CS711 Advanced Programming Languages

Shape Analysis Overview

Radu Rugina

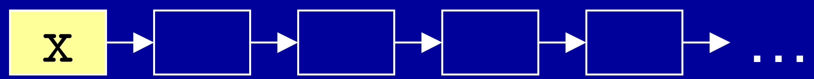
15 Sep 2005

Shape Analysis

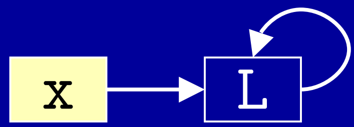
- Shape Analysis
  - A metaphor for invariants or properties that describe "data structure shapes"
  - Focuses on dynamic heap structures
    - "shape analysis" = "heap analysis"
  - Difficult case: recursive structures
  - E.g, "tree structure", "dag", "acyclic list"
  - Even "sorted list", "binary search tree", "tree balancing"

Why Isn't Pointer Analysis Enough?

- Example:



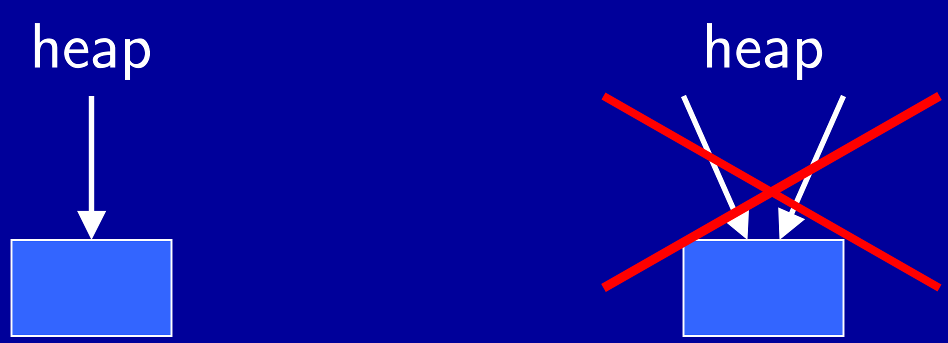
- Typical pointer analysis result:



- Imprecise: doesn't say that list is free of cycles

Example Shape Invariant

- Lack of shared-ness/cyclicity: reference count $= 1$
  - Distinguish trees from graphs, detect (lack of) cycles
  - Invariant expresses non-aliasing



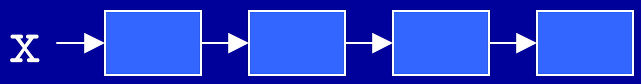
Challenge 1

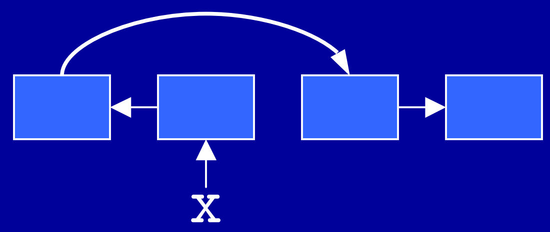
- Heap abstraction:
  - How do we "name" heap cells?
  - Recursive structures: unbounded number of heap locations
    - How to we model them using a finite abstraction?
  - Need more than "one abstract location per-allocation site"

Challenge 2

- Destructive updates: invariants temporarily broken

```
List *swap(List *x) {
   List *y, *t;
   if (x != NULL &&
       x->next != NULL) {
     y = x;
     x = y->n;
     t = x->n;
     y->n = t;
     x->n = y;
   }
   return x;
}
```

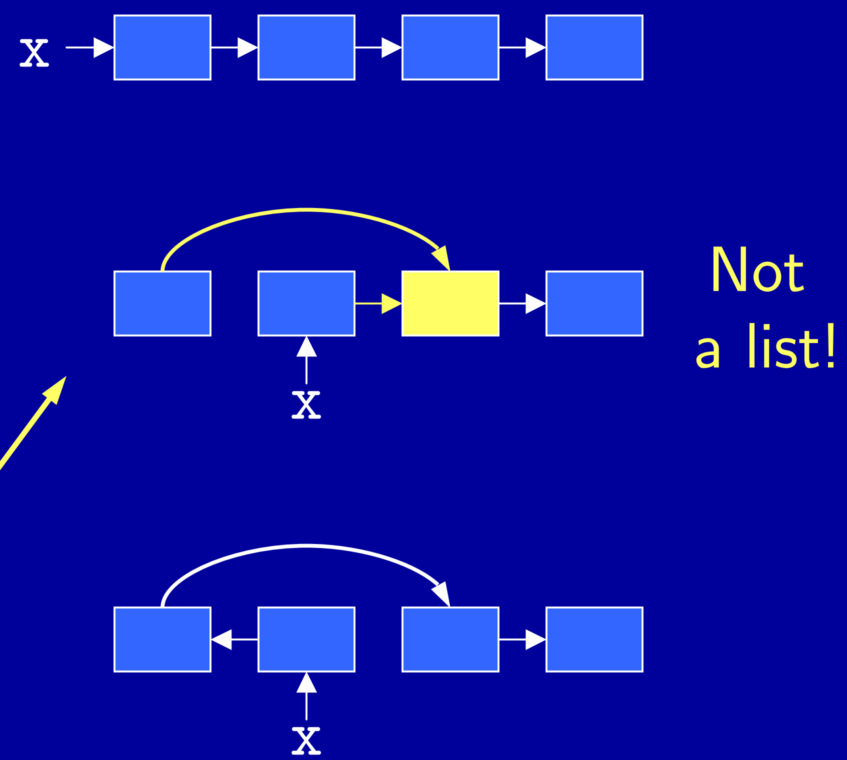




Challenge 2

- Destructive updates: invariants temporarily broken

```
List *swap(List *x) {
   List *y, *t;
   if (x != NULL &&
       x->next != NULL) {
     y = x;
     x = y->n;
     t = x->n;
     y->n = t;
     x->n = y;
   }
   return x;
}
```



Challenge 2

- Destructive updates: invariants temporarily broken
  - Shape analysis is necessarily flow-sensitive
  - Abstraction must be powerful enough to recover invariants
  - Functional languages fundamentally easier

Challenge 3

- Interprocedural analysis:
  - More complicated than for pointer analysis
  - Few shape analyses have an inter-procedural component
  - Even fewer have been implemented
  - And those are expensive
  - Lack of scalability is a big concern

Timeline

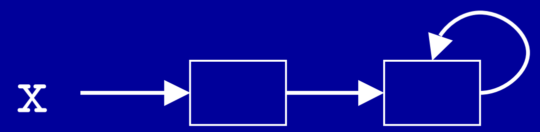
1979: Jones, Muchnick	k-limited heap abstraction
1990: Chase,Wegman,Zadeck	Shape graphs, reference counts
1990: Hendren, Nicolau 1996: Ghyia, Hendren	Reachability/access path matrices
1996: Sagiv, Reps, Wilhelm	Materialization, soundness proof
1999: Sagiv, Reps, Wilhelm	3-valued logic, TVLA
2005: Hackett, Rugina	Local reasoning, tracked locations

k-limiting [JM'79]

- k-limited heap abstraction:
  - k = a constant (e.g, 3)
  - Describe all heap shapes with depth at most k
  - Approximate the rest of the heap with "summaries"
  - Label summaries with:
    - "c" if there may be a cycle
    - "s" if there may be sharing
- Drawbacks:
  - Exponential number of shapes (large even for small k)
  - Does not distinguish between "deep" heap cells.

Storage Shape Graphs [CWZ'90]

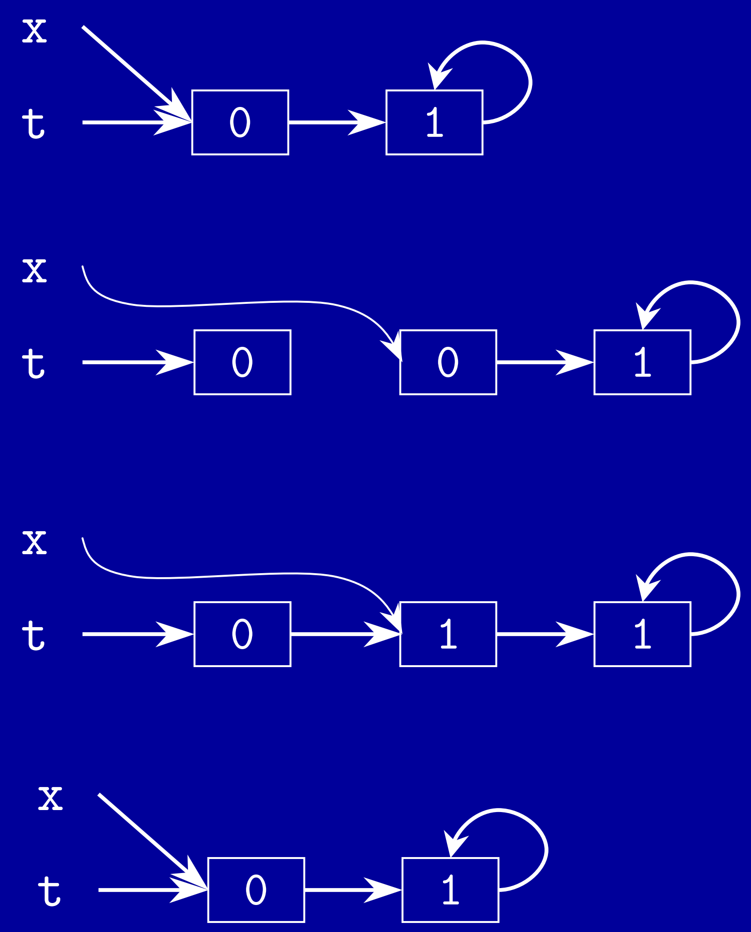
- Shape graph abstraction:
  - Distinguish between:
    - the heap cells directly pointed to by variables
    - "summary nodes" for the "deeper" heap cells (bounded by the number of allocation sites)



  - Strong updates on heap cells
  - Heap reference counts for summaries
    - Lattice $\{0, 1, \infty\}$
    - Ref counts never decreased!

Example [CWZ'90]

```
while() do {

        t = cons()

         t.cdr = x

           x = t

  }
```



Limitations [CWZ'90]

- Ref counts are never decreased
  - Swap example doesn't work
  - Neither does insertion/deletion into the middle of a list or tree
- Once a heap cell is summarized, it cannot be "unsummarized"
  - Cannot perform strong updates in such cases
  - Imprecise for programs that traverse recursive structures and destructively updates them

Quote [CWZ'90]

- "Stransky also proposes a similar analysis in his thesis [Str88], but we are unable to compare our work with his because our French is inadequate."

Shape Graphs [SRW'96]

- Similar to [CWZ'90]:
  - Model each heap cell by the set of variables that point to it
  - Abstraction size bounded by $2^{|Var|}$
  - Exactly one summary node: the "Ø node"
  - Variables always point to non-summary nodes
    - Can always perform strong updates
  - Label the summary with a "sharing" flag

Materialization [SRW'96]

- Key innovation: make non-summary nodes from summary nodes
  - Enables strong updates during structure traversals


```
y = x.n
```

Materialization [SRW'96]

- Key innovation: make non-summary nodes from summary nodes
  - Enables strong updates during structure traversals

y = x.n

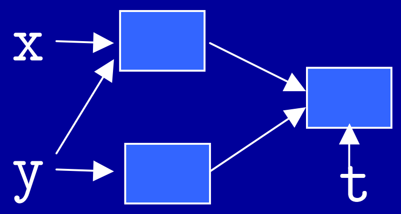
Summarization [SRW'96]

- Summarization = dual operation
  - Similar to [CWZ'90]

`y = nil`

Compatibility [SRW'96]

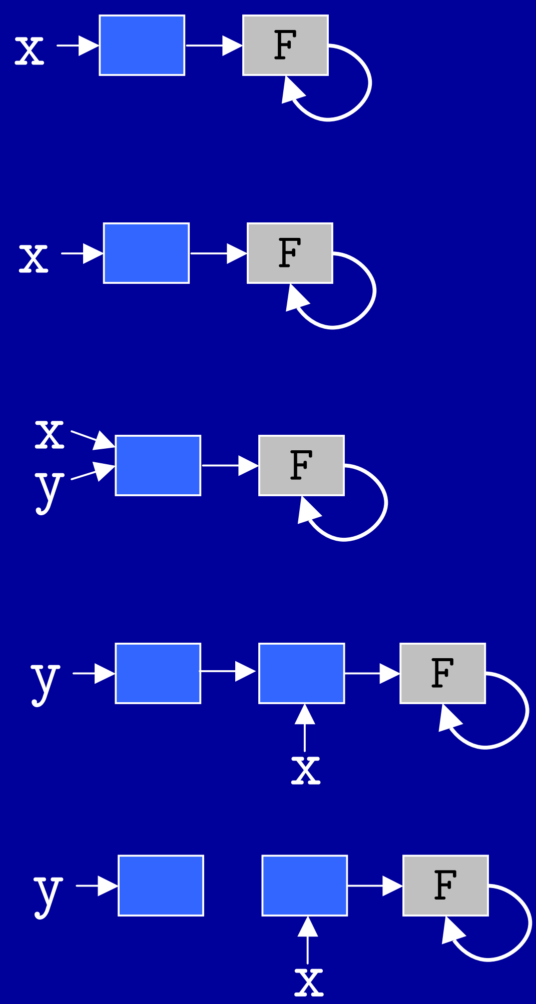
- Some nodes cannot occur in the same concrete heap
  - when the intersection of their pointed-by sets is non-empty
  - Incompatible edges = edges that involve incompatible nodes



- Can use a set of shape graph at each program point to avoid this issue
  - Simpler; presented in then Nielson/Nielson/Fleming book

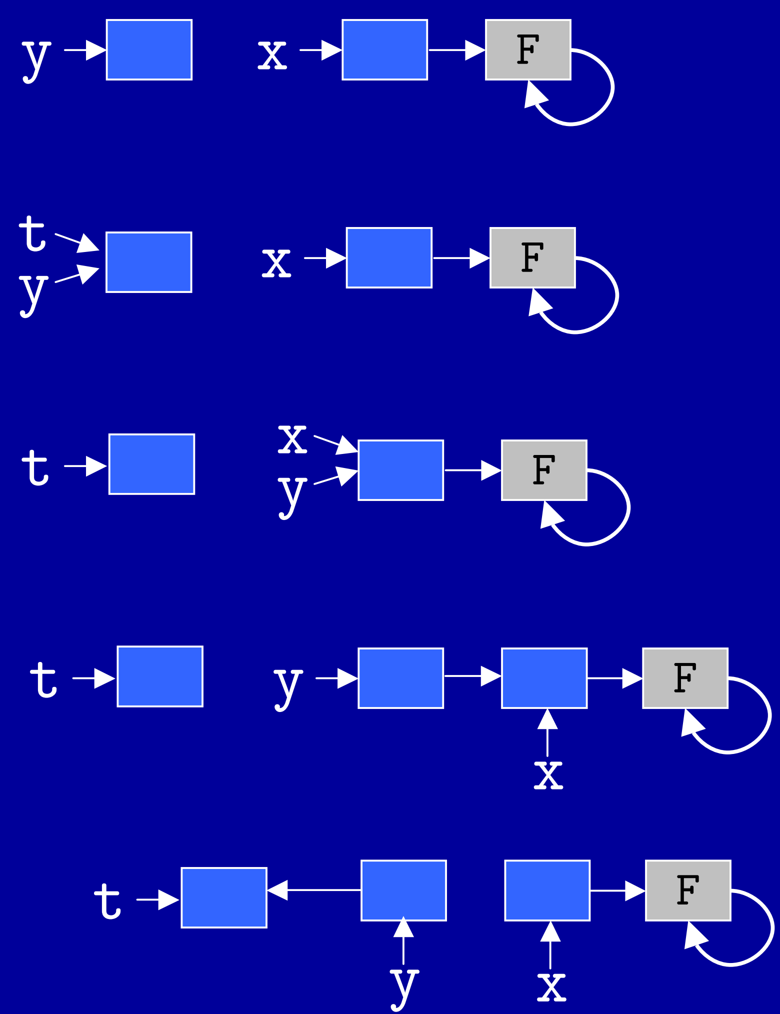
List Reversal

```
while (x != nil) do {

        t = y

        y = x

      x = x.cdr

       y.cdr = t

  }
```



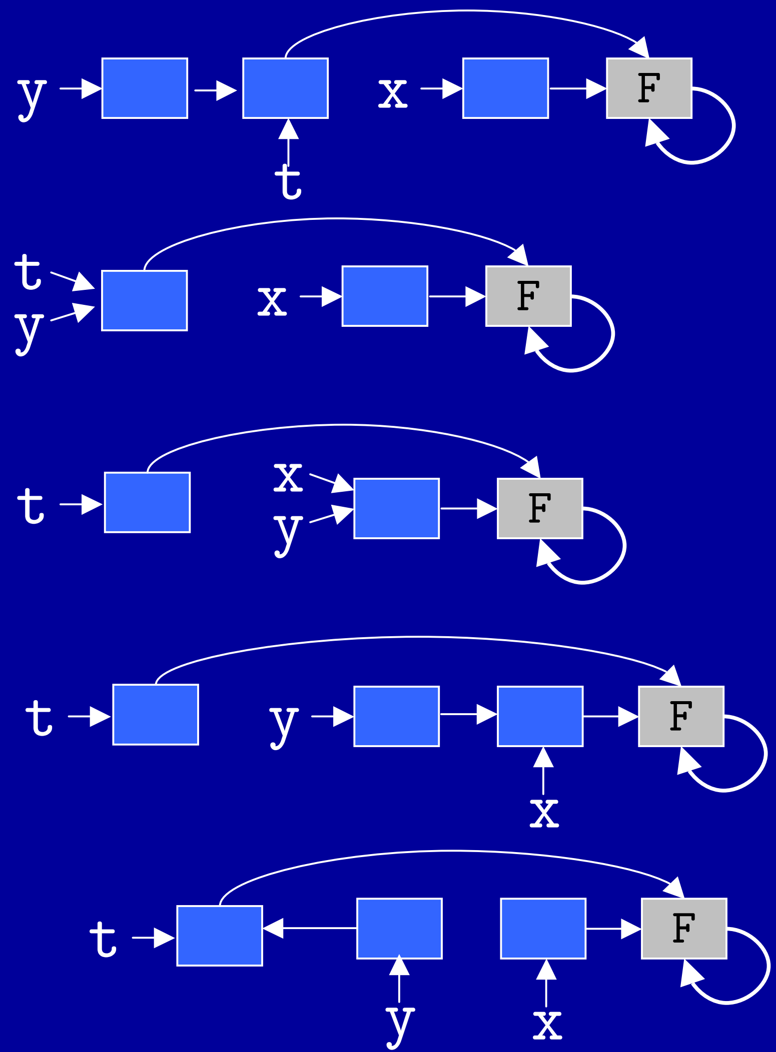
Second Iteration

```
while (x != nil) do {

              t = y

              y = x

            x = x.cdr

             y.cdr = t

    }
```



Fixed Point

```
while (x != nil) do {

            t = y

            y = x

           x = x.cdr

            y.cdr = t

  }
```



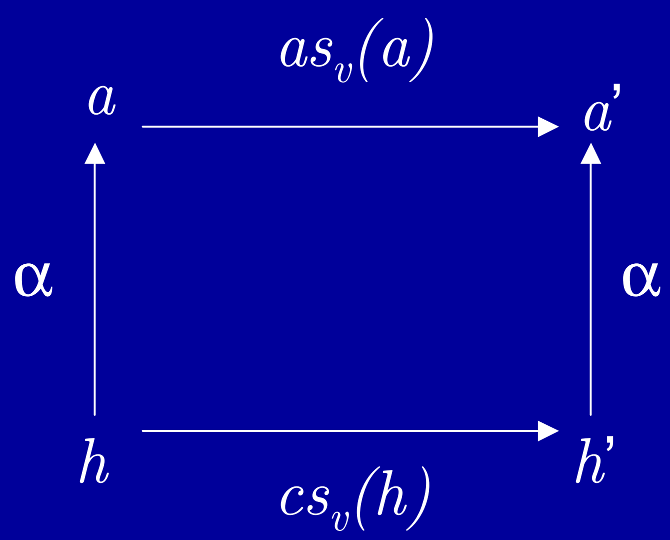
Transfer Functions

- Relatively easy for all except load/store
  - Assume each assignment preceded by a nullification
  - For x = null: remove x from the set, merge into summary if necessary
  - For x = y: add x to all of the nodes that contain y
- Load: may trigger materialization
- Store: perform strong updates
- Additional complexity because of node compatibility

- Overall, fairly sophisticated analysis

Soundness

- Formal proof of soundness:
  - Give a concrete (operational) semantics (cs)
  - Define abstraction function (α)
  - Show that transfer functions (as) and semantics (cs) agree for each assignment statement v



$$\alpha(cs_v(h)) \sqsubseteq as_v(\alpha(h))$$

Complexity

- One shape graph per program point: exponential in the number of variables: $2^{|Var|}$

- Set of shape graphs per program point: doubly exponential: $2^{2^{|Var|}}$