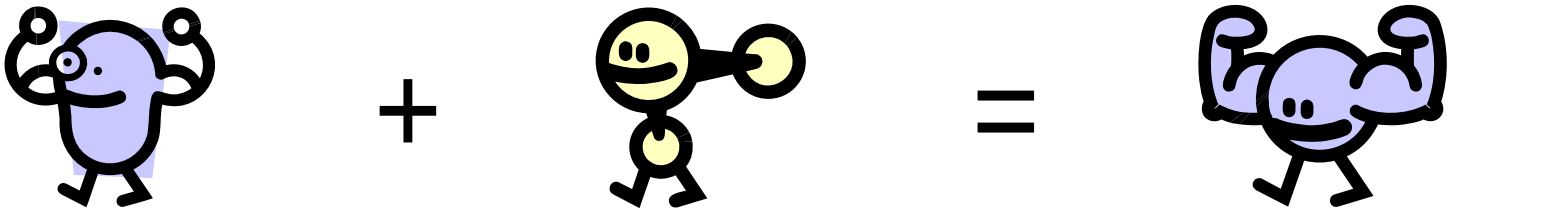
Shape Analysis with Structural Invariant Checkers

Bor-Yuh Evan Chang
Xavier Rival
George C. Necula

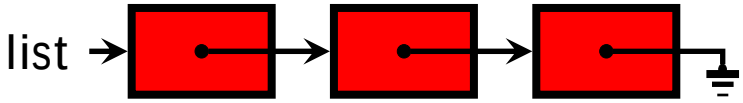
May 10, 2007
OSQ Retreat

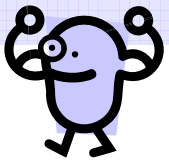
What's shape analysis? What's special?



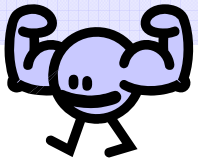
Shape analysis tracks memory manipulation flow-sensitively.

Typestate with shape analysis



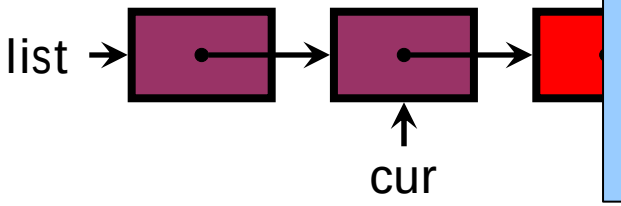


1 region
"list points to a *red* region"



1 region
"list points to a *red* region"

```
cur = list;
while (cur != null) {
   assert(cur is red);
   make_purple(cur);
   cur = cur→next;
}
```



make_purple(·) could be
- free(·)
- open(·)
- ...

…nts to

…o to

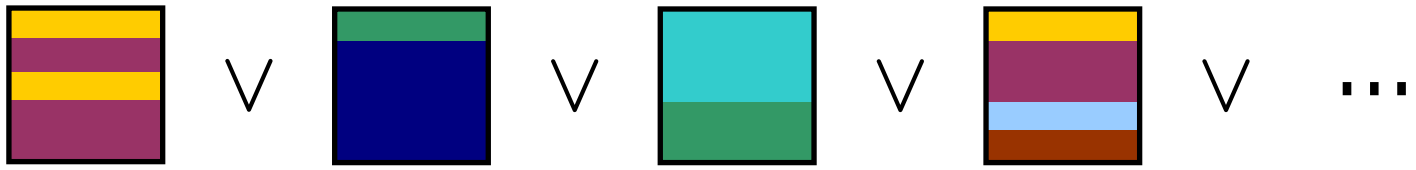
…may be *red* *or purple*"

…cur and cur points to a *red* region"

Shape analysis is not yet practical

- Scalability
  - Finding right amount of abstraction difficult

    ➡ Over-reliance on disjunction for precision



    - Repeated work to transition on each disjunct
- Usability
  - Choosing the abstraction difficult
    - Depends on the program and the properties to verify

Hypothesis

The developer can describe the memory with a small number of abstract descriptions sufficient for the properties of interest.

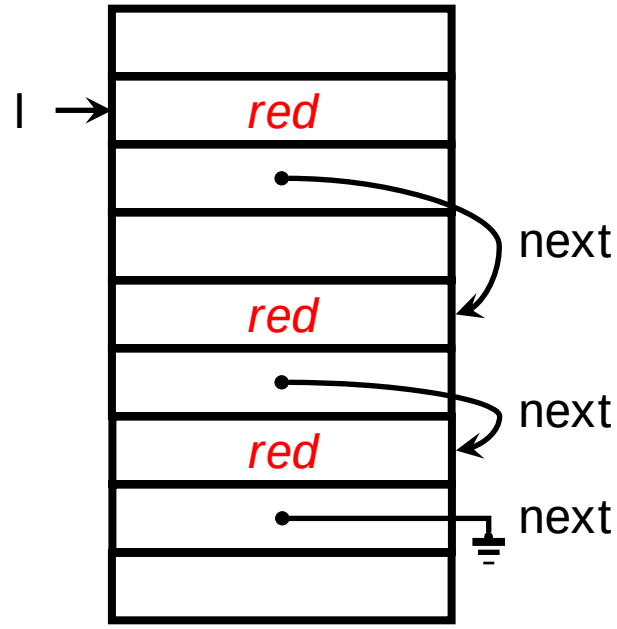
- Good abstraction is program-specific
- Developer can only keep a few cases in her head

- If only the shape analysis could get the developer's abstraction (easily)

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.

- Given: Program + Checker code

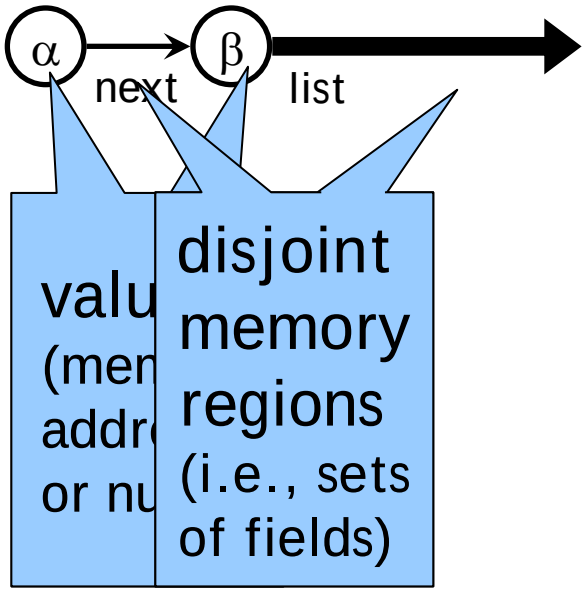
- Extensible
  - Abstraction based on the developer-supplied checkers on a per-structure basis
- Scalable (hopefully, based on hypothesis)

Outline

- **Memory abstraction**
  - **Challenge: Intermediate invariants**
- Analysis algorithm
  - Challenge: Blurring to ensure termination
- Comparison with TVLA
- Experimental Results

Abstract memory using checkers

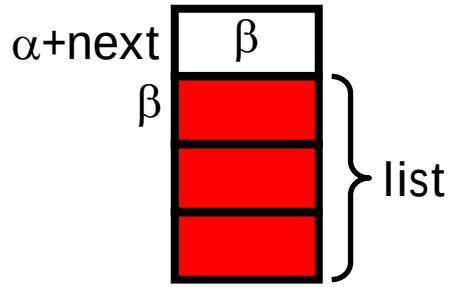
Graphical Diagram



Formula

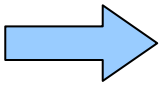
$\alpha@next \mapsto \beta \ * \ list(\beta)$

"α is a list with at least one element"



Checkers as inductive predicates

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



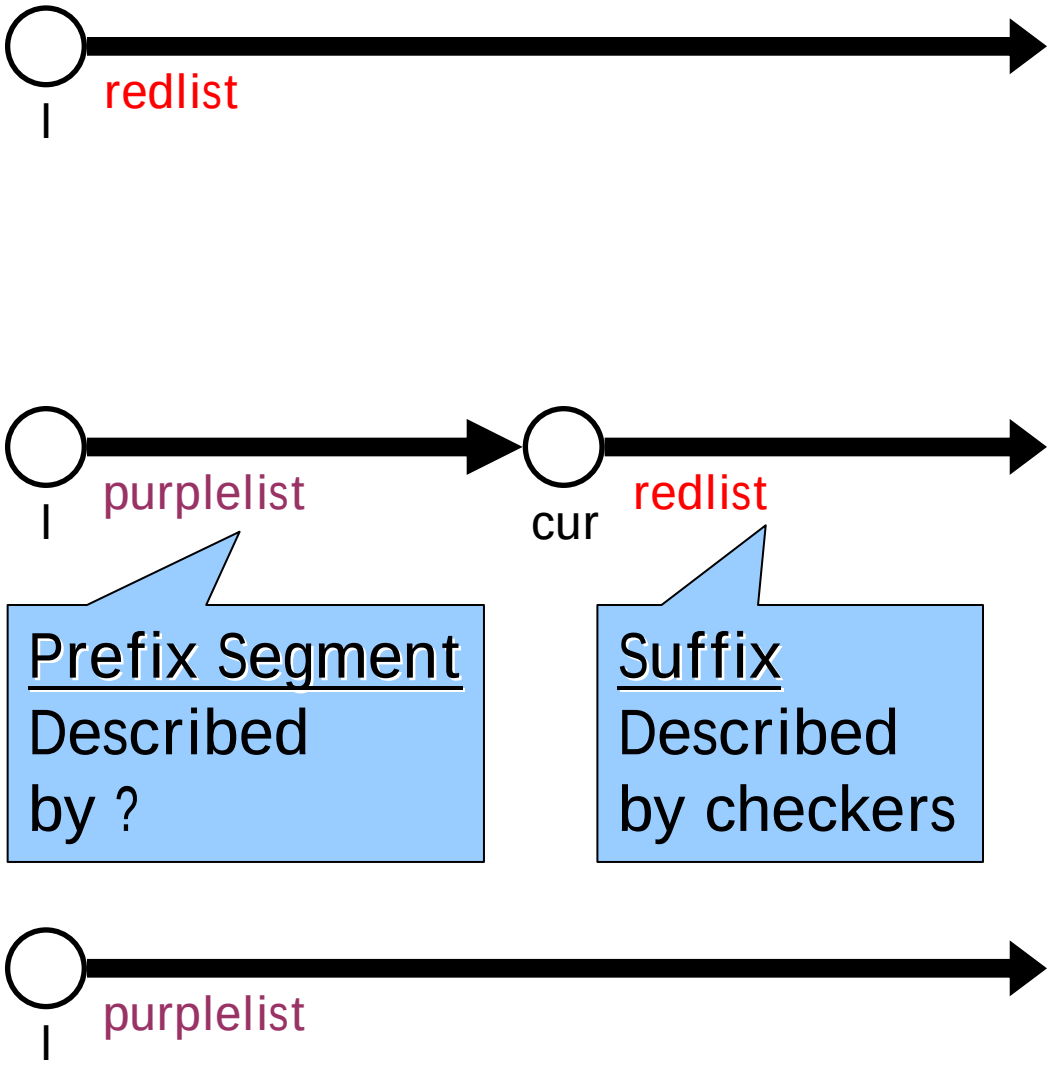
$$\begin{aligned} list(\alpha) = \exists\beta.\ & (emp \wedge \alpha = null) \\ \vee\ & (\alpha@next \mapsto \beta\ *\ list(\beta) \\ & \wedge\ \alpha \neq null) \end{aligned}$$

- Disjoint memory regions

  ➡ Checker run can dereference a field only once

Challenge: Intermediate invariants

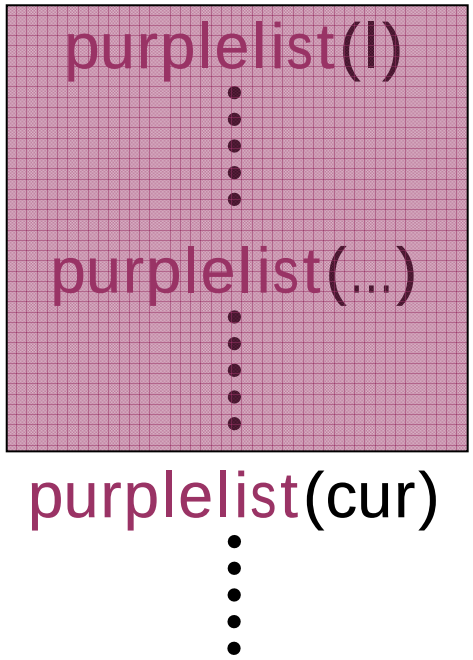
```
assert(redlist(l));
cur = l;
while (cur != null) {

  make_purple(cur);
  cur = cur→next;
}
assert(purplelist(l));
```

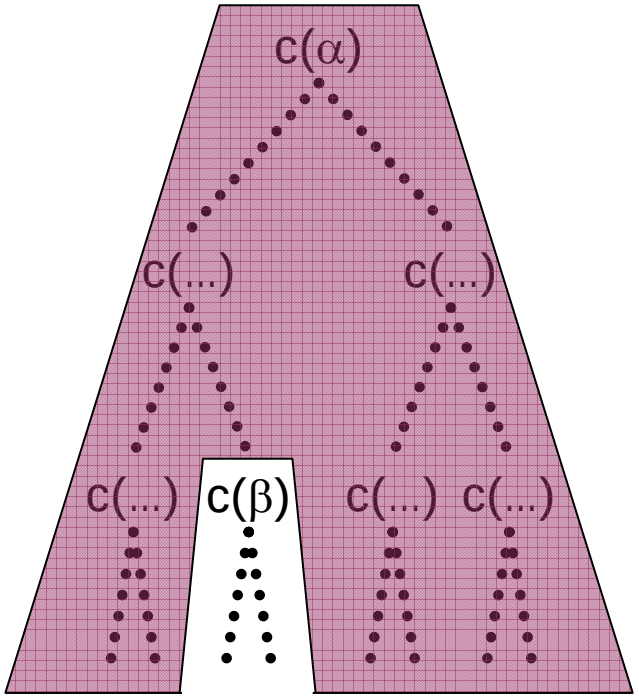


Prefix segments as partial checker runs

Abstraction

Computation Tree of a Checker Run





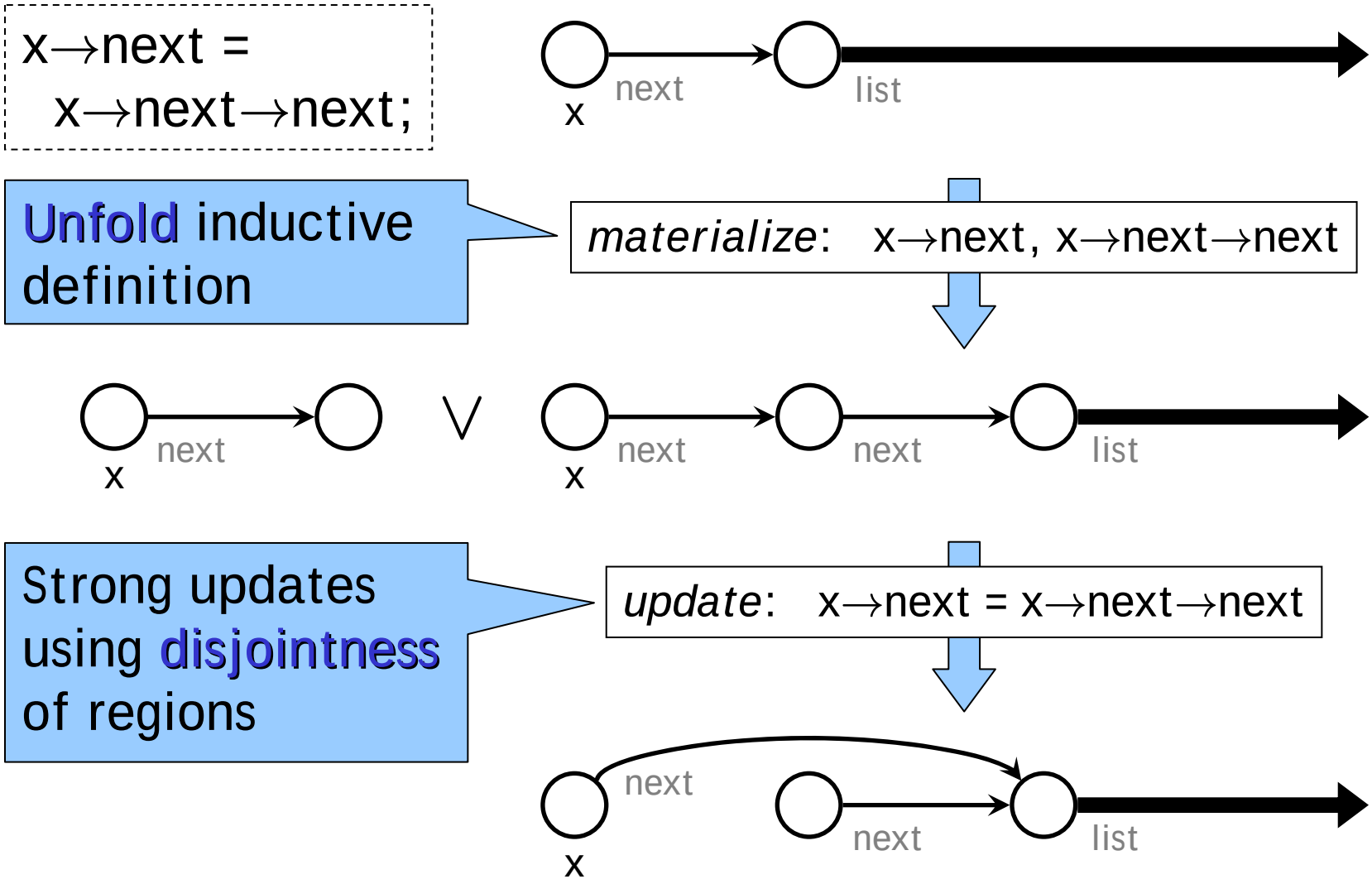
Formula

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

Outline

- Memory abstraction
  - Challenge: Intermediate invariants
- **Analysis algorithm**
  - **Challenge: Blurring to ensure termination**
- Comparison with TVLA
- Experimental Results

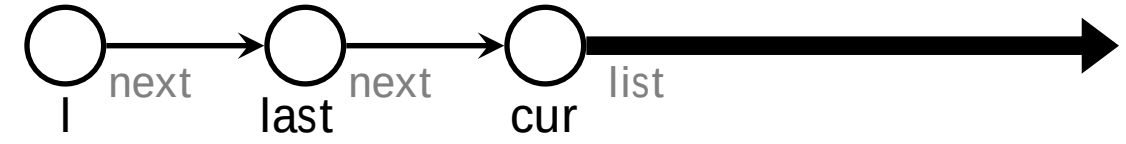
Flow function: Unfold and update edges

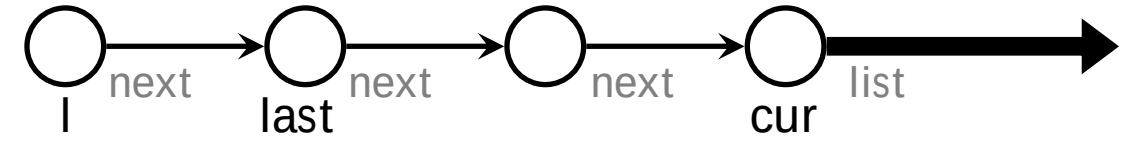


Challenge: Termination and precision

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

Observation
Previous iterates are "less unfolded"





Fold into checker edges

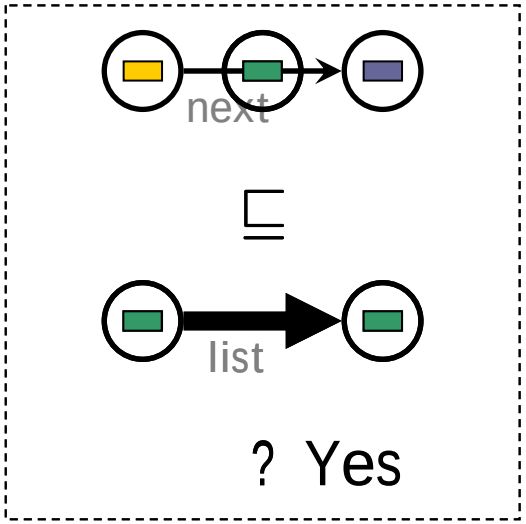
But where and how much?

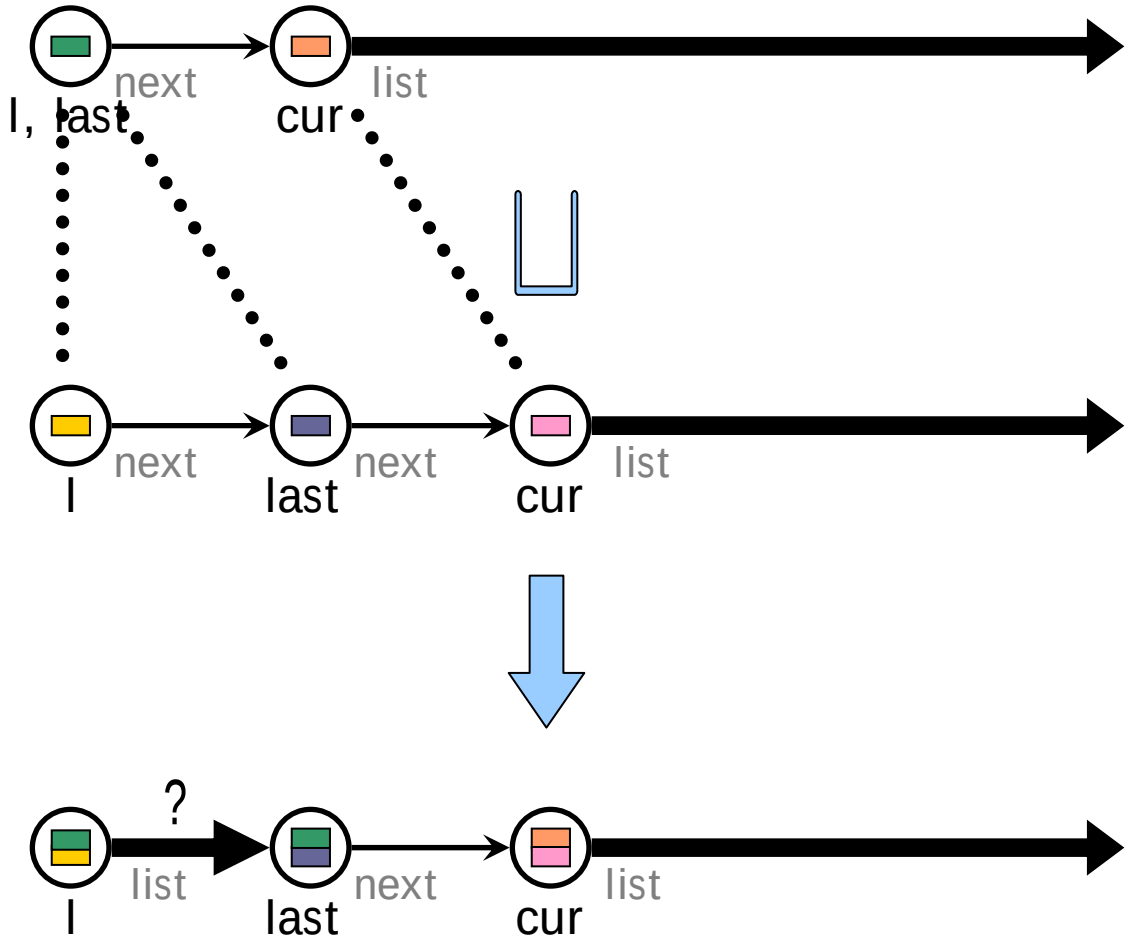
blur

History-guided folding

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

- Traverse starting from variables
- Match same edges to identify where to fold
- Apply weakening rules





Outline

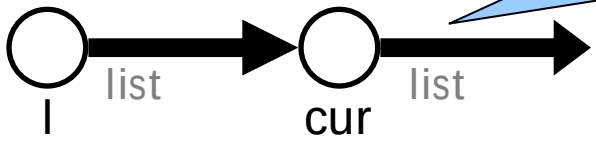
- Memory abstraction
  - Challenge: Intermediate invariants
- Analysis algorithm
  - Challenge: Blurring to ensure termination
- **Comparison with TVLA**
- **Experimental Results**

Qualitative comparison with TVLA

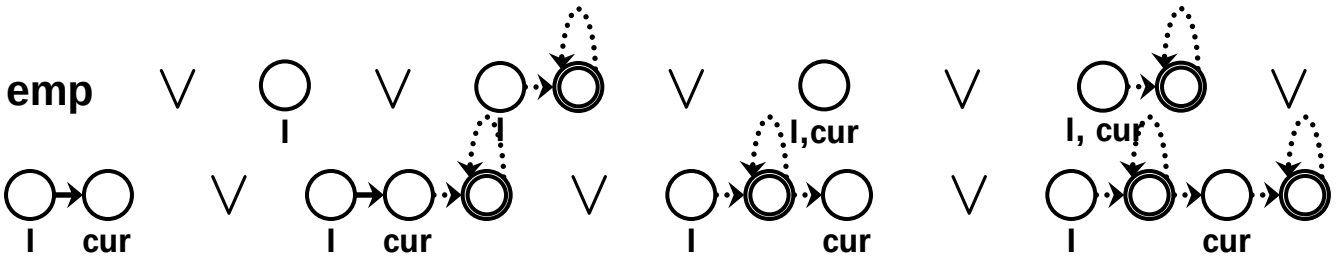
- Scalability
  - Disjunctions

Cost 1: Spec less general
Cost 2: Folding complicated

Proposal



TVLA



- Expressiveness
  - Currently, limited in comparison (no data properties)

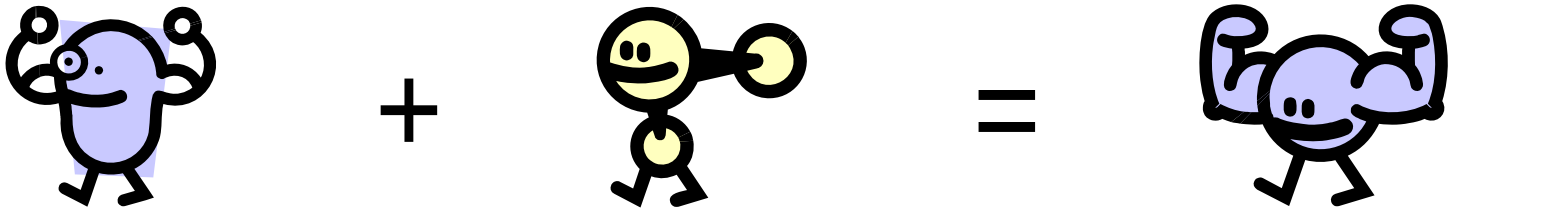
Preliminary results

Benchmark	Lines of Code	Analysis Time	Max. Num. Graphs at a Program Point	Max. Num Iterations at a Program Point
list reverse	31	0.007s	1	3
list insertion sort	80	0.021s	4	7
skip list rebalance	43	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

- Shape analysis can improve higher-level analyses



- Invariant checkers can form the basis of a memory abstraction that
  - Is easily extensible on a per-program basis
  - Expresses developer intent