Shape Analysis with Tracked Cells

Radu Rugina
Cornell University

2 July 2007

Motivation

- Informal definition:
  shape analysis = accurate heap analysis
- Many potential uses:
  - Program verification
  - Automatic parallelization
  - Memory management
  - Scalable error detection
- Shape analyses are considered expensive
  - mostly used for verification
- This talk: practical shape analysis

Why Is It Difficult?

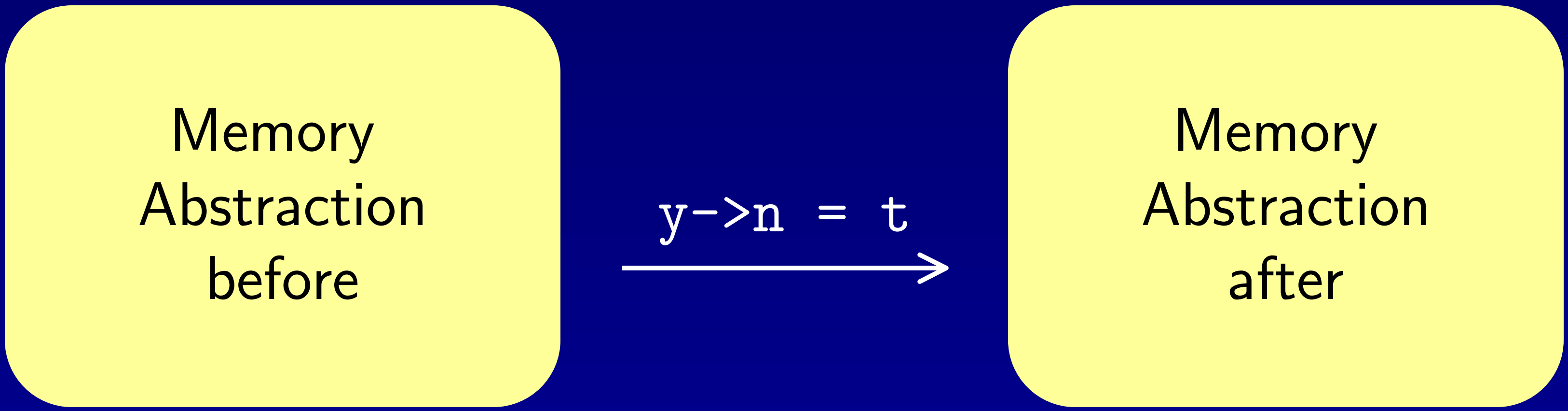
- Reason 1: Unbounded number of heap cells
  - No lexical scopes to bound their lifetimes
- Reason 2: Destructive updates
  - Structure invariants temporarily invalidated
- Reason 3: Inter-procedural interactions, recursion
  - Inter-procedural reasoning is difficult and expensive
  - Main scalability obstacle
- Many challenges, many possible solutions
  - That's why we're here today!

Overview

- Quick background
- Shape analysis with tracked cells
- Applications
- Future directions
- Conclusions

Flow-Sensitive Analysis

- Shape analysis is inherently a flow-sensitive analysis
  - E.g., abstract interpretation or dataflow analysis
- Use a memory abstraction, analyze each statement



Memory Abstraction before

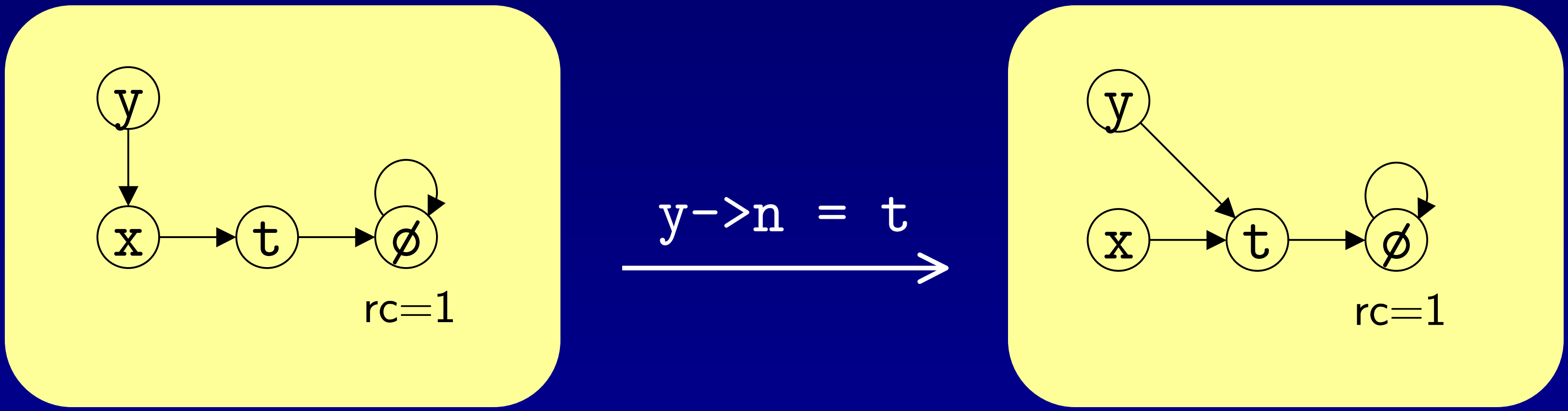
`y->n = t`

Memory Abstraction after

Abstraction: Shape Graph

- Shape analysis is inherently a flow analysis
  - E.g., abstract interpretation or dataflow analysis
- Use a memory abstraction, analyze each statement

- Shape graph = finite graph abstraction



`y->n = t`

Global Abstractions

- Global heap abstractions don't scale well
- Intra-procedural analysis is heavyweight
  - Several abstract heaps per point
- Inter-procedural shape analysis is expensive
  - Must propagate abstractions across procedures
  - Efficient inter-procedural analysis is a challenge

Claim: need local abstractions and analyses to achieve scalability

Shape Analysis with Tracked Cells

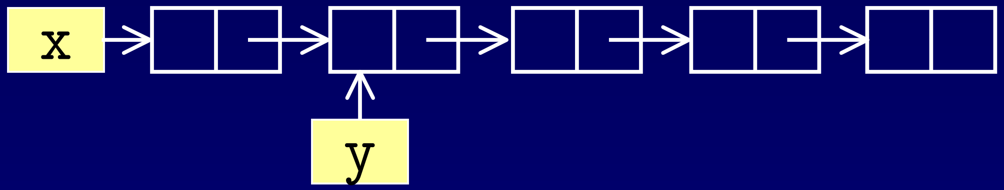
[POPL'05, ISMM'06, VMCAI'07]

Single Cell Abstraction

- The idea: abstract and analyze one heap cell at a time
  - Not the entire heap
- Local abstraction: describe the state of one cell
  - Called a “configuration”
  - No knowledge about the rest of the heap
  - Use reference counting to express heap shapes
- Local reasoning: analyze one cell at a time
  - Algorithms are easier, more efficient

Example

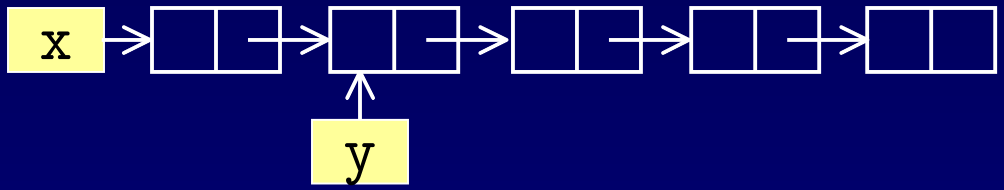
A concrete list:



```
struct list {
  int d;
  struct list *n;
} *x, *y;
```

Example

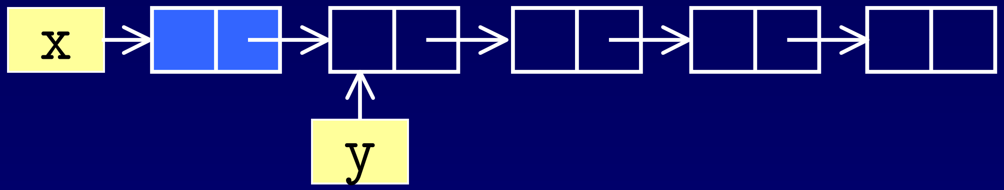
A concrete list:



Abstraction:

Example

A concrete list:

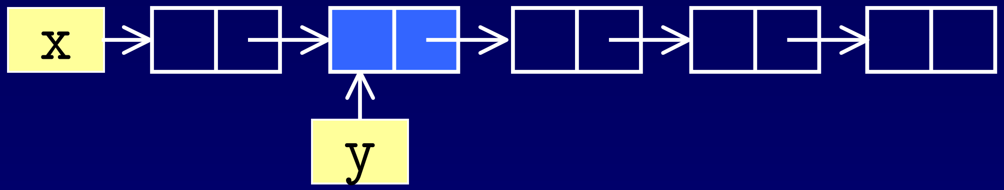


Abstraction:

(x^1)

Example

A concrete list:



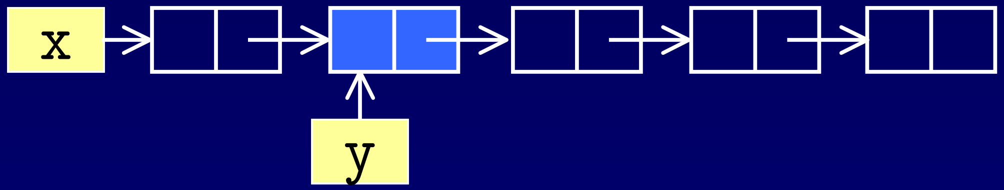
Abstraction:

(x^1)

$(y^1 n^1)$

Example

A concrete list:



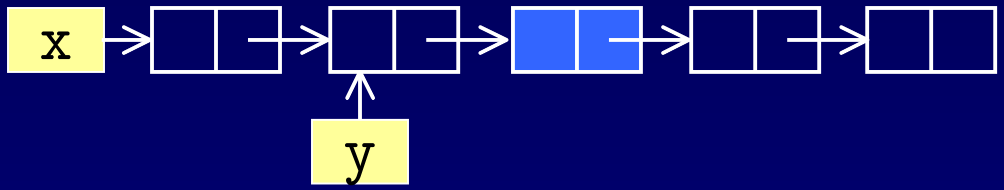
Abstraction:

(x^1)

$(y^1n^1, + x\text{->}n)$

Example

A concrete list:



Abstraction:

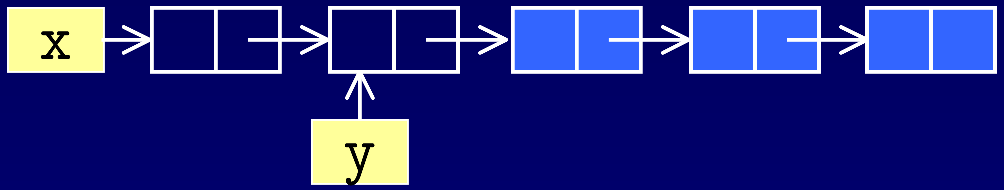
`(x`^{1}`)`

`(y`^{1}`n`^{1}`, + x->n)`

`(n`^{1}`,   - x->n)`

Example

A concrete list:



Abstraction:

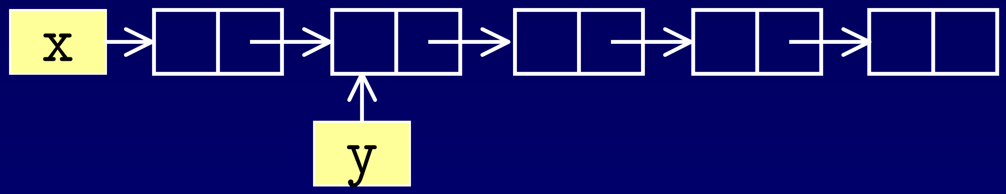
(x^1)

$(y^1 n^1, + \ x\text{->}n)$

$(n^1, \ - \ x\text{->}n)$

Example

A concrete list:



Abstraction:

(x^1)

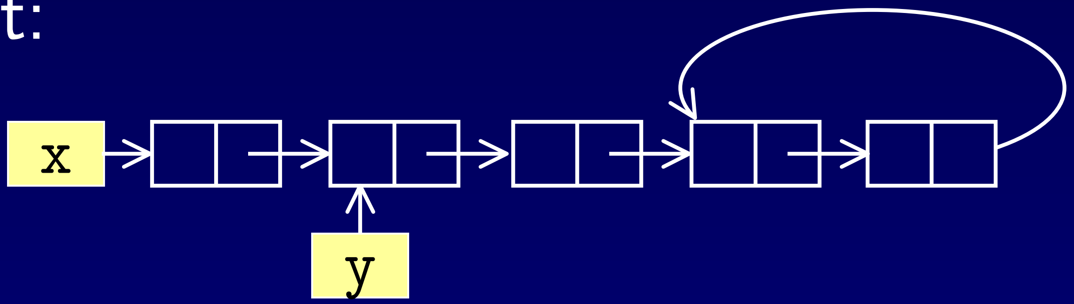
$(y^1n^1,\ +\ x\text{->}n)$

$(n^1,\ \ -\ x\text{->}n)$

"x and y point to the first two cells in an acyclic and unshared list"

Example

A cyclic list:



Abstraction:

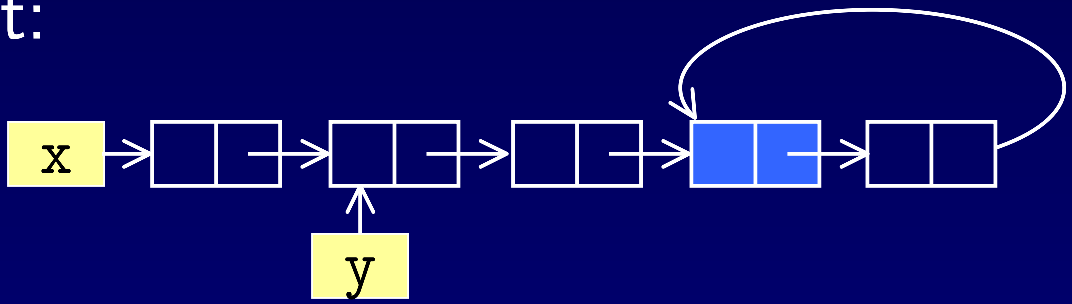
$(\mathtt{x}^1)$

$(\mathtt{y}^1\mathtt{n}^1,\ +\ \mathtt{x\text{->}n})$

$(\mathtt{n}^1,\ \ -\ \mathtt{x\text{->}n})$

Example

A cyclic list:



Abstraction:

(x^1)

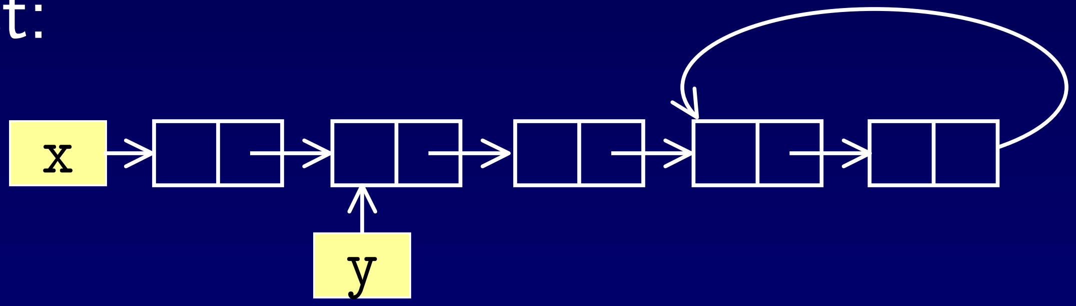
$(y^1n^1, + \ x\text{->}n)$

$(n^1, \ - \ x\text{->}n)$

$(n^2, \ - \ x\text{->}n)$

Example

A cyclic list:



Abstraction:

(x^1)

(y^{1}n^1, + x->n)

(n^1, - x->n)

(n^2, - x->n)

Analyzing List Reversal

```
List *reverse(List *x) {
   List *t, *y;
   y = NULL;
   while (x != NULL) {
     t = x->n;
     x->n = y;
     y = x;
     x = t;
   }
   return y;
}
```

Show that:

returned list `y` is acyclic

if input list `x` is acyclic

List `x` is acyclic:

(x^1) : list head

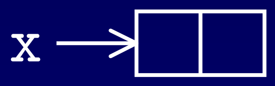
(n^1) : list tail

Loop Body Analysis

Abstraction

$\boxed{x^1}$

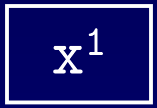
Heap Cell



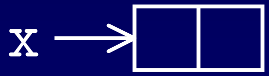
```
t = x->n;

x->n = y;

y = x;

x = t;
```

Loop Body Analysis

Abstraction



Heap Cell



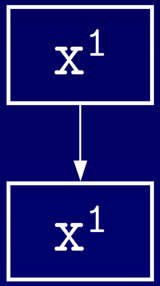
```
➡ t = x->n;

  x->n = y;

  y = x;

  x = t;
```

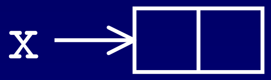
Loop Body Analysis

Abstraction

Heap Cell

```
➡ t = x->n;

  x->n = y;

  y = x;

  x = t;
```

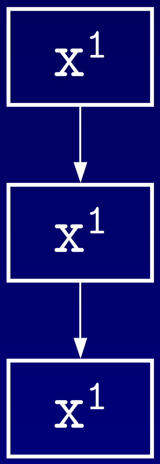




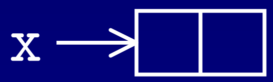
Loop Body Analysis

```
t = x->n;

x->n = y;

y = x;

x = t;
```

Abstraction



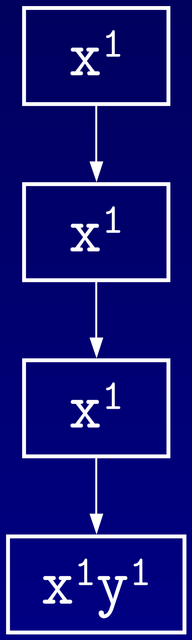
Heap Cell



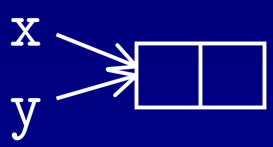
Loop Body Analysis

```
   t = x->n;

   x->n = y;

➡ y = x;

   x = t;
```

Abstraction



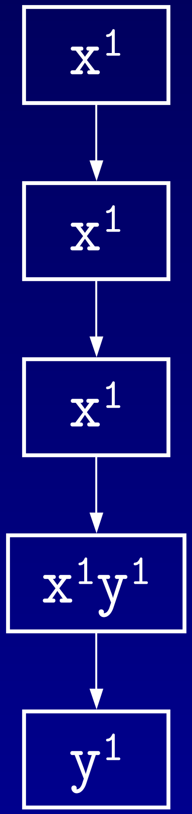
Heap Cell



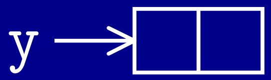
Loop Body Analysis

```
  t = x->n;

  x->n = y;

  y = x;

➡ x = t;
```

Abstraction



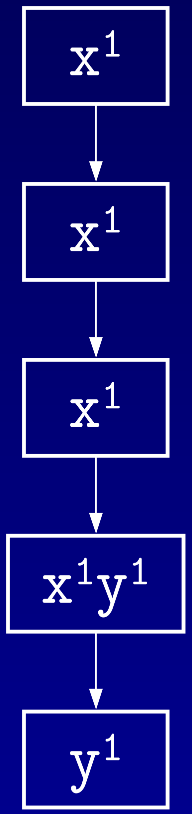
Heap Cell



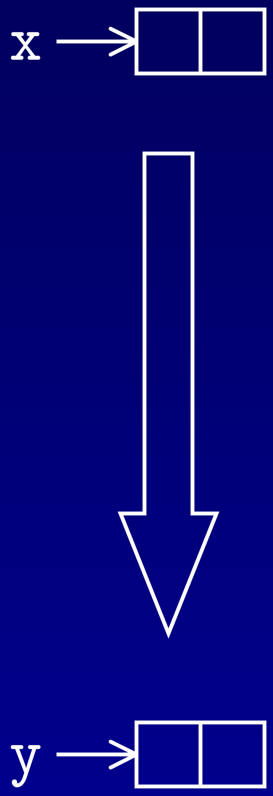
Loop Body Analysis

```
t = x->n;

x->n = y;

y = x;

x = t;
```

Abstraction



Heap Cell



Loop Body Analysis

Abstraction

Heap Cell


n^1

`?->n`

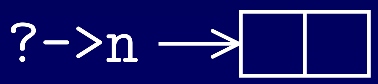
```
t = x->n;

x->n = y;

y = x;

x = t;
```

Loop Body Analysis

Abstraction

$\boxed{n^1}$

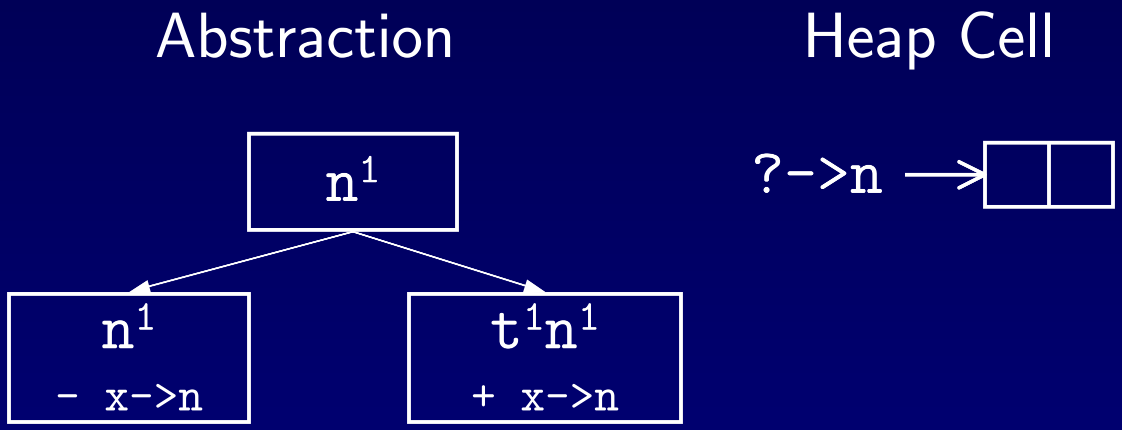
Heap Cell



```
➡ t = x->n;

  x->n = y;

  y = x;

  x = t;
```

Loop Body Analysis

```
t = x->n;

x->n = y;

y = x;

x = t;
```



Abstraction

Heap Cell

n^1

n^1
- x->n

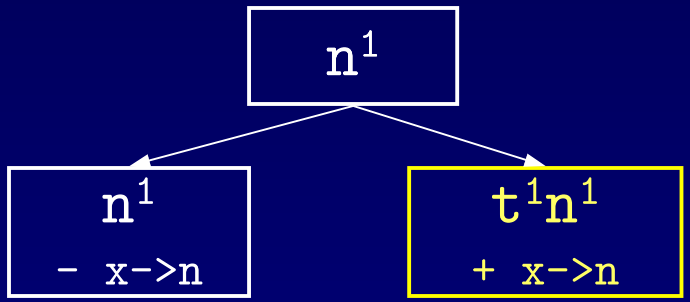
$t^1 n^1$
\+ x->n

?->n

Loop Body Analysis

```
➡ t = x->n;

  x->n = y;

  y = x;

  x = t;
```

Abstraction

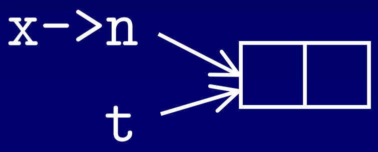


n^1

n^1
- x->n

t^1n^1
+ x->n

Heap Cell

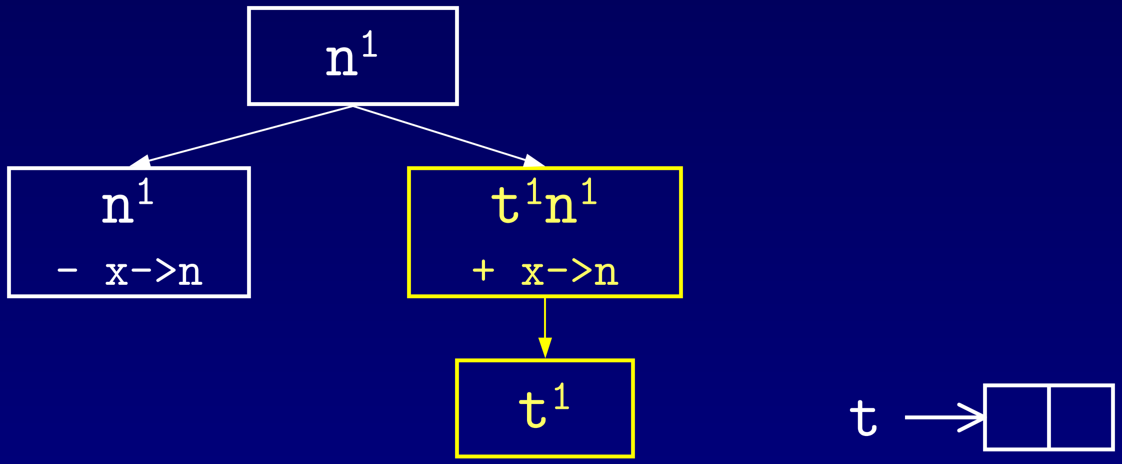


Loop Body Analysis

Abstraction

Heap Cell

```
t = x->n;

➡ x->n = y;

y = x;

x = t;
```

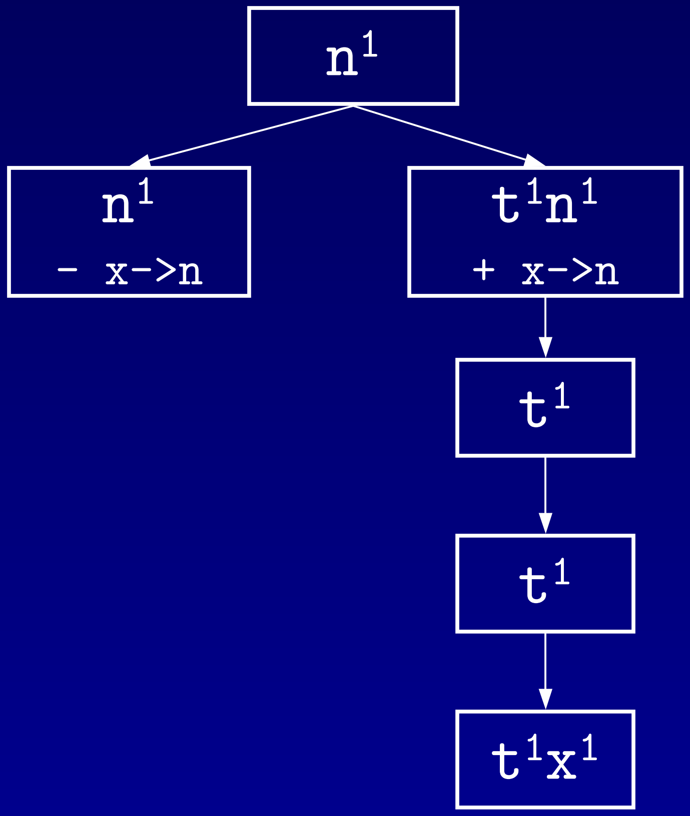


Loop Body Analysis

Abstraction

Heap Cell

```
t = x->n;

x->n = y;

y = x;

x = t;
```

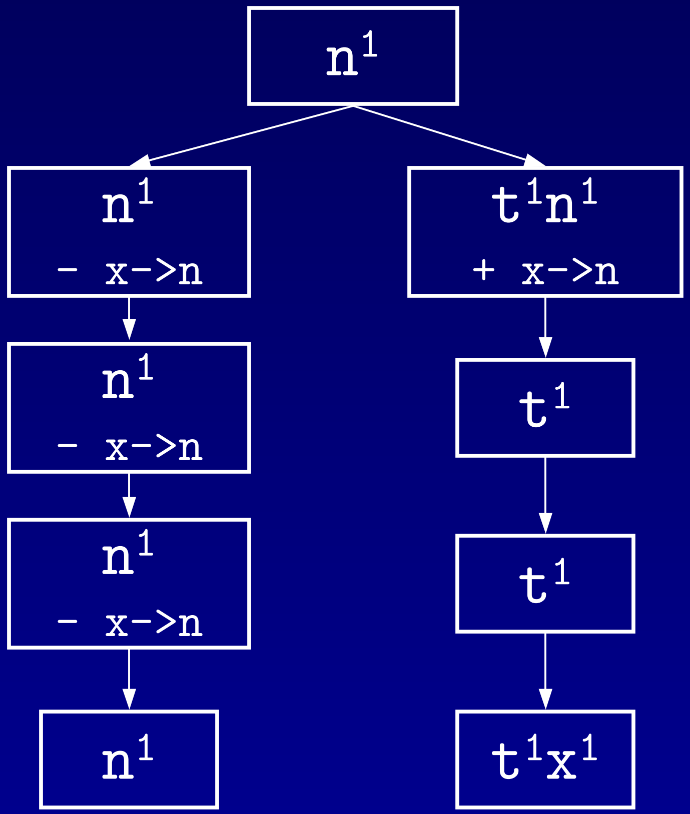


Loop Body Analysis

```
t = x->n;

x->n = y;

y = x;

x = t;
```

Abstraction

Heap Cell



n^1

n^1
- x->n

$t^1 n^1$
+ x->n

n^1
- x->n

t^1

n^1
- x->n

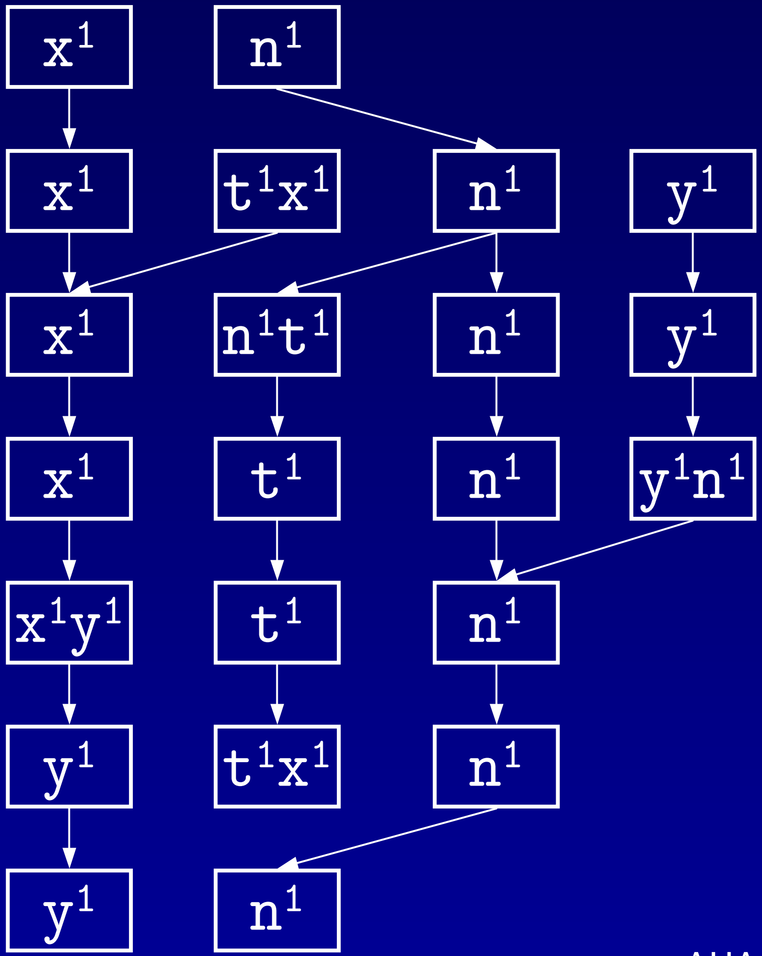
t^1

n^1

$t^1 x^1$

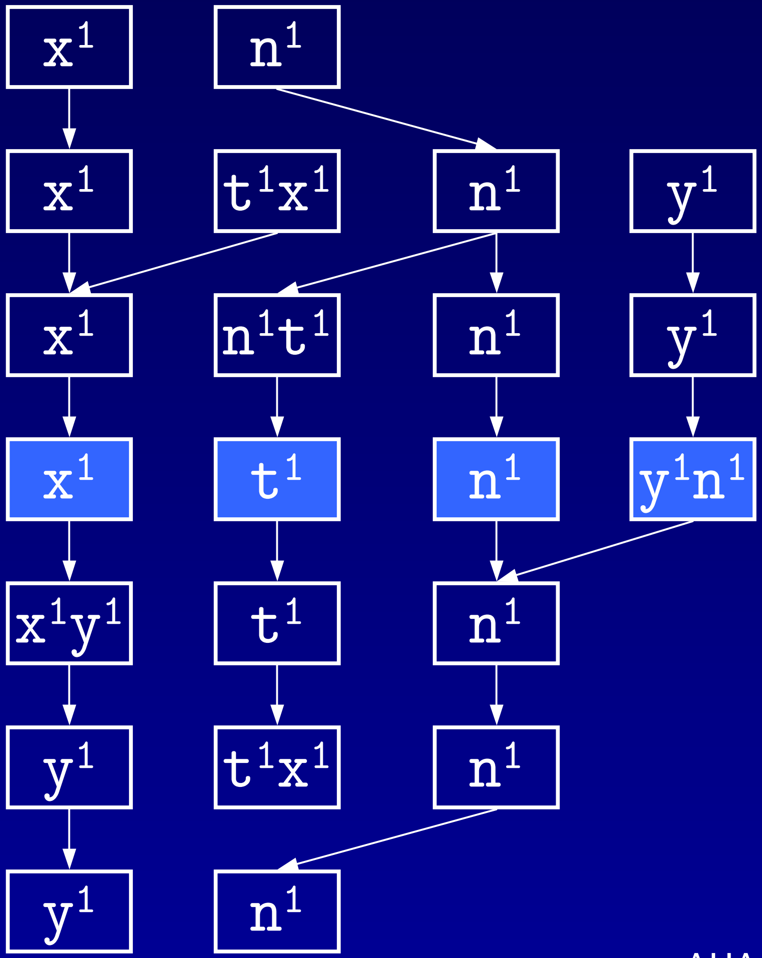
Analysis Result

```
List *reverse(List *x) {
   List *t, *y;
   y = NULL;
   while (x != NULL) {

     t = x->n;

     x->n = y;

     y = x;

     x = t;
   }
   return y;
}
```



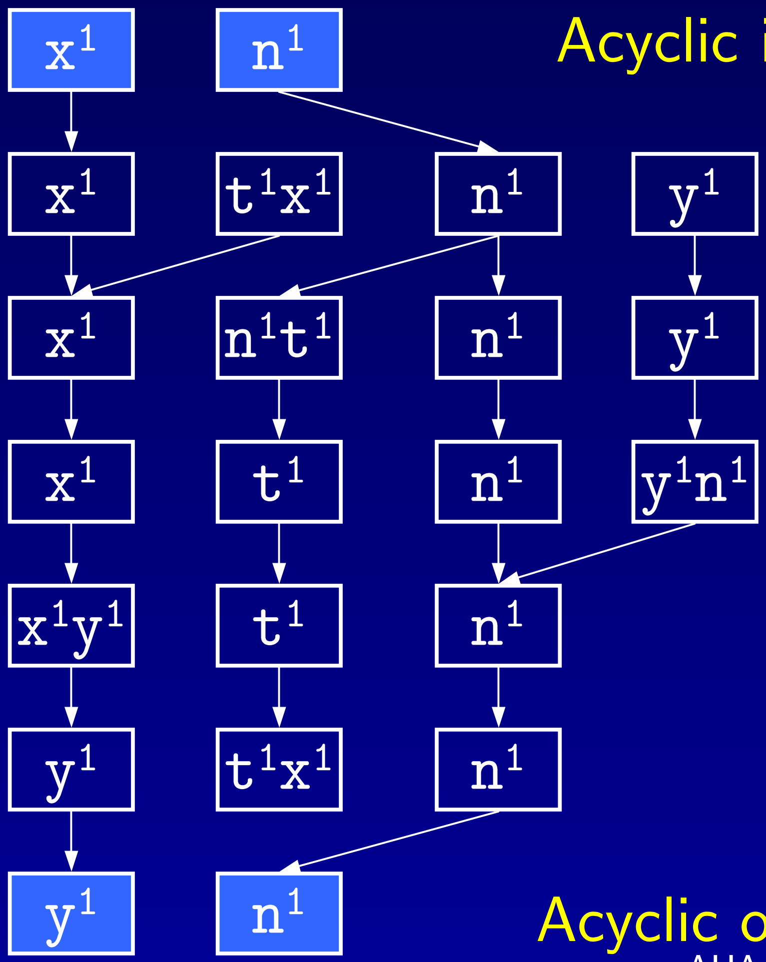
Analysis Result

```
List *reverse(List *x) {
   List *t, *y;
   y = NULL;
   while (x != NULL) {

     t = x->n;

     x->n = y;

     y = x;

     x = t;
   }
   return y;
}
```



Property Verified

```
List *reverse(List *x) {
   List *t, *y;
   y = NULL;
   while (x != NULL) {

     t = x->n;

     x->n = y;

     y = x;

     x = t;
   }
   return y;
}
```

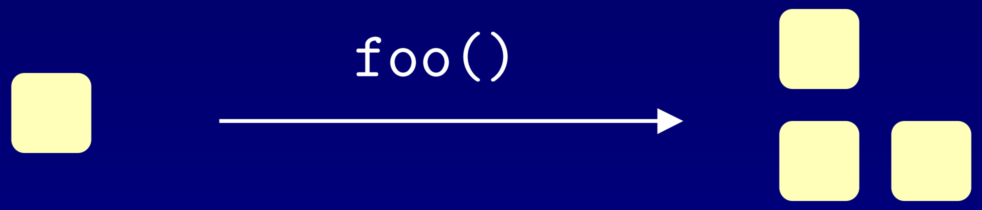


Acyclic input

Acyclic output

Inter-Procedural Analysis

- Context-sensitive analysis
- Procedure summaries: map each input configuration set of corresponding output configurations



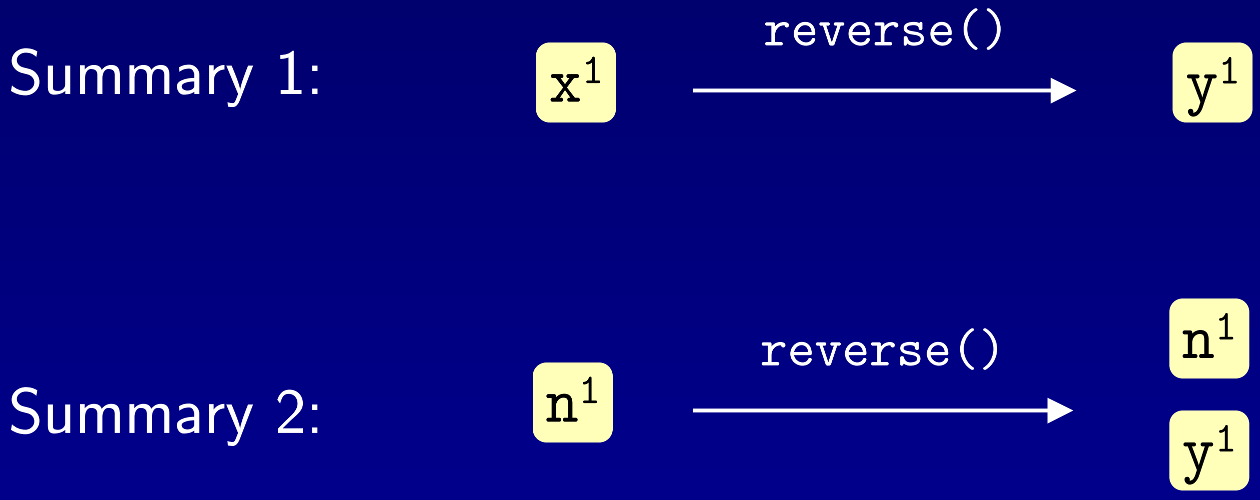
`foo()`

input

output

Inter-Procedural Analysis

- Context-sensitive analysis
- Procedure summaries: map each input configuration set of corresponding output configurations

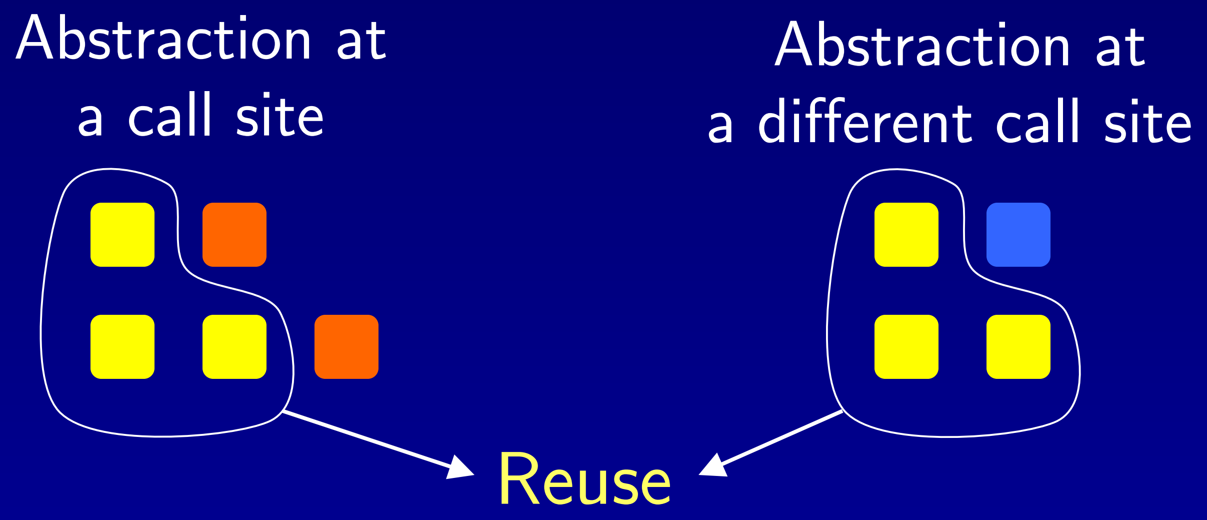


Summary 1: $\boxed{\mathtt{x}^1} \xrightarrow{\texttt{reverse()}} \boxed{\mathtt{y}^1}$

Summary 2: $\boxed{\mathtt{n}^1} \xrightarrow{\texttt{reverse()}} \boxed{\mathtt{n}^1}\ \boxed{\mathtt{y}^1}$

Inter-Procedural Analysis

- Efficient: reuse previous analyses of functions
  - Match individual configurations
    - Not entire heap abstractions
  - Works even if there is only partial overlap



Abstraction at a call site

Abstraction at a different call site

Reuse

Extensions

- Local abstraction and analysis for doubly-linked lists [VMCAI'07]
  - Describe the state of one cell and its neighbors
  - Captures local structural invariants
- Analysis by contradiction [SAS'06]
  - Backward heap dataflow analysis
  - Tracks the state of single cells backwards

Applications

Heap Error Detection

- Goal: find memory errors in C programs [POPL'05]
- Extend configurations with a boolean flag F
  - F is true when the cell has been freed

- Dangling pointer access $*e$ if:
  - e may hit a configuration with F $=$ true
  - Same for double free's

- Memory leak if:
  - A configuration has all reference counts zero
  - And F flag is false

Heap Error Detection

- Methodology
  - Analyze each allocation site in turn
  - Track cells from allocation point
  - Use fixed exploration budget per allocation site
- Results:
  - Open-source programs: `OpenSsh`, `OpenSsl`, `binutils`
  - Analyzed 70 KLOC in 2 minutes
  - 98 warnings
  - 38 errors found
- Analysis scales and is applicable to larger programs

Memory Management

- Static reclamation of heap objects [ISMM06]
  - Compile-time program transformation for garbage collected languages (e.g., Java)
  - Insert "free" statements
  - Useful for real-time and embedded systems
  - Can be integrated with mark-sweep garbage collection

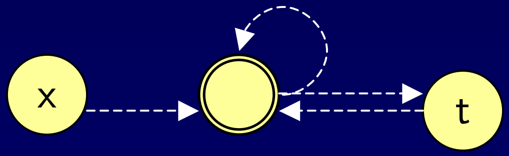
- Results:
  - SpecJVM98 benchmarks + Java library code
  - Analysis takes about 3 min per 2000 methods
  - Analysis can reclaim more than 50% of the total memory

Concluding Remarks

Current and Future Directions

- Shape analysis versus types:
  - Unique types and uniqueness inference [ISMM'07]
  - Connecting shape analysis and general alias types
- Multithreaded heap analysis
  - Use local reasoning about shared heap cells
- Refinement-based error detection
  - Gradually use more sophisticated analyses
  - Heap analysis via guarded value-flow analysis [PLDI'07]

Comparison



Shape Graphs/TVLA (all heap)
[Sagiv et al. '96,'99]

Procedure-local heaps
[Sagiv et al, '05]

ls(x,t) * ls(t,nil)

Separation Logic (list segments)
[Distefano et al. '06, Gotsman et al. '06]

x^1 n^1 t^1n^1

Tracked Cells (single location)
[Hackett,Rugina'05, Cherem,Rugina'06]

Acknowledgements

Brian Hackett	Lonnie Princehouse
Siggi Cherem	Maksim Orlovich

`http://www.cs.cornell.edu/~rugina/satc`

`http://www.cs.cornell.edu/projects/crystal`

Conclusions

- Practical shape analysis
  - Local abstraction of single heap cells
  - Local reasoning and analysis
  - Inter-procedural analysis
  - Analyses scale to larger applications

- Applications:
  - Heap shape verification
  - Find heap errors in larger programs
  - Memory management transformations