Shape Analysis with Structural Invariant Checkers

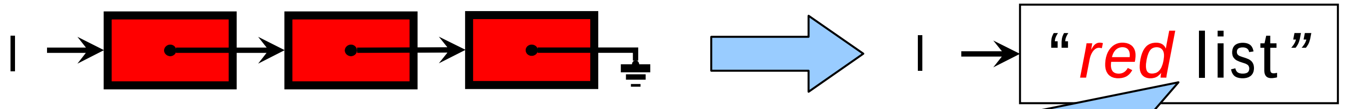
Bor-Yuh Evan Chang
Xavier Rival
George C. Necula

University of California, Berkeley

SAS 2007

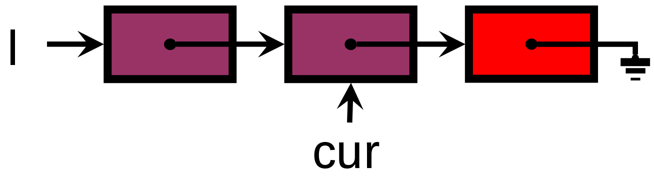
Example: Typestate with shape analysis

Concrete Example | **Abstraction**



program-specific predicate

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



```
   cur = cur→next;
}
```

heap abstraction flow-sensitive

make_purple(·) could be

- lock(·)
- free(·)
- open(·)
- …

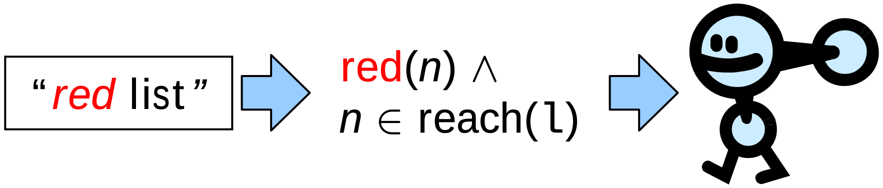
Shape analysis is not yet practical

Usability: Choosing the heap abstraction difficult

Space Invader
[Distefano et al.]

Built-in high-level predicates

- - Hard to extend
- + No additional user effort

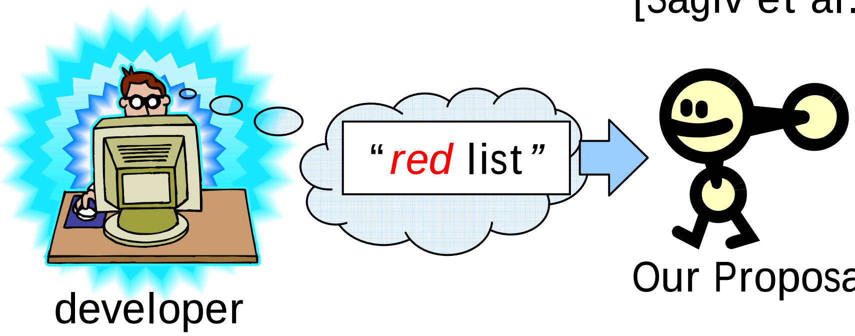


$\mathrm{red}(n) \land n \in \mathrm{reach}(l)$

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

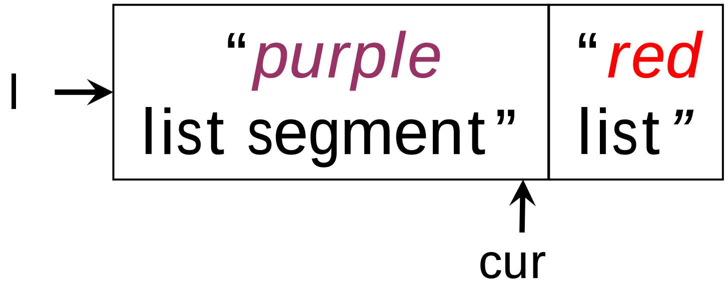
- + Extensible
- + Easier for developers

Shape analysis is not yet practical

Scalability: Finding right level of abstraction difficult

➡ Over-reliance on disjunction for precision

developer

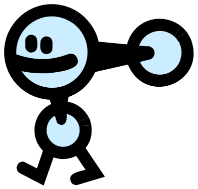


l

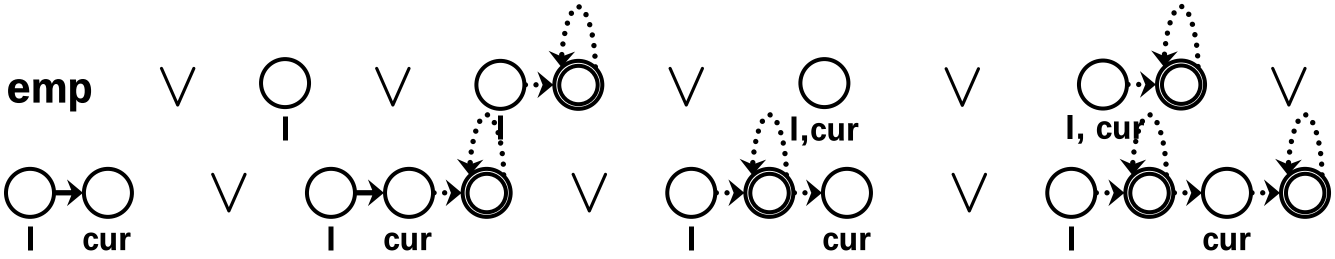
“*purple* list segment”

“*red* list”

cur



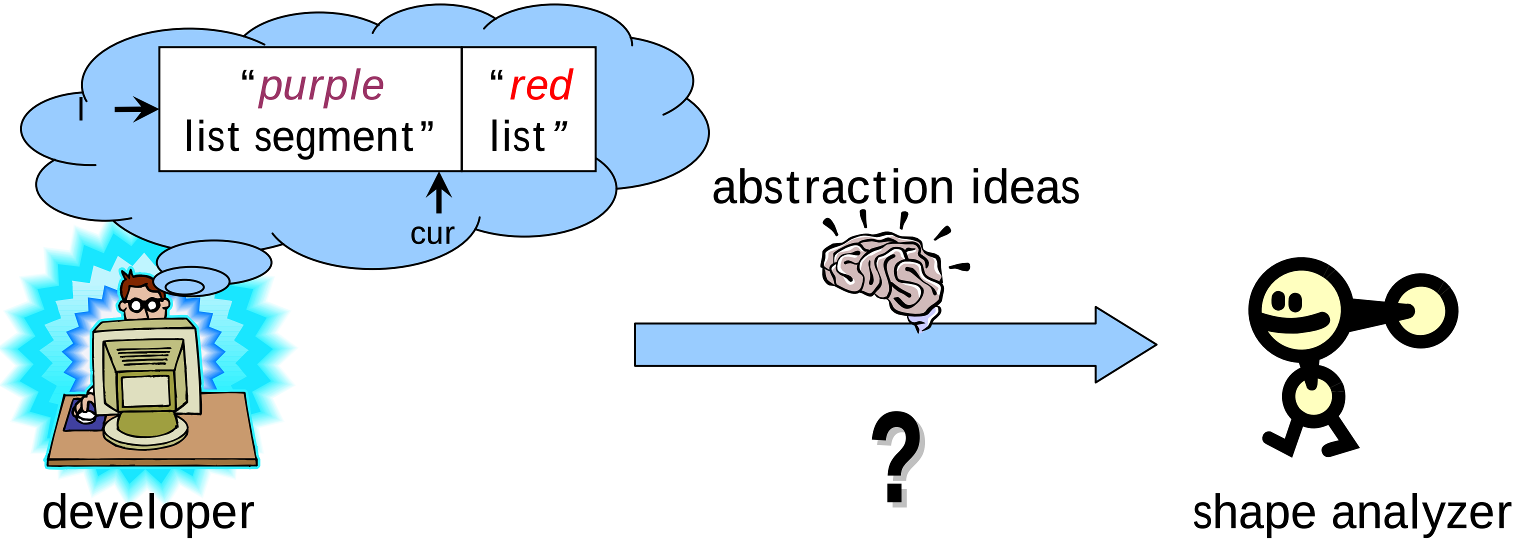
shape analyzer



Hypothesis

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

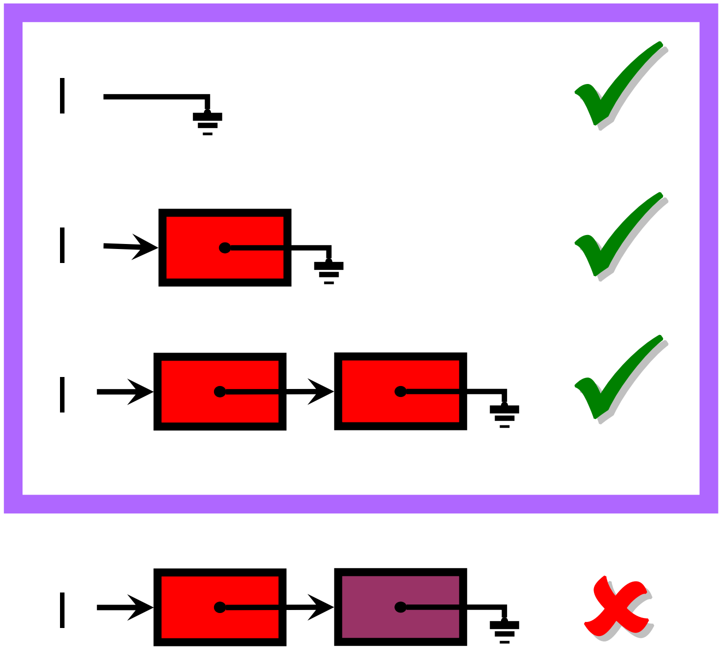
- Good abstraction is program-specific



Observation

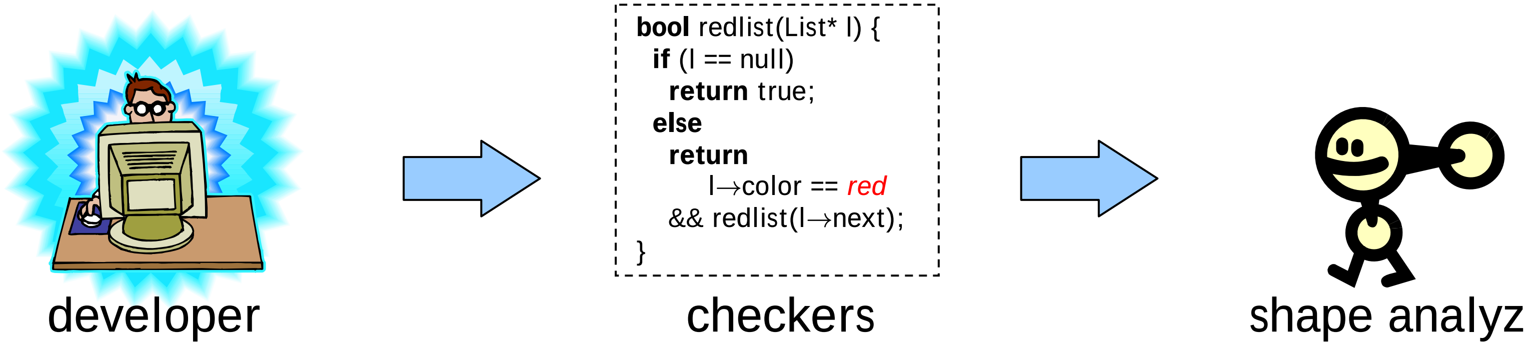
Checking code expresses a shape invariant and an intended usage pattern.

```
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 based on invariant checkers.



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

- Extensible
  - Abstraction based on the developer-supplied checkers
- Targeted for Usability
  - Code-like global specification, local invariant inference
- Targeted for Scalability
  - Based on the hypothesis

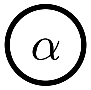
Outline

- **Memory abstraction**
  - **Restrictions on checkers**
  - **Challenge: Intermediate invariants**
- Analysis algorithm
  - Strong updates
  - Challenge: Ensuring termination
- Experimental results

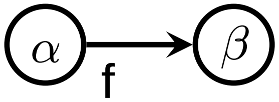
Abstract memory using checkers

“Some number of points-to edges that satisfies checker c”

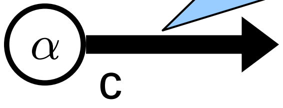
Graphs



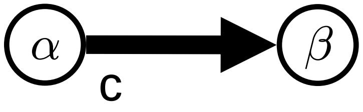
values (address or null)



points-to relation (memory cell)



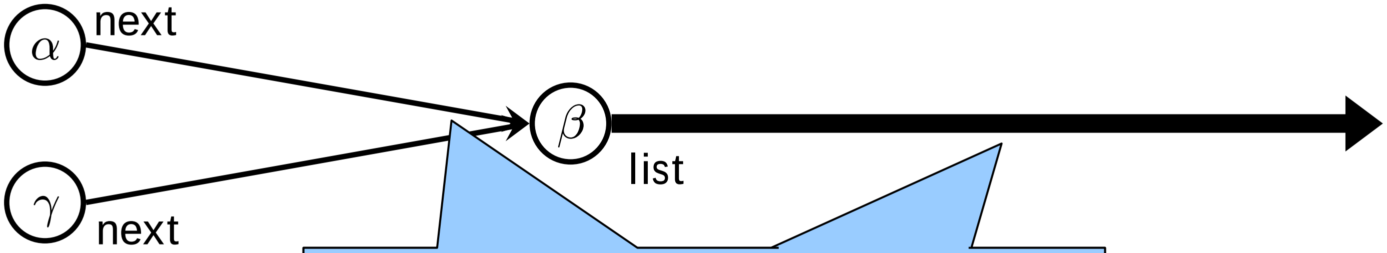
checker run



partial run

Example

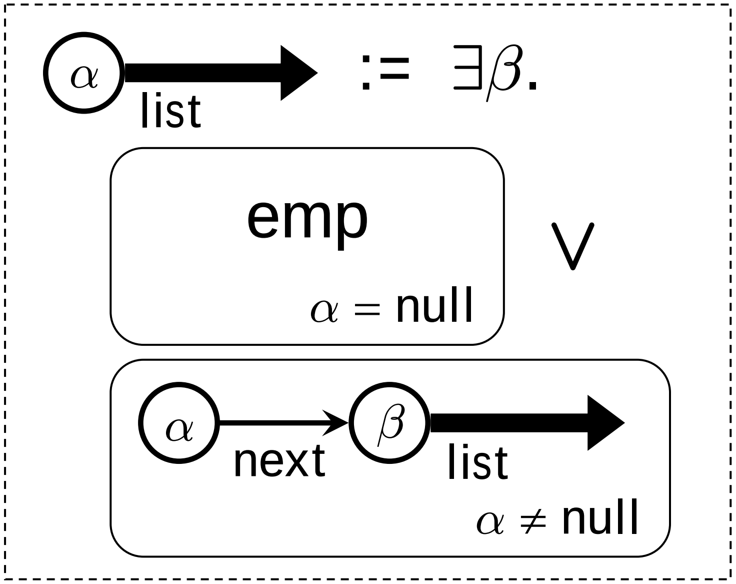
“Disjointly, $\alpha \to$next = β, $\gamma \to$next = β, and β is a list.”



disjoint memory regions

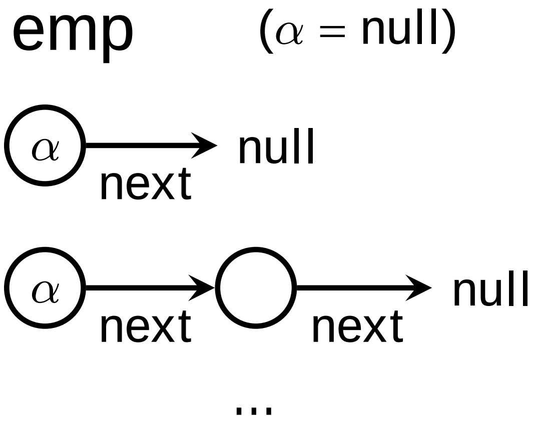
Checkers as inductive definitions

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



Disjointness

Checker run can dereference any object field only once



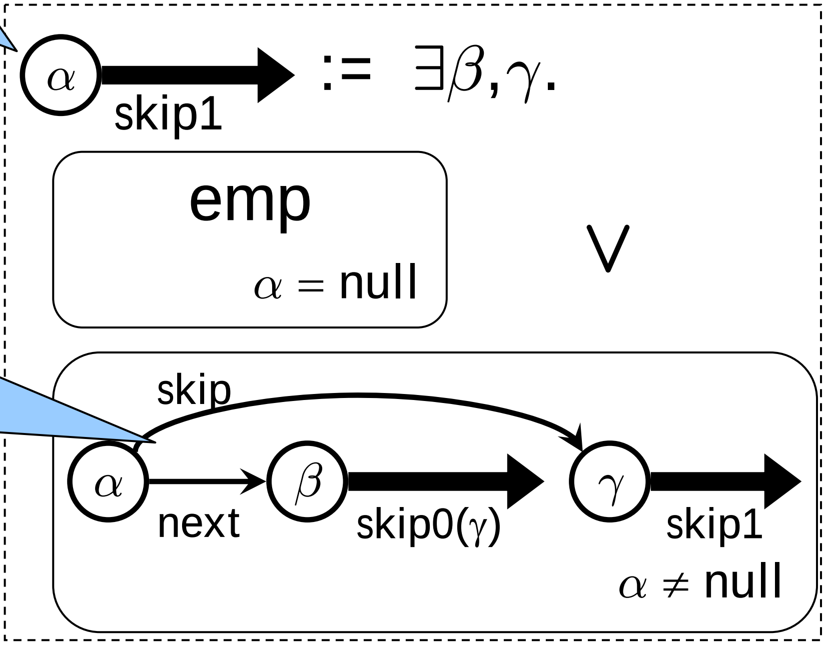
What can a checker do?

- In this talk, a *checker* …
  - is a pure, recursive function
  - dereferences any object field only once during a run
  - only one argum dereferenced (traversal arg)

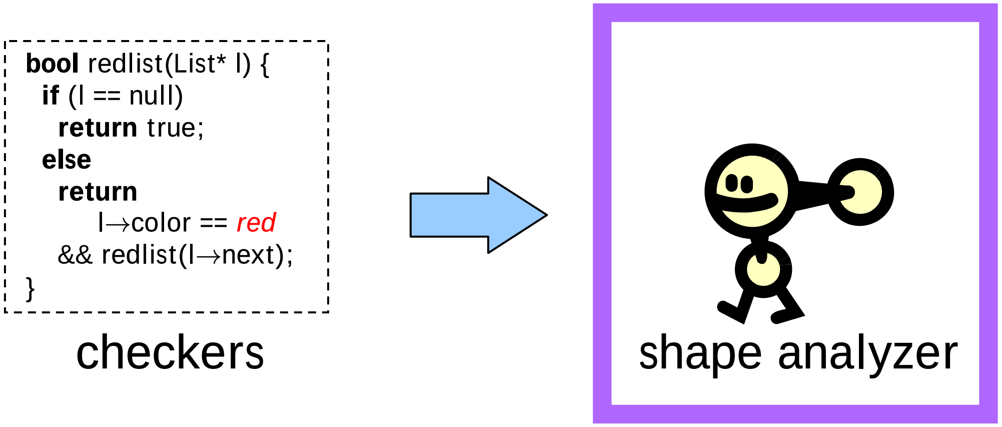
Traversal argument

```
bool skip1(Skip* l) {
  if (l == null) return true;
  else {
    Skip* s = l→s
    return skip0
        && skip1(s); }
}
```

Only fields from traversal argument



back to the abstract domain ...



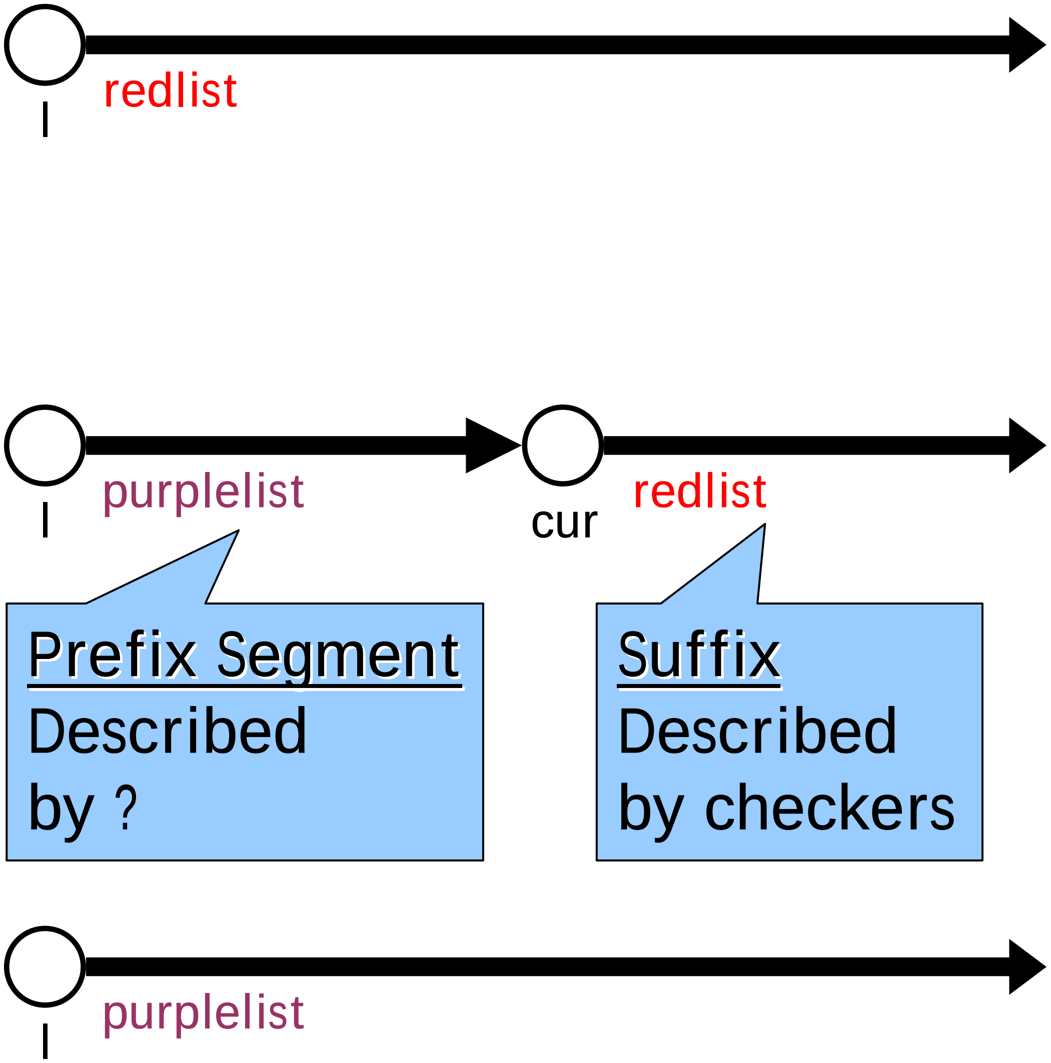
```
bool redlist(List* l) {
  if (l == null)
    return true;
  else
    return
        l→color == red
    && redlist(l→next);
}
```

checkers

shape analyzer

Challenge: Intermediate invariants

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

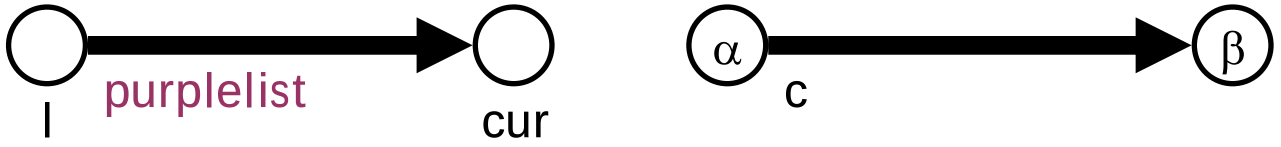


Prefix Segment
Described
by ?

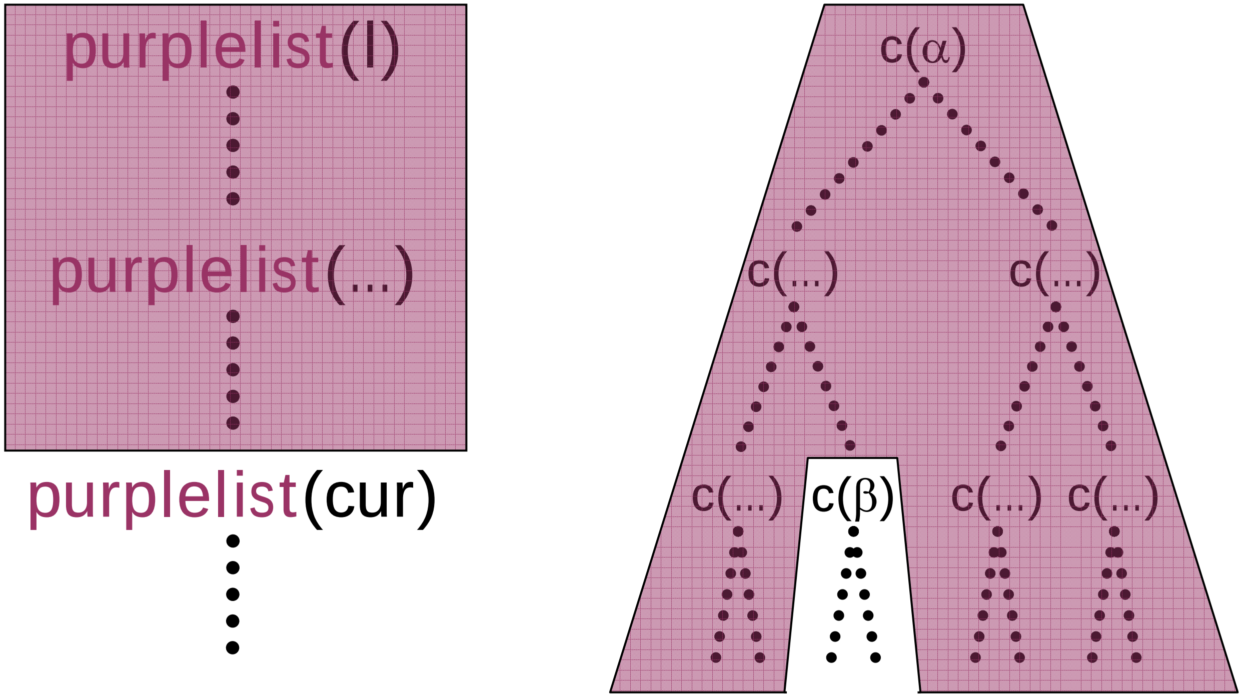
Suffix
Described
by checkers

Prefix segments as partial checker runs

Abstraction



Checker Run

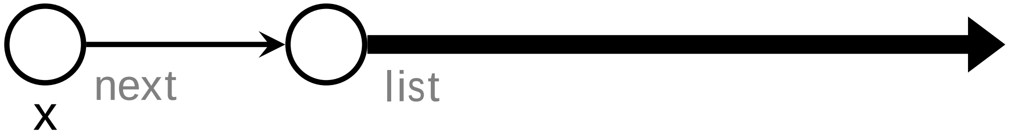


Outline

- Memory abstraction
  - Restrictions on checkers
  - Challenge: Intermediate invariants
- **Analysis algorithm**
  - **Strong updates**
  - **Challenge: Ensuring termination**
- Experimental results

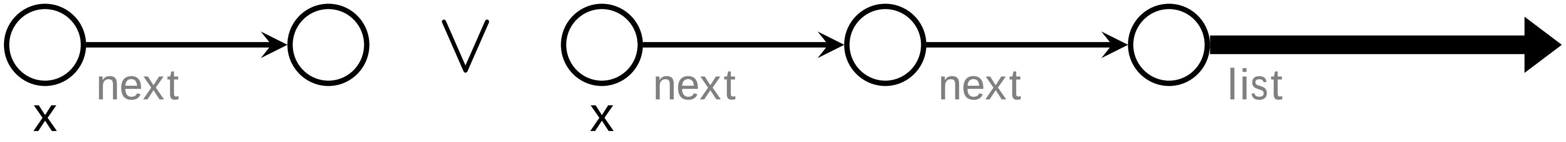
Flow function: Unfold and update edges

```
x→next =
  x→next→next;
```



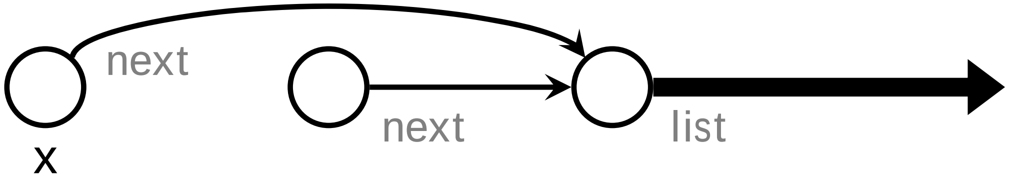
Unfold inductive definition

materialize: x→next, x→next→next



Strong updates using disjointness of regions

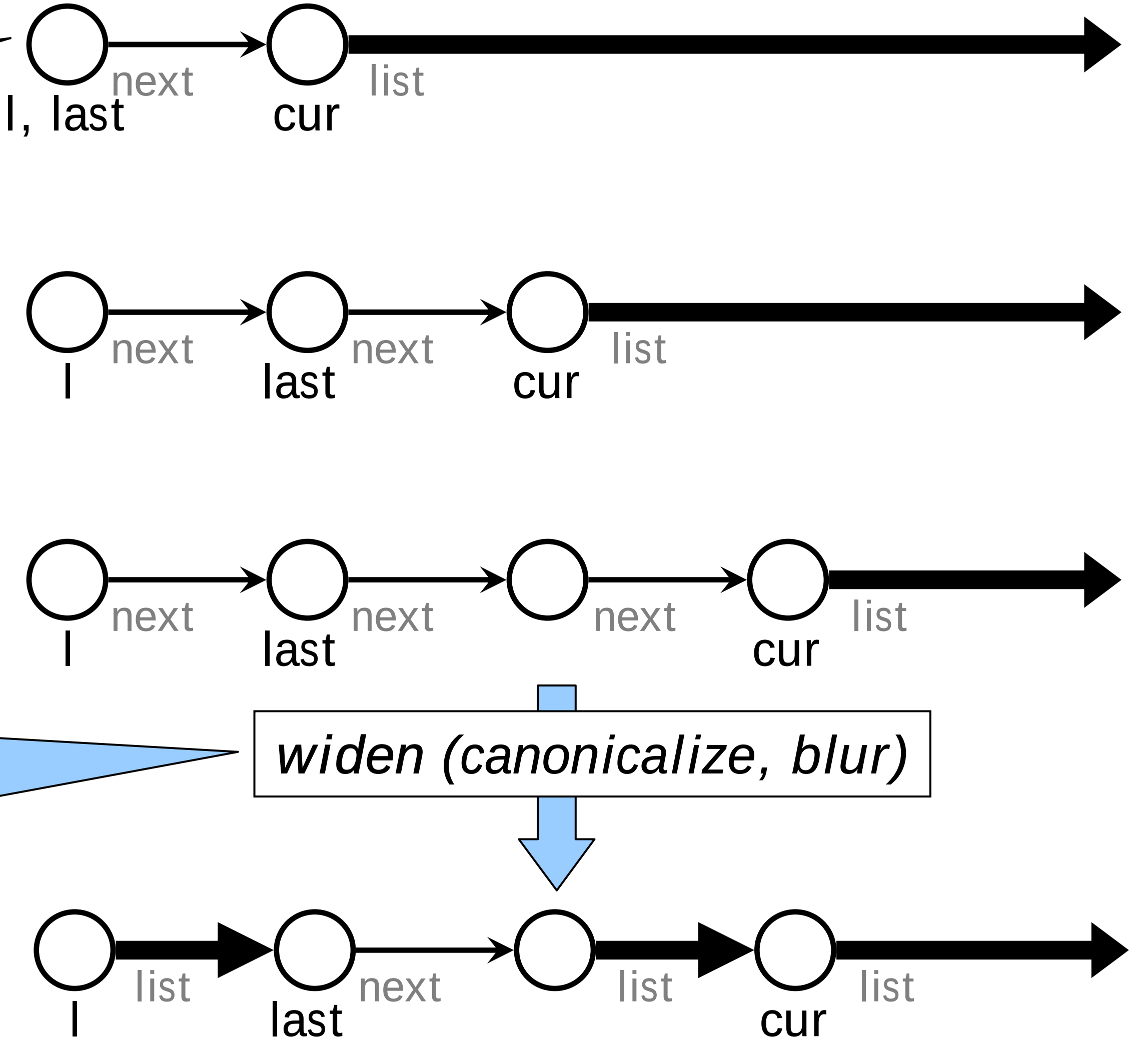
update: x→next = x→next→next



Challenge: Termination and precision

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

Observation
Previous iterates are “less unfolded”



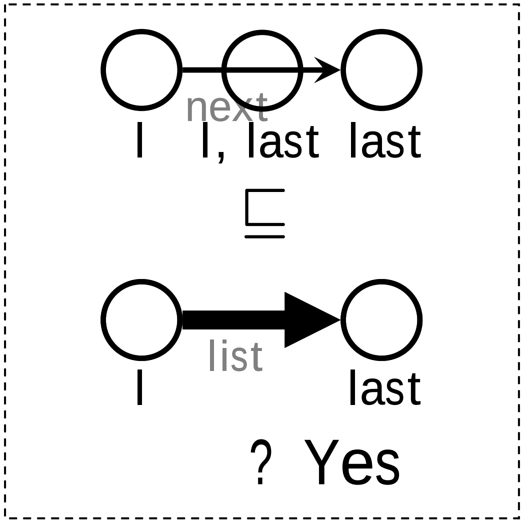
Fold into checker edges

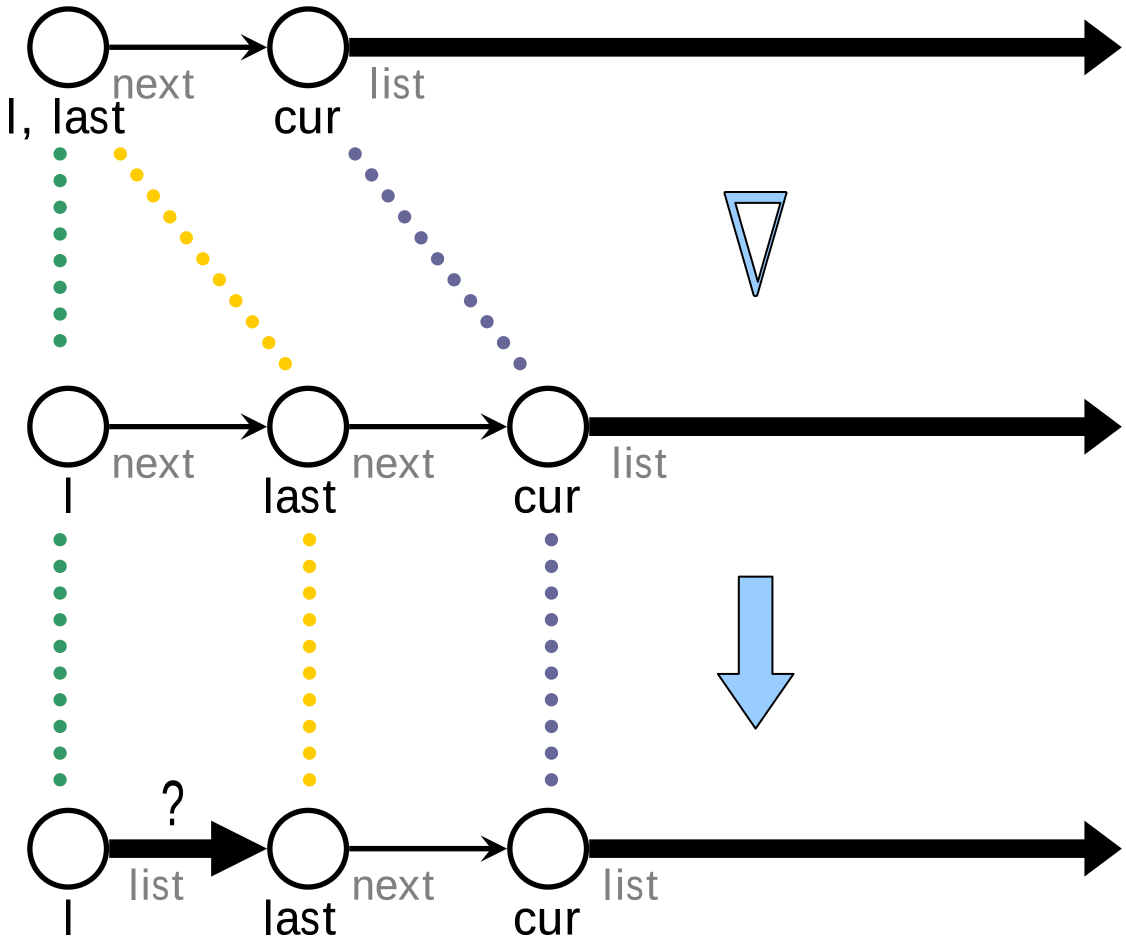
But where and how much?

History-guided folding

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

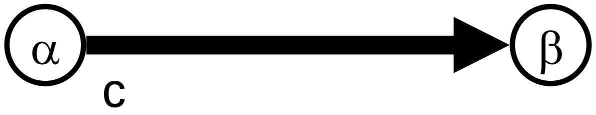
- Match edges to identify where to fold
- Apply local folding rules



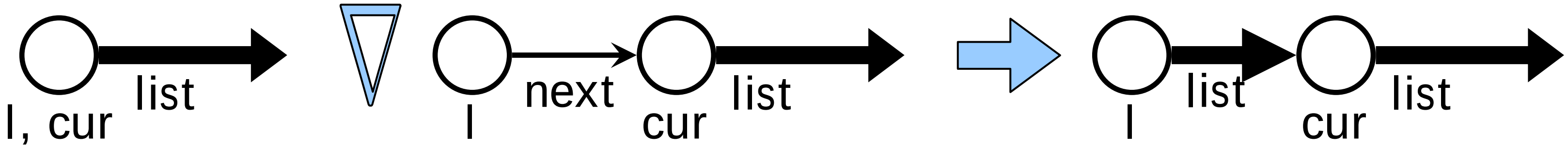


Summary:
Enabling checker-based shape analysis

- 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)



Outline

- Memory abstraction
  - Restrictions on checkers
  - Challenge: Intermediate invariants
- Analysis algorithm
  - Strong updates
  - Challenge: Ensuring termination
- **Experimental results**

Experimental results

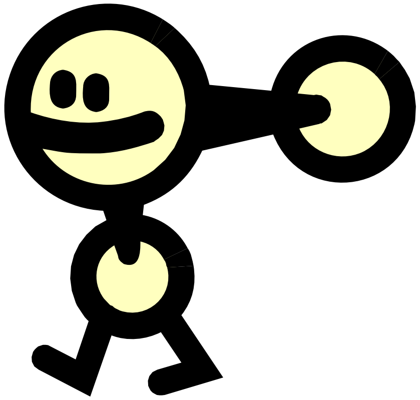
Benchmark	Lines of Code	Analysis Time	Max. Num. Graphs at a Program Point	Max. Num Iterations at a Program Point
list reverse	19	0.007s	1	3
list remove element	27	0.016s	4	6
list insertion sort	56	0.021s	4	7
search tree find	23	0.010s	2	4
skip list rebalance	33	0.087s	6	7
`scull` driver	894	9.710s	4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

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

- Start with usability
- Work towards expressivity



What can checker-based shape analysis do for you?