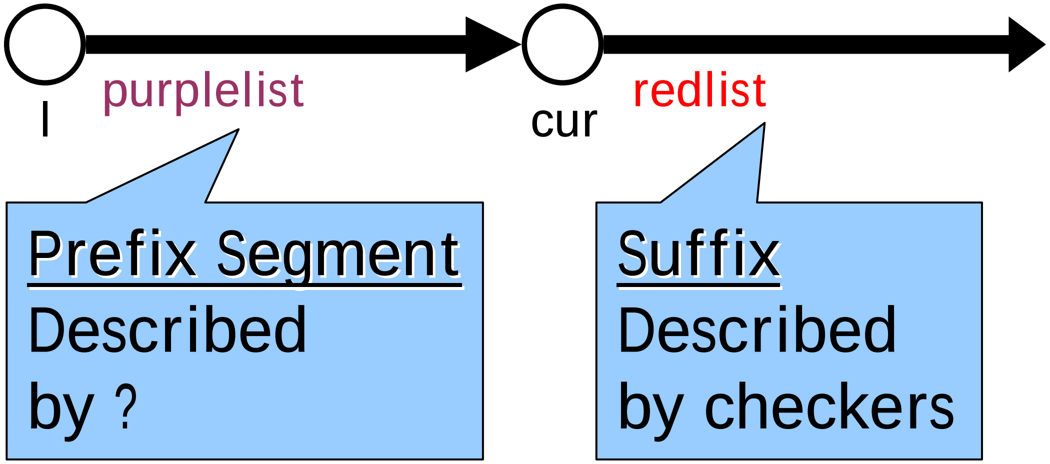


shape analyzer

Challenge: Intermediate invariants

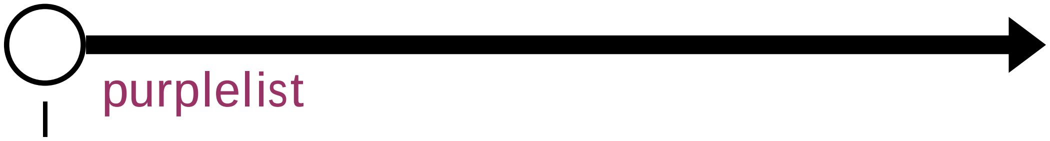
```
assert(redlist(l));
cur = l;
while (cur != null) {
  make_purple(cur);
  cur = cur→next;
}
assert(purplelist(l));
```





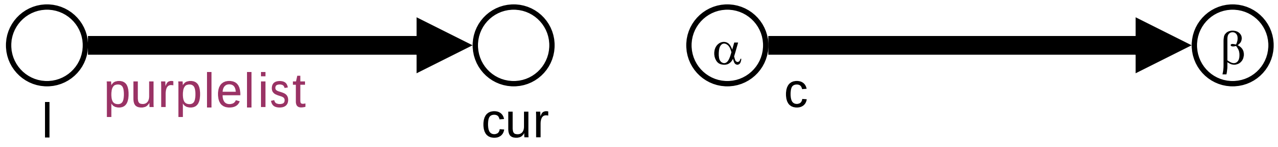
Prefix Segment
Described
by ?

Suffix
Described
by checkers

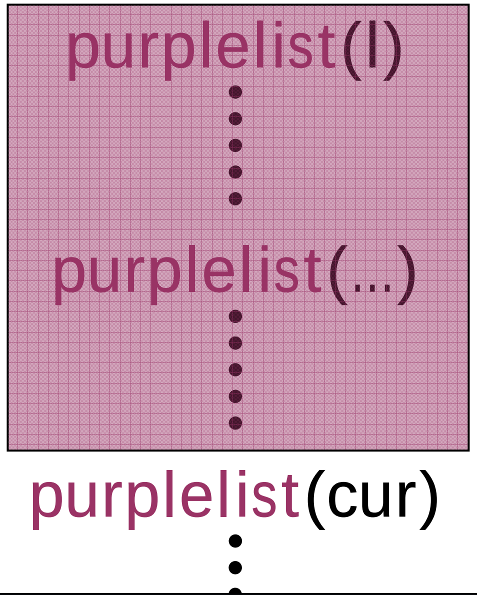


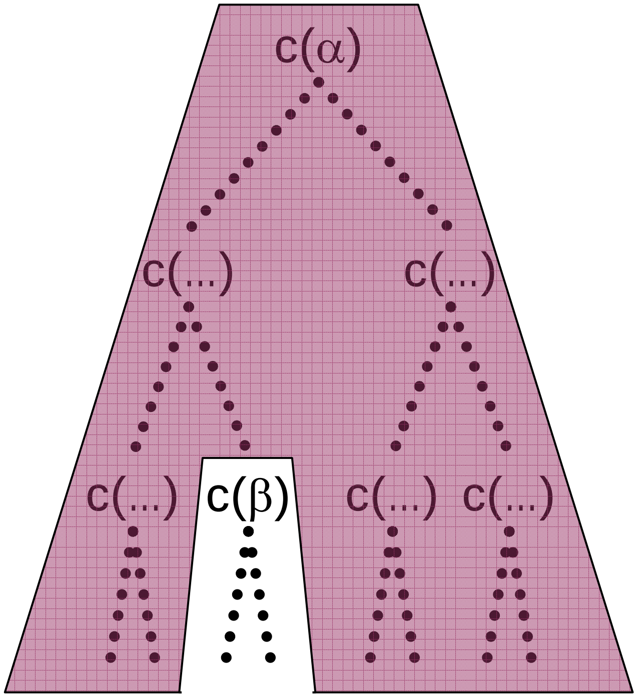
Prefix segments as partial checker runs

Abstraction



Checker Run





Formula

Doesn’t quite work because we need materialization

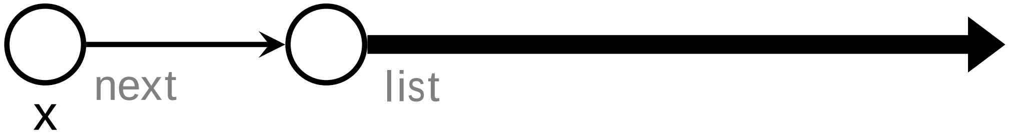
$c(\alpha) \ast\!- c(\beta)$ **?**

Outline

- Memory abstraction
  - Restrictions on checkers
  - Challenge: Intermediate invariants
- **Materialization by forward unfolding**
  - **Where and how**
  - **Challenge: Unfolding segments**
- Materialization by backward unfolding
  - Challenge: Back pointers
- Deciding where to unfold generically

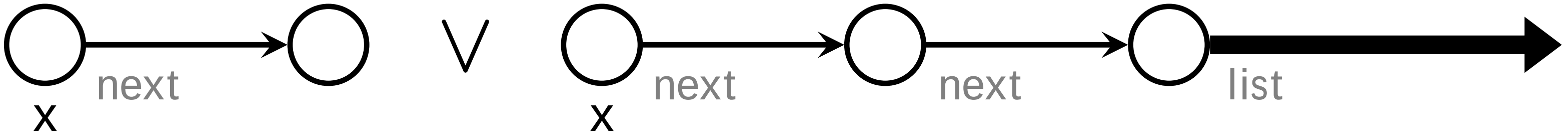
Flow function: Unfold and update edges

```
x→next =
  x→next→next;
```



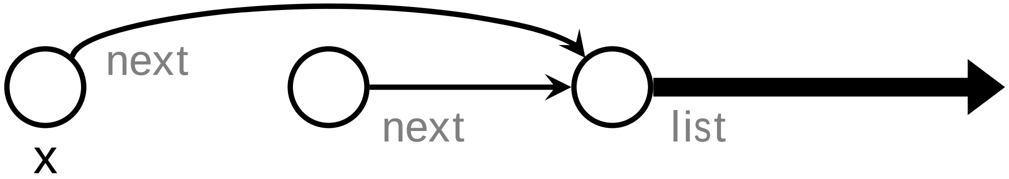
Unfold inductive definition

materialize: x→next, x→next→next



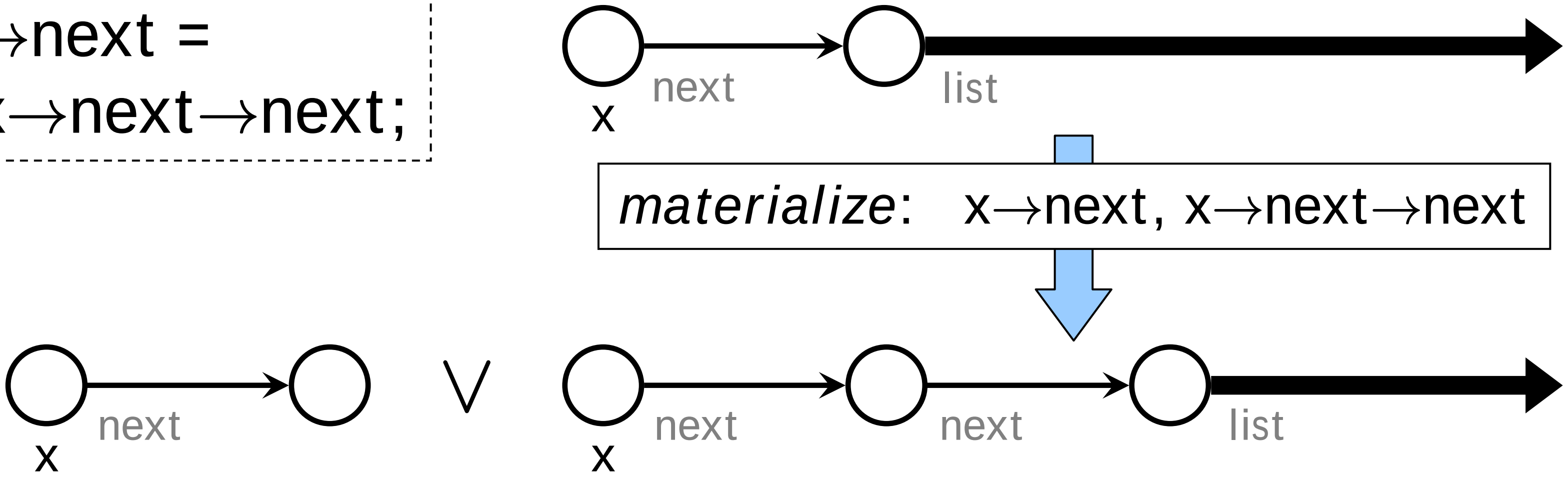
Strong updates using disjointness of regions

update: x→next = x→next→next



Unfolding: where, how, and why ok

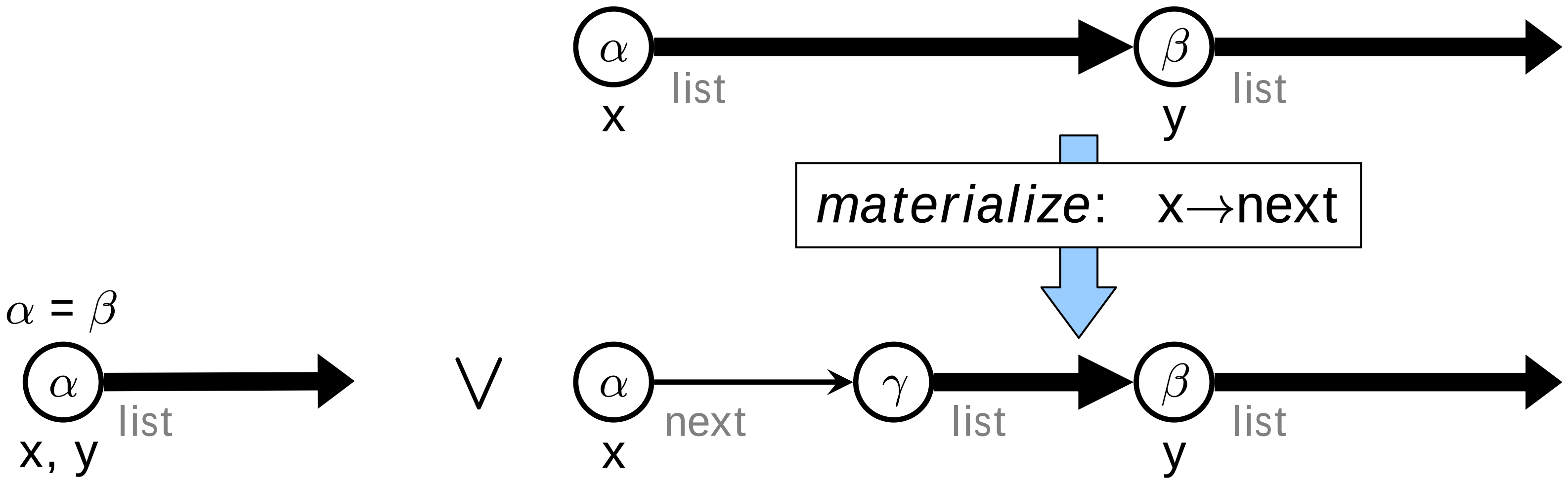
```
x→next =
  x→next→next;
```



materialize: x→next, x→next→next

- Where
  - “Reach” a traversal argument with x→next
- How and Why Ok (concretizations same)
  - By definition

What about unfolding segments?



$$\mathrm{list}(\alpha) \mathrel{\ast\!-} \mathrm{list}(\beta)$$

$$\cancel{\Downarrow}$$

$$\mathrm{emp} \;\lor\; \alpha@\mathrm{f} \mapsto \gamma \ast (\mathrm{list}(\gamma) \mathrel{\ast\!-} \mathrm{list}(\beta))$$

Segment connector (for unfolding)

Concrete

store σ : Val $\rightarrow$ Val

valuation ν : SymVal $\rightarrow$ Val

“unfolded” points-to

“folded” recursive calls

pure formula

t t

c(α) ::=

... $\vee$ ($M^{\mathsf{u}} * M^{\mathsf{f}} \wedge F$) $\vee$ ...

$\sigma, \nu \vDash$ c(α) $*$= c′(α')

iff there exists an i such that c(α) $*{=}^{i}$ c′(α')

$[\cdot], \nu \vDash$ c(α) $*{=}^{0}$ c(α')

iff $\nu(\alpha) = \nu(\alpha')$

$\sigma, \nu \vDash$ c(α) $*{=}^{i+1}$ c′(α')

iff there exists a disjunct ($M^{\mathsf{u}} * M^{\mathsf{f}} *$ c″(β) $\wedge F$) such that

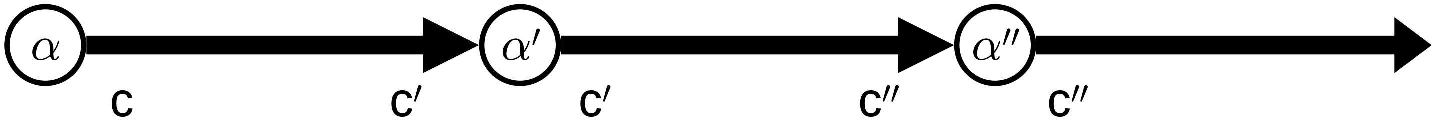
ν satisfies [actuals/formals]F and

$\sigma, \nu \vDash$ [actuals/formals]($M^{\mathsf{u}} * M^{\mathsf{f}} *$ c″(β) $*{=}^{i}$ c′(α'))

Basic properties of segments

- If $\sigma, \nu \models \mathsf{c}(\alpha) \ast\!= \mathsf{c}'(\alpha')$, then $\sigma, \nu \models \mathsf{c}(\alpha) \ast\!- \mathsf{c}'(\alpha')$
  - If $\sigma, \nu \models (\mathsf{c}(\alpha) \ast\!= \mathsf{c}'(\alpha')) \ast \mathsf{c}'(\alpha')$, then $\sigma, \nu \models \mathsf{c}(\alpha)$ (elimination)

- $[\cdot], \nu \models \mathsf{c}(\alpha) \ast\!= \mathsf{c}(\alpha)$ (reflexivity)
- If $\sigma, \nu \models (\mathsf{c}(\alpha) \ast\!= \mathsf{c}'(\alpha')) \ast (\mathsf{c}'(\alpha') \ast\!= \mathsf{c}''(\alpha''))$, then $\sigma, \nu \models \mathsf{c}(\alpha) \ast\!= \mathsf{c}''(\alpha'')$ (transitivity)



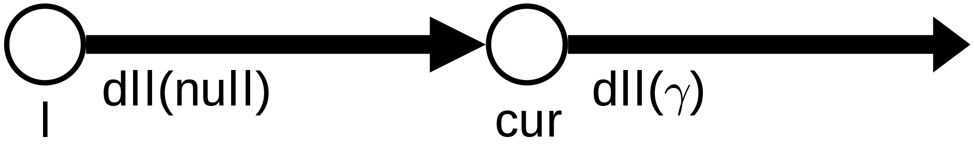
Outline

- Memory abstraction
  - Restrictions on checkers
  - Challenge: Intermediate invariants
- Materialization by forward unfolding
  - Where and how
  - Challenge: Unfolding segments
- **Materialization by backward unfolding**
  - **Challenge: Back pointers**
- Deciding where to unfold generically

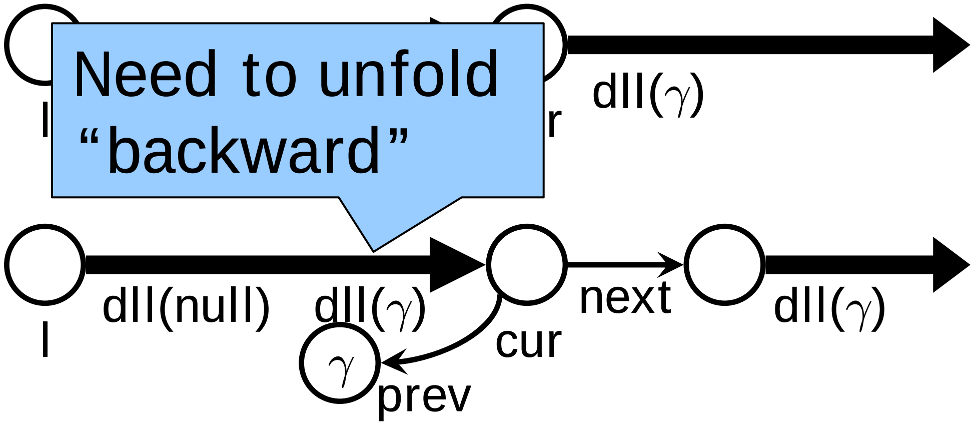
Challenge: Back pointers

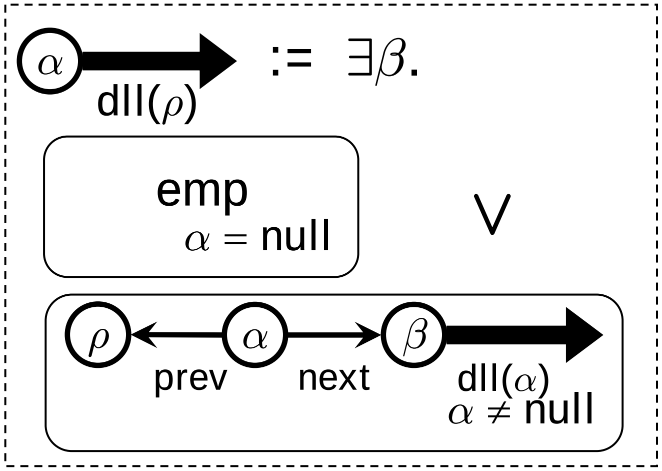
Example: Removal in doubly-linked lists

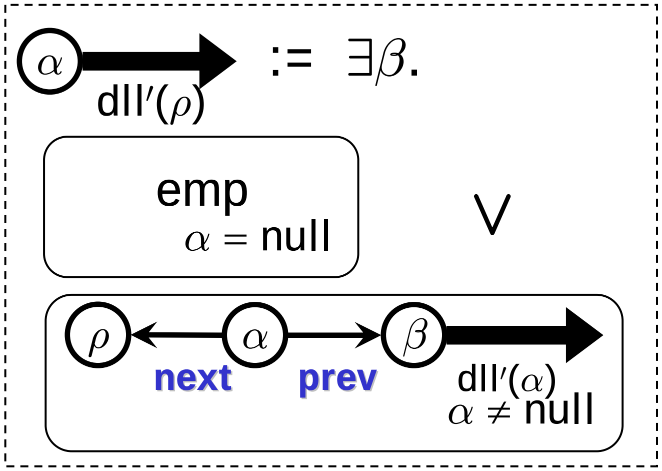
- Traversal on ‘next’ field to find element to remove:



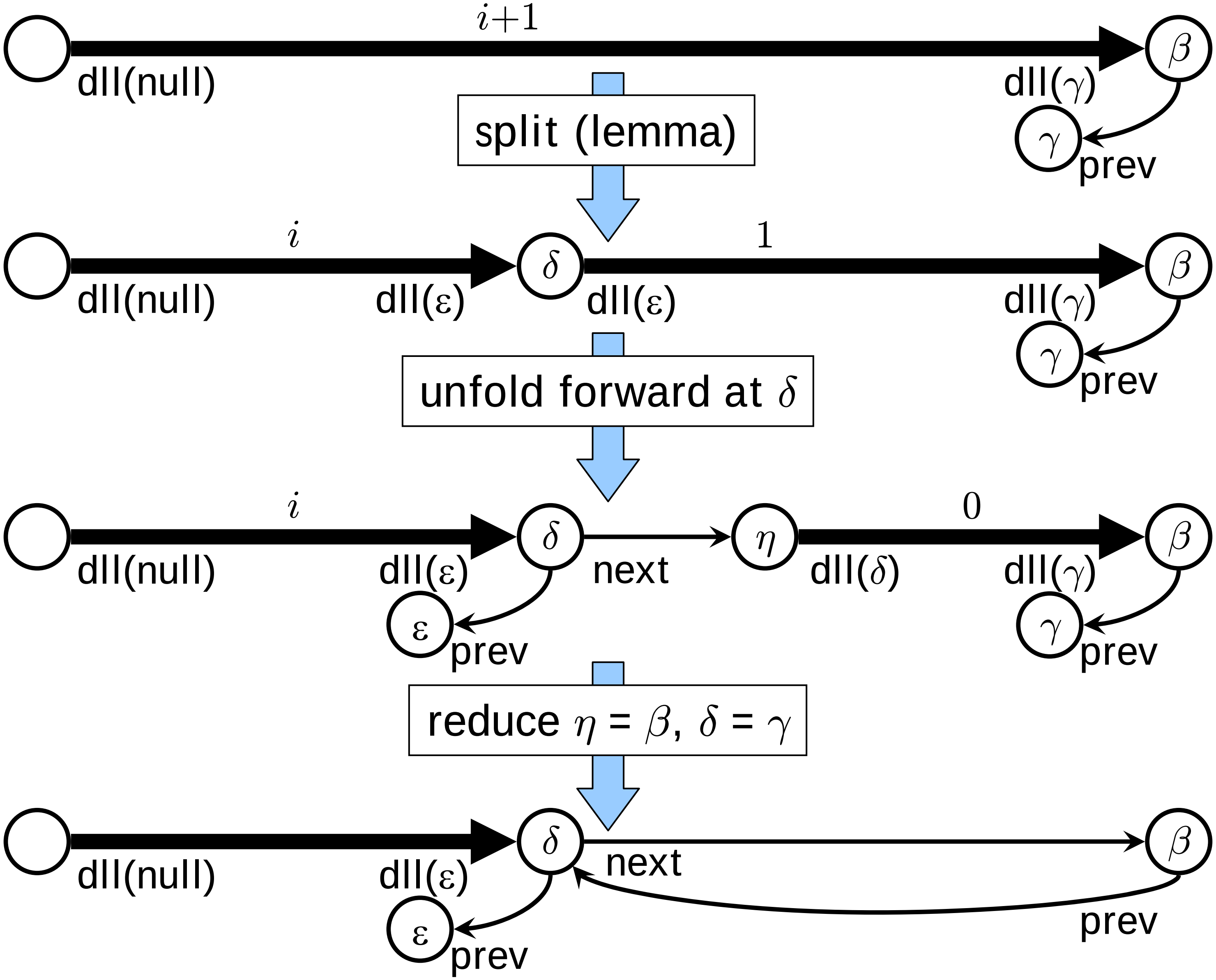
- Materialize ‘cur→prev’ and remove ‘cur’:


Need to unfold “backward”





Backwards unfolding by forwards unfolding



Outline

- Memory abstraction
  - Restrictions on checkers
  - Challenge: Intermediate invariants
- Materialization by forward unfolding
  - Where and how
  - Challenge: Unfolding segments
- Materialization by backward unfolding
  - Challenge: Back pointers
- **Deciding where to unfold generically**

Deciding where to unfold

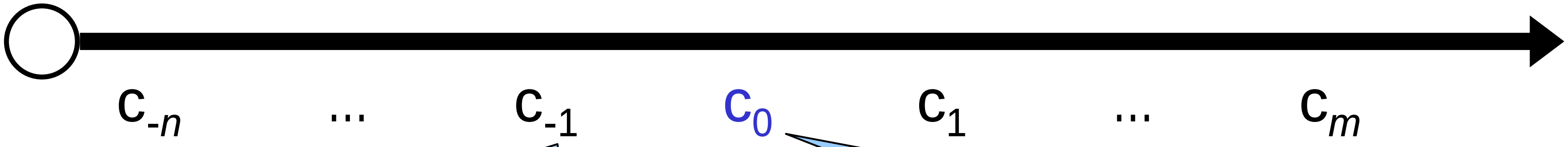
- Observations: Ca n fields are materi k

A pointer that may materialize these fields

Where in the traversal it may be materialized

types	τ	::=	$\{\ f_1\langle l_1\rangle,\ \ldots,\ f_n\langle l_n\rangle\ \}$
levels	l	::=	n | unk

- Levels

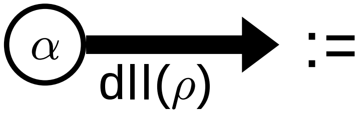


Level -1: Materialized just before this call

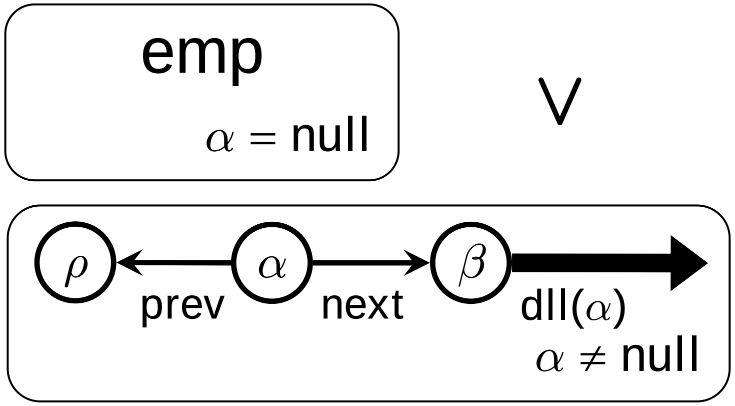
Level 0: Materialized in this call.

Example: Doubly-linked lists

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



$\exists$(β : {next$\langle 1 \rangle$, prev$\langle 1 \rangle$}).

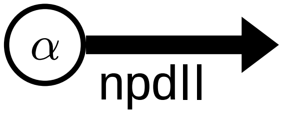


Before:
Traversal argument had level 0 fields (implicitly)

Backward unfolding parameter ρ has level -1

Example: Alternative doubly-linked list

α : {next⟨0⟩, prev⟨-1⟩}



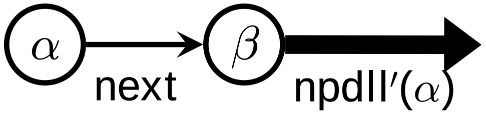
:=

$\exists(\beta$: {next⟨2⟩, prev⟨1⟩}).

emp

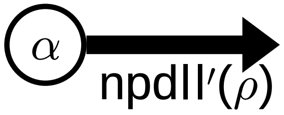
α = null

$\vee$



$\alpha \neq$ null

α : {next⟨1⟩, prev⟨0⟩},
ρ : {next⟨-1⟩, prev⟨-2⟩}



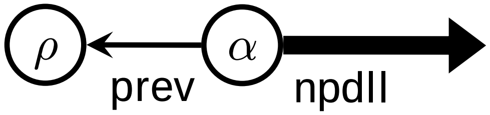
:=

$\exists(\beta$: {next⟨1⟩, prev⟨1⟩}).

emp

α = null

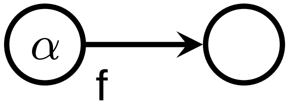
$\vee$



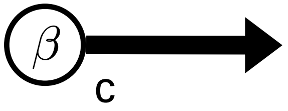
$\alpha \neq$ null

Types can be inferred automatically

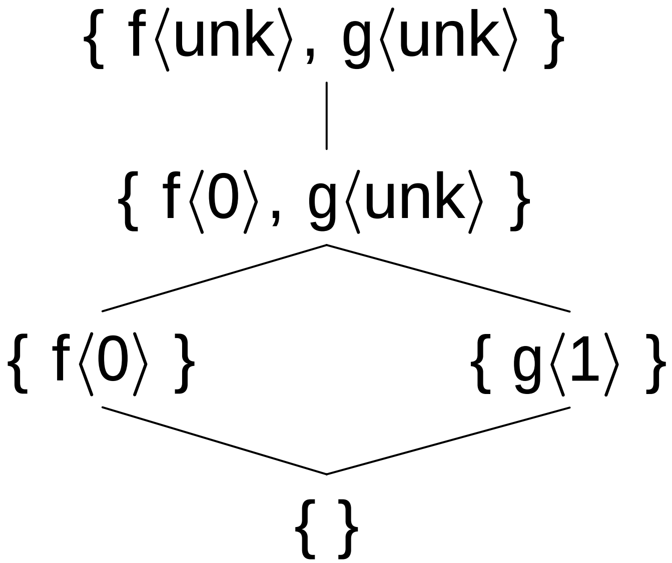
Checking



$\{ f\langle 0\rangle \}$ <: typeof(α)



typeof(β) – 1
<: declared_typeof(π)
(where c(π) := ...)



$\{ f\langle \text{unk}\rangle, g\langle \text{unk}\rangle \}$

$\{ f\langle 0\rangle, g\langle \text{unk}\rangle \}$

$\{ f\langle 0\rangle \}$ $\{ g\langle 1\rangle \}$

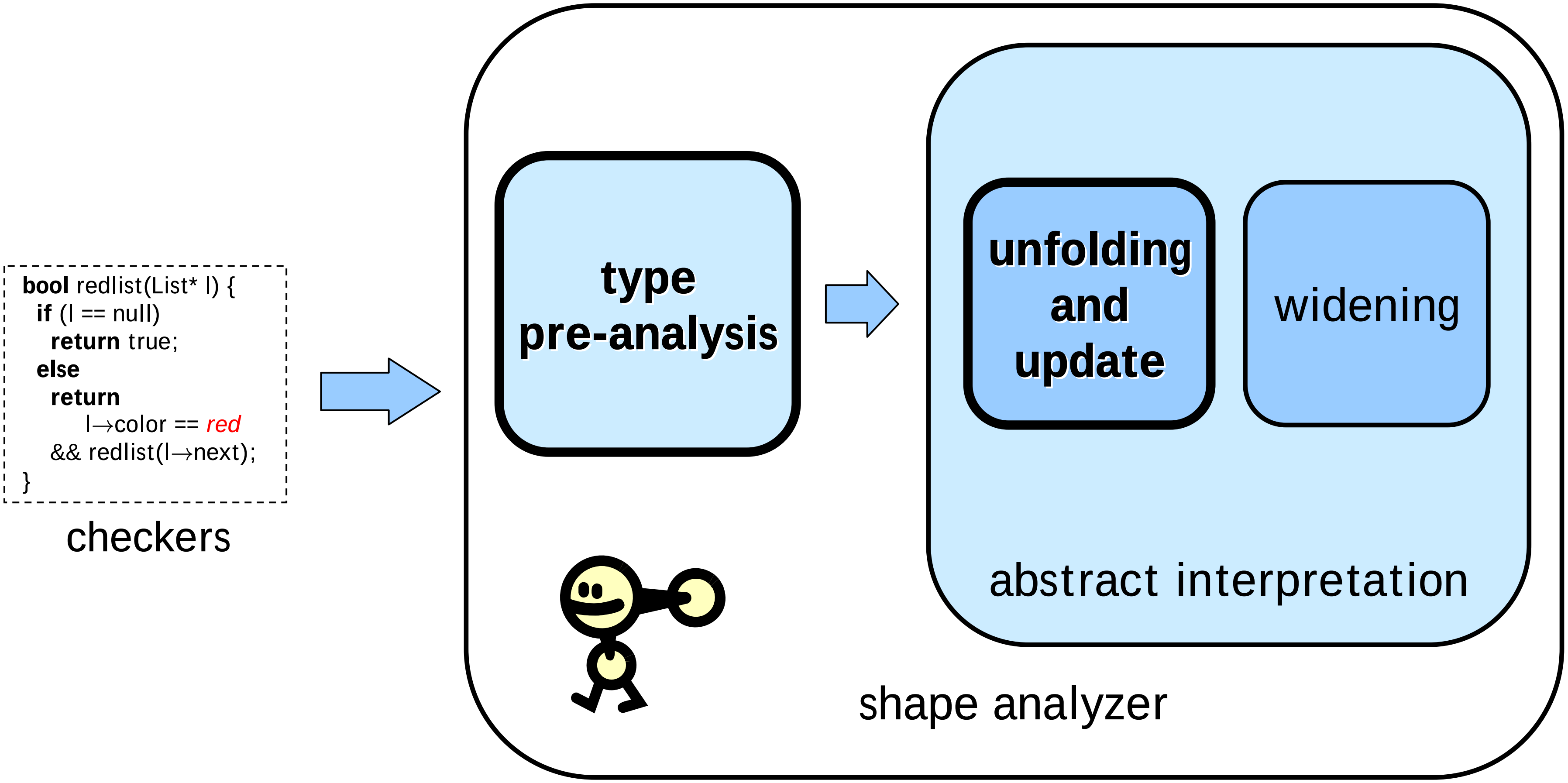
$\{ \}$

Inference using a fixed-point computation with types initialized to { }

Summary:
Enabling materialization anywhere

- Defined segments as partial checker runs directly (inductively)
  - For forward unfolding
  - Backward unfolding derived from forward unfolding
- Checker parameter types with levels
  - For deciding where to unfold
  - Inferable and does not affect soundness

Summary: Given checkers, everything is automatic



```
bool redlist(List* l) {
  if (l == null)
    return true;
  else
    return
        l→color == red
      && redlist(l→next);
}
```

checkers

type pre-analysis

unfolding and update

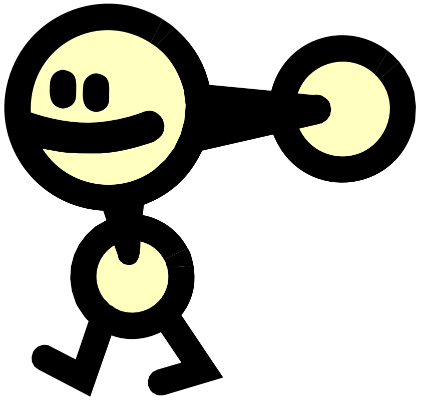
widening

abstract interpretation

shape analyzer

Conclusion

- Invariant checkers can form the basis of a memory abstraction that
  - Is easily extensible on a per-program basis
  - Expresses developer intent
    - Critical for usability
    - Prerequisite for scalability
- Enabling materialization anywhere
  - Inductive segments
  - Pre-analysis on checkers to decide where to unfold robustly

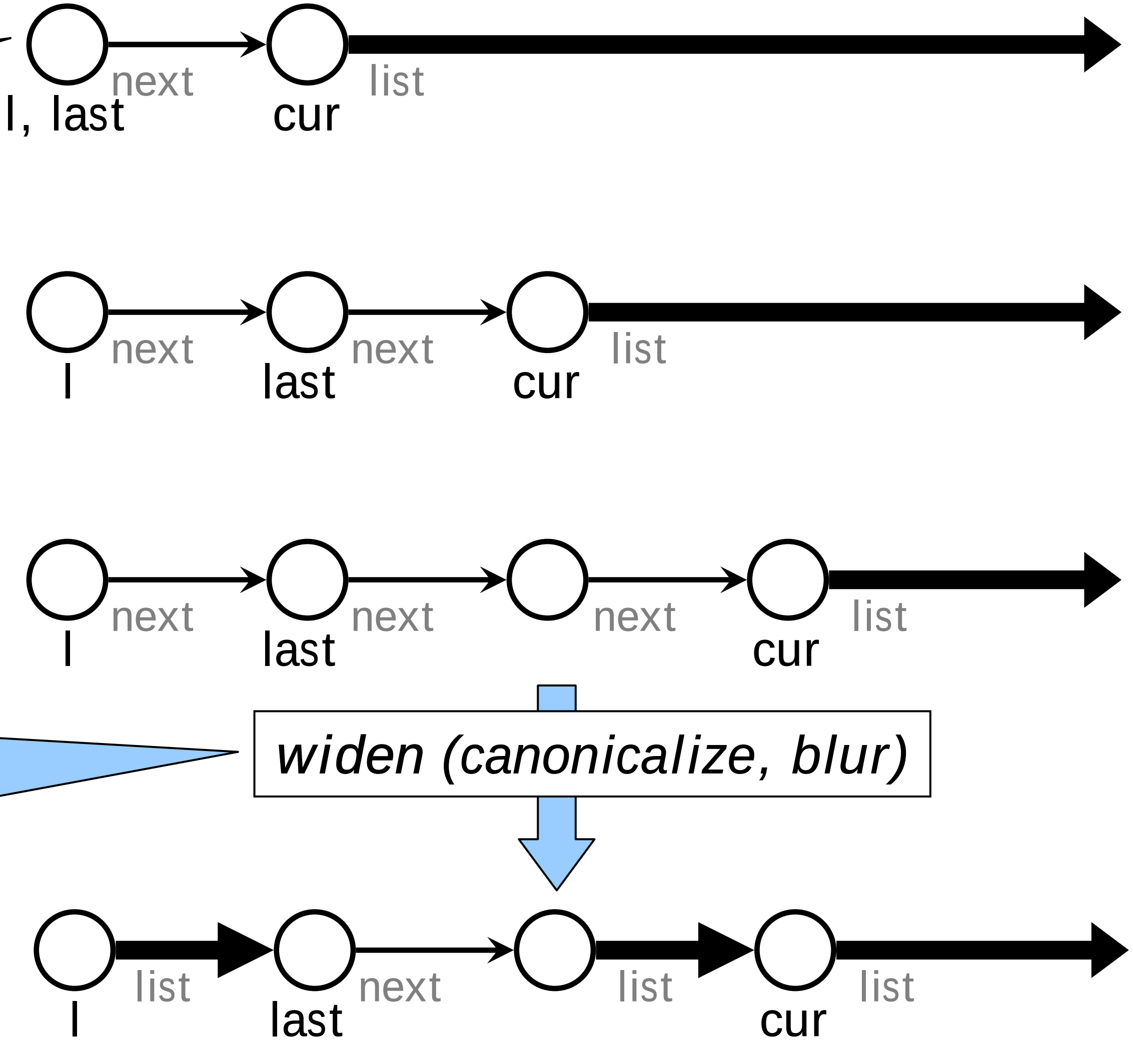


What can checker-based shape analysis do for you?

Challenge: Termination and precision

```
last = l;
c
w
   if (...) last = cur;
   cur = cur→ next;
}
```

Observation
Previous iterates are “less unfolded”



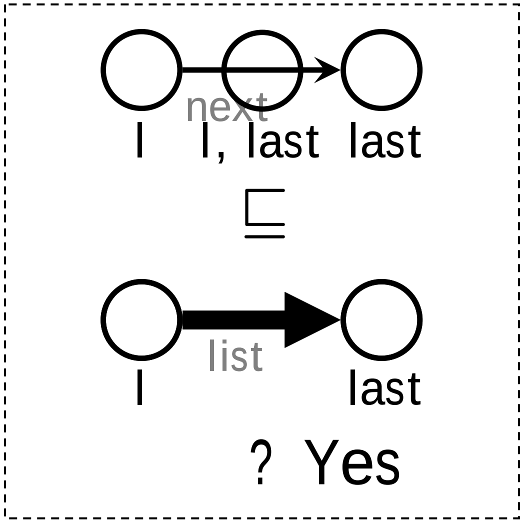
Fold into checker edges

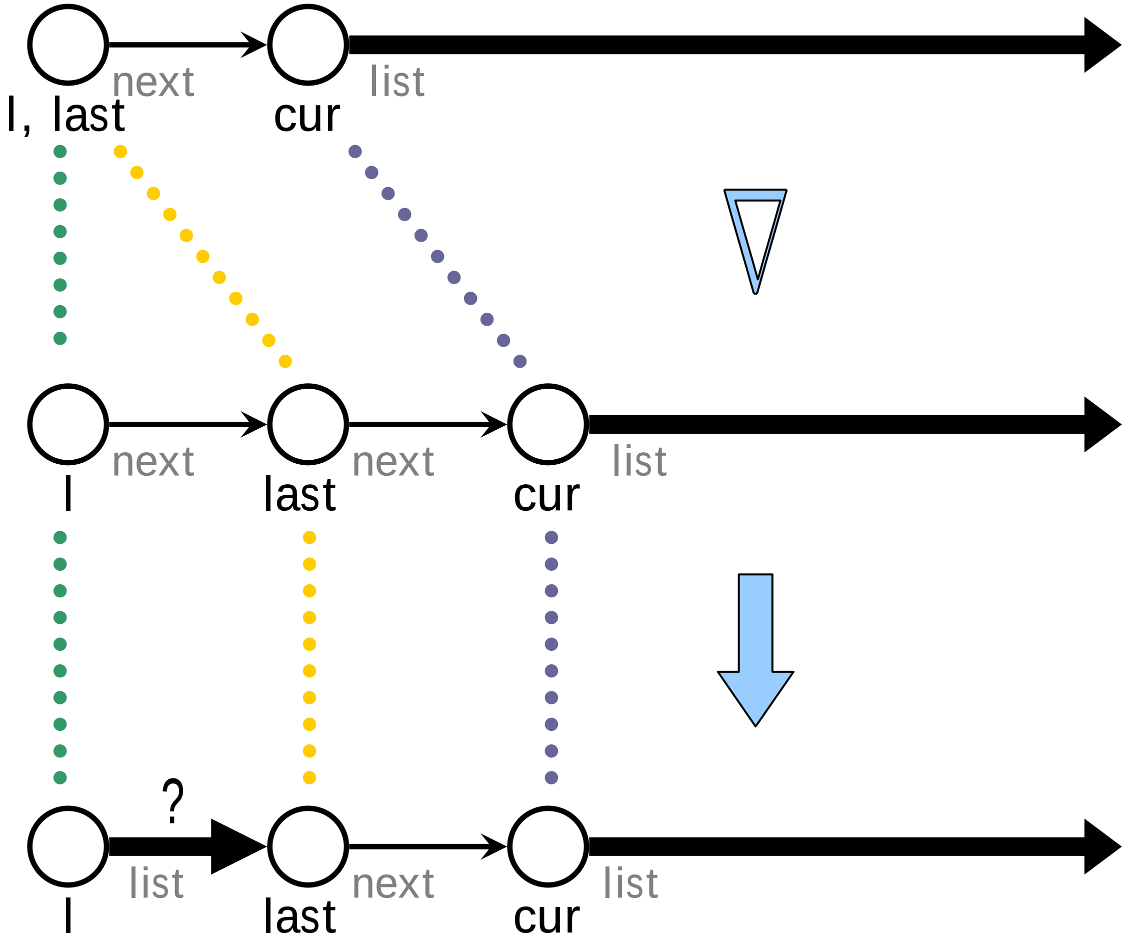
But where and how much?

History-guided folding

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

- Match edges to identify where to fold
- Apply local folding rules

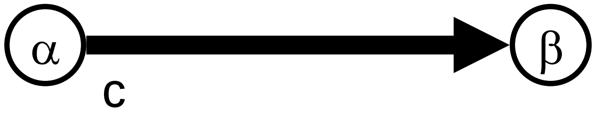




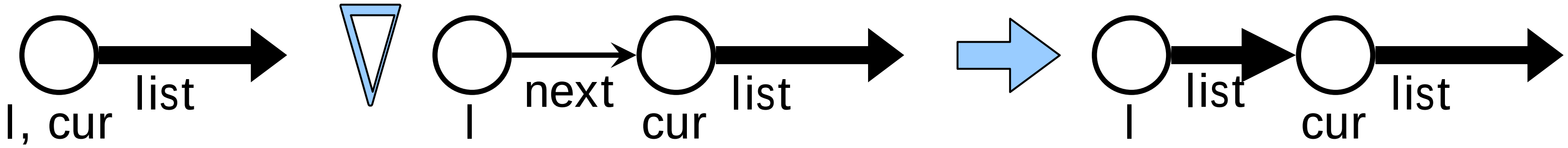
Summary:
Enabling checker-based shape analysis

- Built-in disjointness of memory regions
  - As in separation logic
  - Checkers read any object field only once in a run

- Generalized segment abstraction
  - Based on partial checker runs



- Generalized folding into inductive predicates
  - Based on iteration history (i.e., a widening operator)



Experimental results

Benchmark	Lines of Code	Analysis Time	Max. Num. Graphs at a Program Point	Max. Num Iterations at a Program Point
list reverse	19	0.007s	1	3
list remove element	27	0.016s	4	6
list insertion sort	56	0.021s	4	7
search tree find	23	0.010s	2	4
skip list rebalance	33	0.087s	6	7
`scull` driver	894	9.710s	4	16

- Verified structural invariants as given by checkers are preserved across data structure manipulation
- Limitations (in `scull` driver)
  - Arrays not handled (rewrote as linked list), char arrays ignored
- Promising as far as number of disjuncts