Lecture 08(c) – Predicate Abstraction and SLAM

PROGRAM ANALYSIS & SYNTHESIS

Eran Yahav

Previously

- Typestate Verification

Today

- Predicate Abstraction (via SLAM)

- Acks
  - Slides cannibalized from Ball&Rajamani's PLDI'03 Tutorial

Predicate Abstraction

- Use formulas to observe properties in a state
  - e.g., (x==y), $\forall$i.a[i]=0
- Boolean abstraction
  - over a finite vocabulary F of predicates
  - abstract state is an assignment of truth values to predicates in F

Example

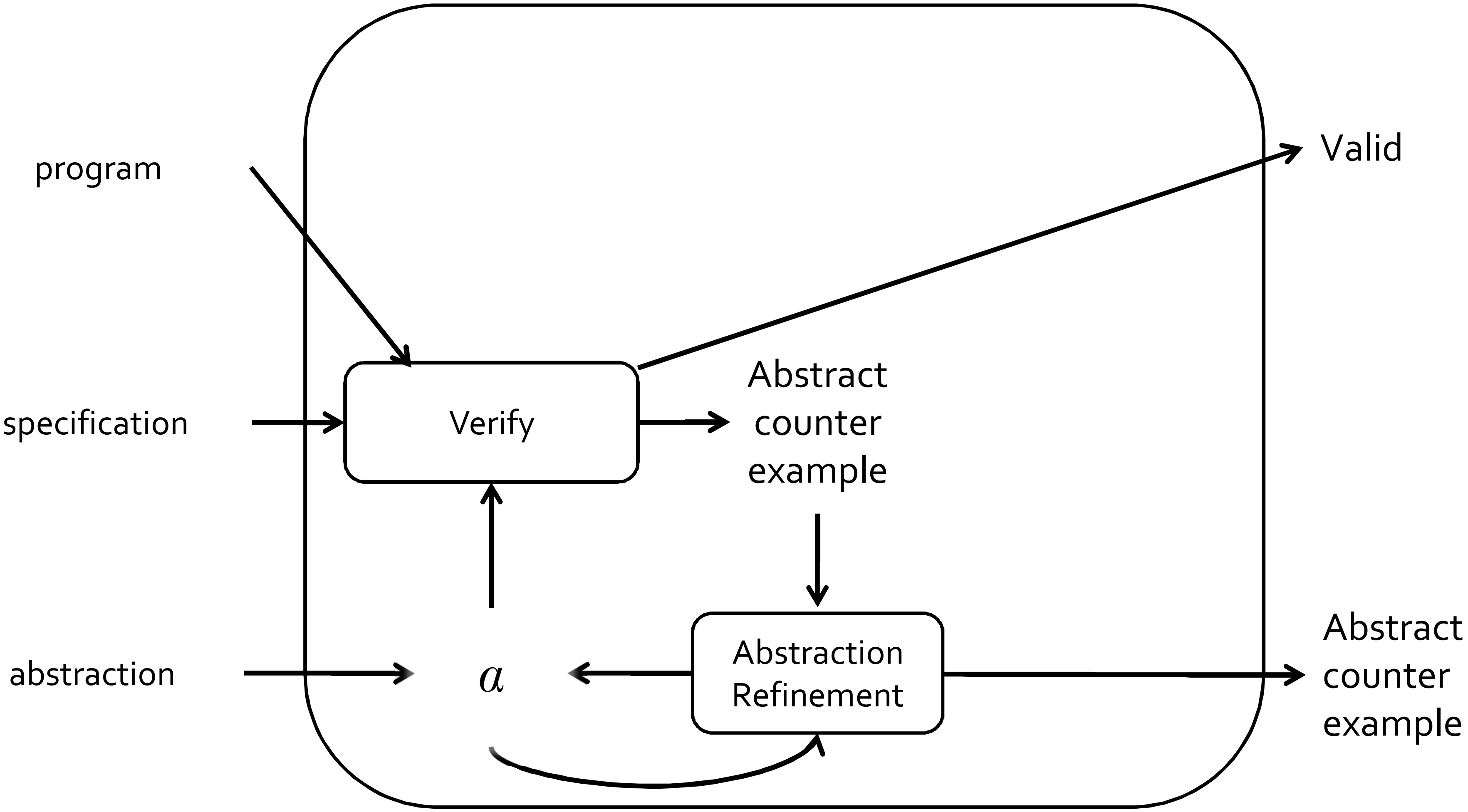
```
x=abs(?);          (x==y) = ?, (x==y-1) = ?
y=x;               (x==y) = T, (x==y-1) = F
while(x!=0) {
  x--;             (x==y) = F, (x==y-1) = T
  y--;             (x==y) = T, (x==y-1) = F
}
assert x==y;       (x==y) = T, (x==y-1) = F
```

$$F = \{ (x==y), (x==y-1) \}$$

Example

```
x=abs(?);
y=x;
while(x!=0) {
  x--;
  y--;
}
assert x==y;
```

Abstraction Refinement



Change the **abstraction** to match the **program**

Predicate Abstraction and Boolean Programs

- Given a program P and a finite vocabulary F we can take two similar (but different!) approaches

- compute $[\![P]\!]^{\#}$ abstract interpretation over F

- construct a Boolean program BP = B(P,F) and compute $[\![BP]\!]$ concrete interpretation
  - BP guaranteed to be finite-state

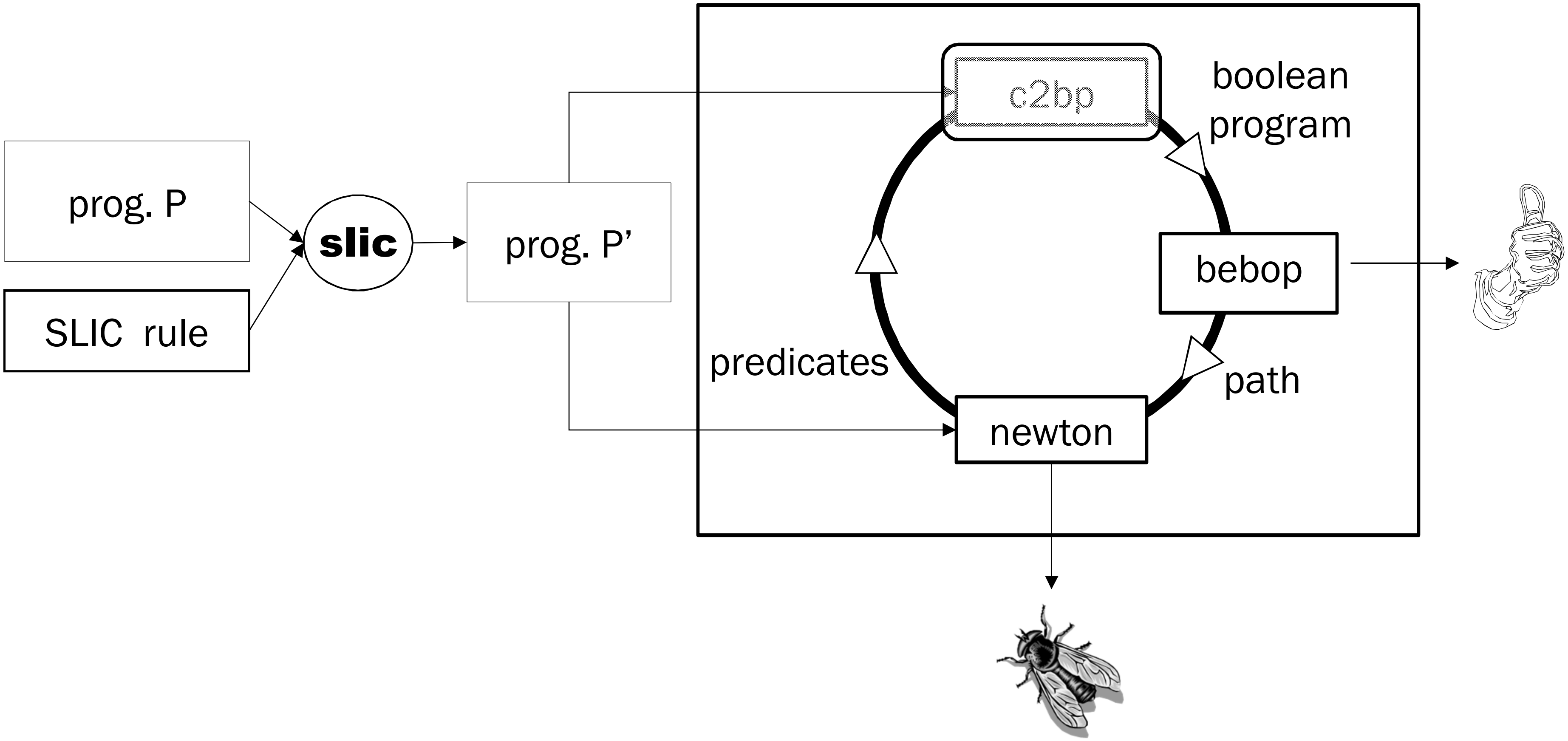
Boolean Program: Example

```
x=abs(?);
y=x;
while(x!=0) {
  x--;
  y--;
}
assert x==y;
```

```
x=abs(?);
(x==y) = T
while(?) {
  (x==y-1) =T; (x==y) = F
  (x==y-1) =F; (x==y) = T
}
assert ( (x==y) == T );
```

Use the predicates to construct a Boolean program
(how? coming up soon)

The SLAM Process



prog. P

SLIC rule

slic

prog. P'

c2bp

boolean program

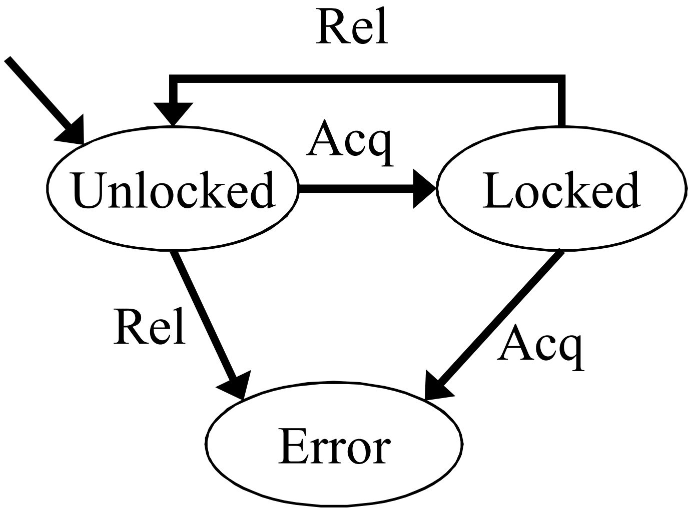
bebop

path

newton

predicates

State Machine for Locking



```
state {
  enum {Locked,Unlocked}
    s = Unlocked;
}

KeAcquireSpinLock.entry {
  if (s==Locked) abort;
  else s = Locked;
}

KeReleaseSpinLock.entry {
  if (s==Unlocked) abort;
  else s = Unlocked;
}
```

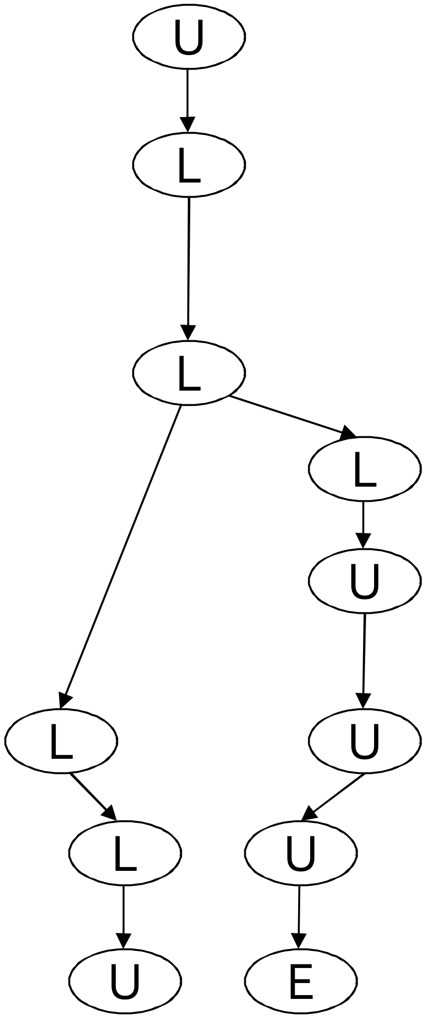
Example (SLAM)

Does this code obey the locking rule?

```
do {
   KeAcquireSpinLock();

   nPacketsOld = nPackets;

   if(request){
       request = request->Next;
       KeReleaseSpinLock();
       nPackets++;
   }
} while (nPackets != nPacketsOld);

KeReleaseSpinLock();
```

Example (SLAM)

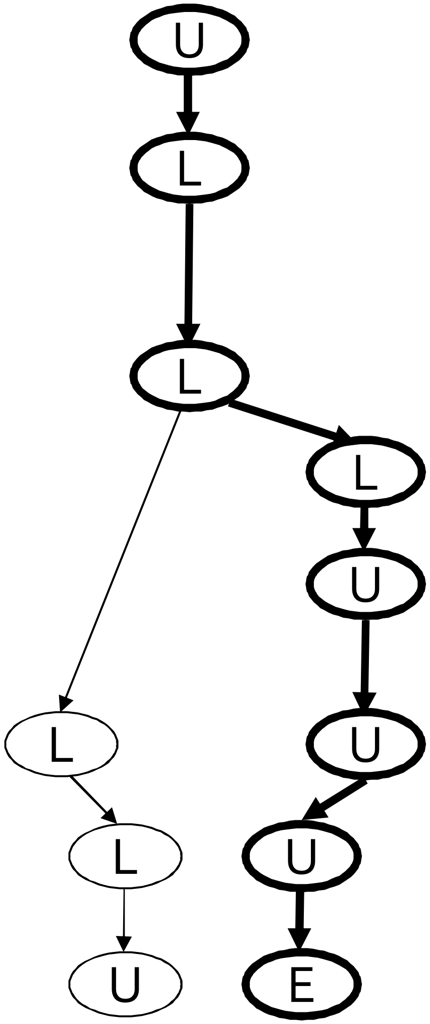
Model checking
boolean program
(bebop)



```
do {
  KeAcquireSpinLock();

  if(*){

      KeReleaseSpinLock();

  }
} while (*);

KeReleaseSpinLock();
```

Example (SLAM)

Is error path feasible
in C program?
(newton)

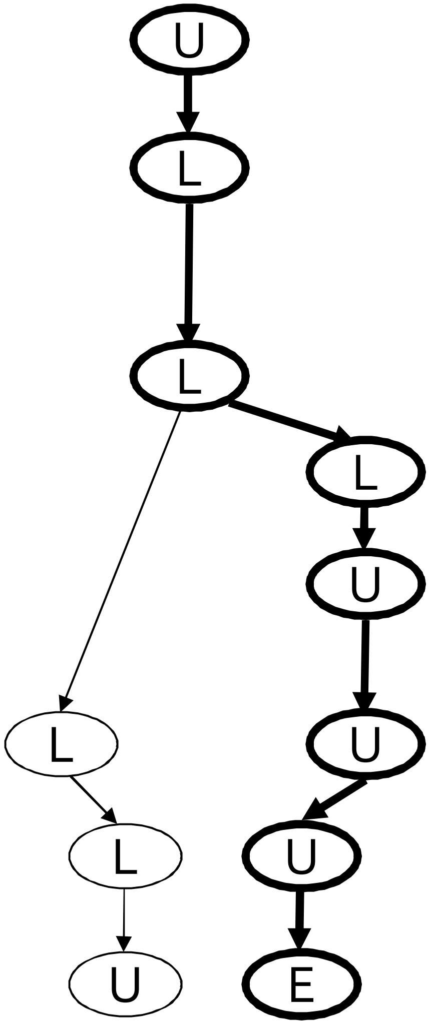


```
do {
   KeAcquireSpinLock();

   nPacketsOld = nPackets;

   if(request){
        request = request->Next;
        KeReleaseSpinLock();
        nPackets++;
   }
} while (nPackets != nPacketsOld);

KeReleaseSpinLock();
```

Example (SLAM)

b : (nPacketsOld == nPackets)

Add new predicate to boolean program (c2bp)

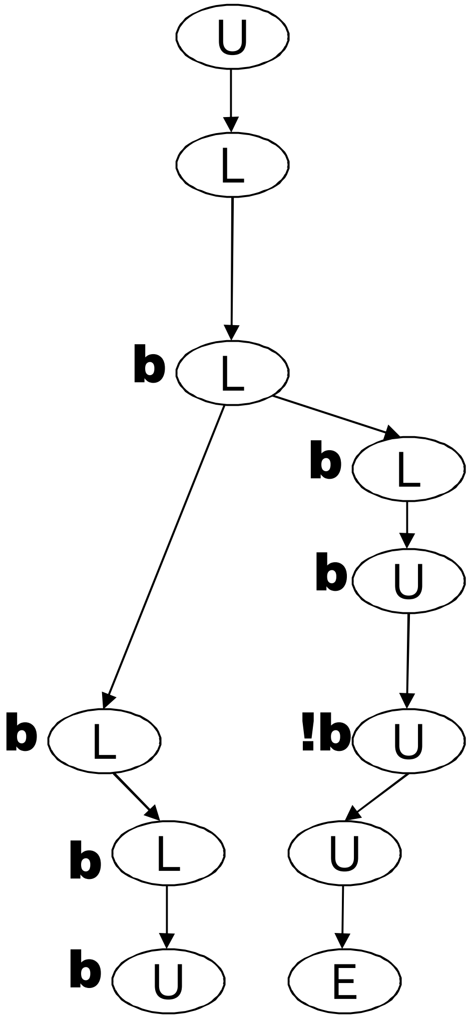


```
do {
  KeAcquireSpinLock();

  nPacketsOld = nPackets; b = true;

  if(request){
      request = request->Next;
      KeReleaseSpinLock();
      nPackets++; b = (b ? false : *);
  }
} while (nPackets != nPacketsOld);  !b

KeReleaseSpinLock();
```

Example (SLAM)

Model checking
refined
boolean program
(bebop)

b : (nPacketsOld == nPackets)

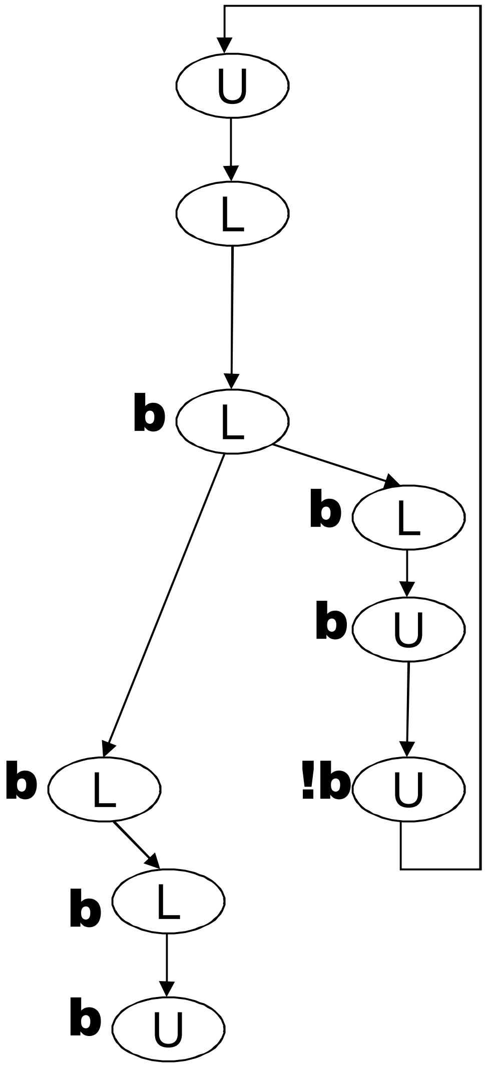


```
do {
   KeAcquireSpinLock();

   b = true;

   if(*){

        KeReleaseSpinLock();
        b = b ? false : *;
   }
} while ( !b );

KeReleaseSpinLock();
```

Example (SLAM)

Model checking
refined
boolean program
(bebop)

b : (nPacketsOld == nPackets)



```
do {
  KeAcquireSpinLock();

  b = true;

  if(*){

      KeReleaseSpinLock();
      b = b ? false : *;
  }
} while ( !b );

KeReleaseSpinLock();
```

c2bp: Predicate Abstraction for C Programs

Given

- P : a C program
- F = {e_1,…,e_n}
  - each e_i a pure boolean expression
  - each e_i represents set of states for which e_i is true

Produce a *boolean program* B(P,F)

- same control-flow structure as P
- boolean vars {b_1,…,b_n} to match {e_1,…,e_n}
- soundness: properties true of B(P,F) are true of P

Assumptions

Given

- P : a C program
- F = $\{e_1,\ldots,e_n\}$
  - each e_i a pure boolean expression
  - each e_i represents set of states for which e_i is true
- Assume: each e_i uses either:
  - only globals (global predicate)
  - local variables from some procedure (local predicate for that procedure)
- Mixed predicates:
  - predicates using both local variables and global variables
  - complicate “return” processing
  - advanced… we won’t cover it

C2bp Algorithm

- Performs modular abstraction
  - abstracts each procedure in isolation
- Within each procedure, abstracts each statement in isolation
  - no control-flow analysis
  - no need for loop invariants

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```

```
void cmp (int a , int b) {
  goto L1, L2

  L1: assume(a==b);
      g = 0;
      return;

 L2: assume(a!=b);
      g = 1;
      return;
}
```

Preds: {x==y}

{g==0}

{a==b}

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```

```
void cmp (int a , int b) {
  goto L1, L2

  L1: assume(a==b);
      g = 0;
      return;

  L2: assume(a!=b);
      g = 1;
      return;
}
```

```
decl {g==0} ;

main( {x==y} ) {




}
```

Preds: {x==y}

{g==0}

{a==b}

```
void cmp ( {a==b} ) {




}
```

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```

```
void cmp (int a , int b) {
  goto L1, L2

  L1: assume(a==b);
      g = 0;
      return;

 L2: assume(a!=b);
      g = 1;
      return;
}
```

Preds: {x==y}

{g==0}

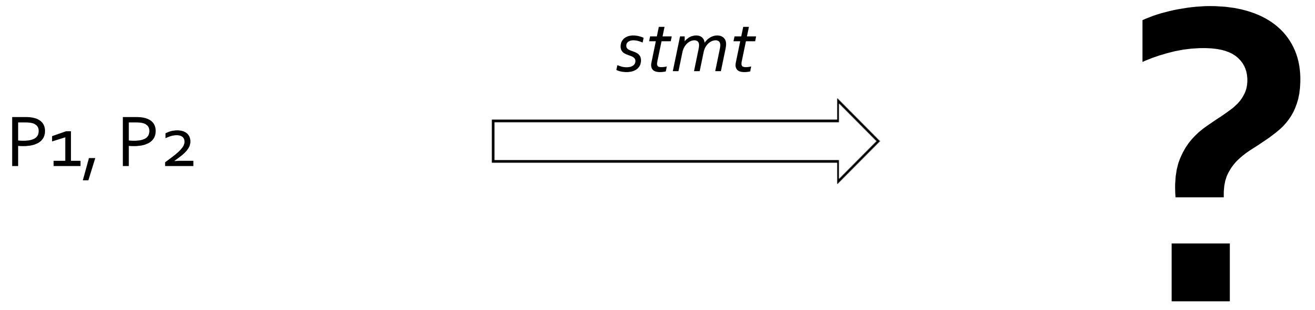
{a==b}

```
decl {g==0} ;

main( {x==y} ) {

  cmp( {x==y} );

  assume( {g==0} );
  assume( !{x==y} );
  assert(0);
}
```

```
void cmp ( {a==b} ) {
  goto L1, L2;

  L1: assume( {a==b} );
      {g==0} = T;
      return;

 L2: assume( !{a==b} );
      {g==0} = F;
      return;
}
```

Abstract Transformers?

abstract state $S^{\#}$



P1, P2

stmt

?

which predicates should hold in the resulting abstract state?

Abstract Transformers?

abstract state $S^\#$

$$P_1 \xrightarrow{[\![y = y + 1]\!]^\#} \ ?$$

vocabulary = {
$p_1 = (y < 4)$,
$p_2 = (y < 5)$
}

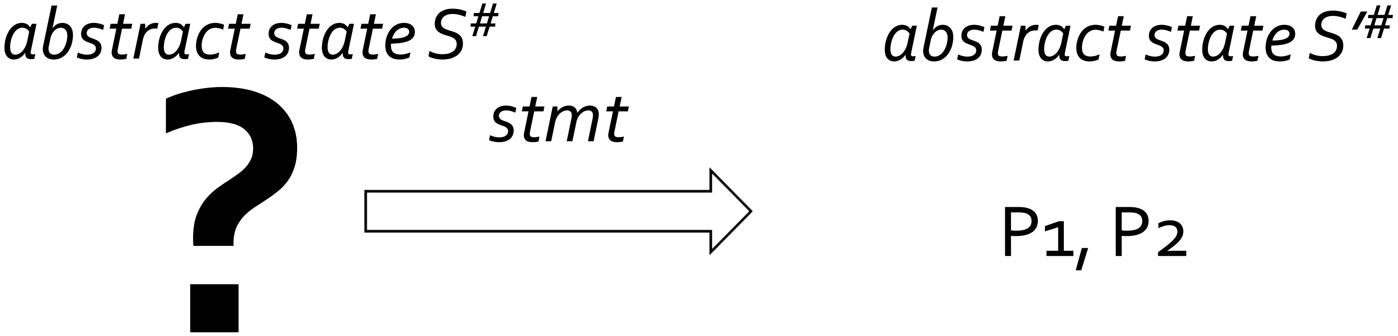
Abstract Transformers?

$$\begin{tikzcd}
(y < 4) \arrow[r, "{[\![y = y + 1]\!]}"] \arrow[d, leftarrow] & (y < 5) \arrow[d] \\
P1 \arrow[r, "{[\![y = y + 1]\!]^{\#}}"] & P2
\end{tikzcd}$$

p1 = (y < 4), p2 = (y < 5)

Can also phrase it with WP

- To check if a predicate should hold in the next abstract state
- check the weakest precondition of the predicate’s defining formula wrt statement



abstract state $S^\#$

?

stmt

abstract state $S'^\#$

P1, P2

which predicates should hold in $S^\#$ *so P1,P2 hold in* $S'^\#$*?*

Abstracting Assigns via WP

- WP(x=e,Q) = Q[x $\mapsto$ e]

- Effect of statement y = y + 1
- Vocabulary = { y < 4, y < 5 }

- WP(y=y+1, y<5) =

  (y<5) [y $\mapsto$ y+1] =

  (y+1<5) =

  (y<4)

WP Problem

- WP(s, e_i) not always expressible via { e_1,…,e_n }
- (Vocabulary not closed under WP)

- Example
  - F = { x==0, x==1, x<5 }
  - WP(x=x+1 , x<5) = x<4
  - Best possible: x==0 || x==1

Transformers via WP

$$\begin{tikzcd}
(x < 4) \arrow[r, Rightarrow, "wp(x=x+1{,}x<5)"] \arrow[d, Rightarrow] & (x < 5) \\
\text{No matching predicates} \arrow[r, Rightarrow, "\llbracket x = x + 1 \rrbracket^{\#}"] & P3 \arrow[u, Rightarrow]
\end{tikzcd}$$

$p1 = (x==0), p2 = (x==1), p3 = (x < 5)$

Transformers via WP

$$\begin{tikzcd}
(x < 4) \arrow[r, "wp(x=x+1{,}x<5)"] \arrow[d] & (x < 5) \\
\begin{array}{l}\text{idea:}\\ (x==0) \text{ or } (x==1)\\ \textbf{imply} \text{ that}\\ (x<4)\end{array} \arrow[r, "{[\![x = x + 1]\!]^{\#}}"] & P3 \arrow[u]
\end{tikzcd}$$

$p1 = (x==0), p2 = (x==1), p3 = (x < 5)$

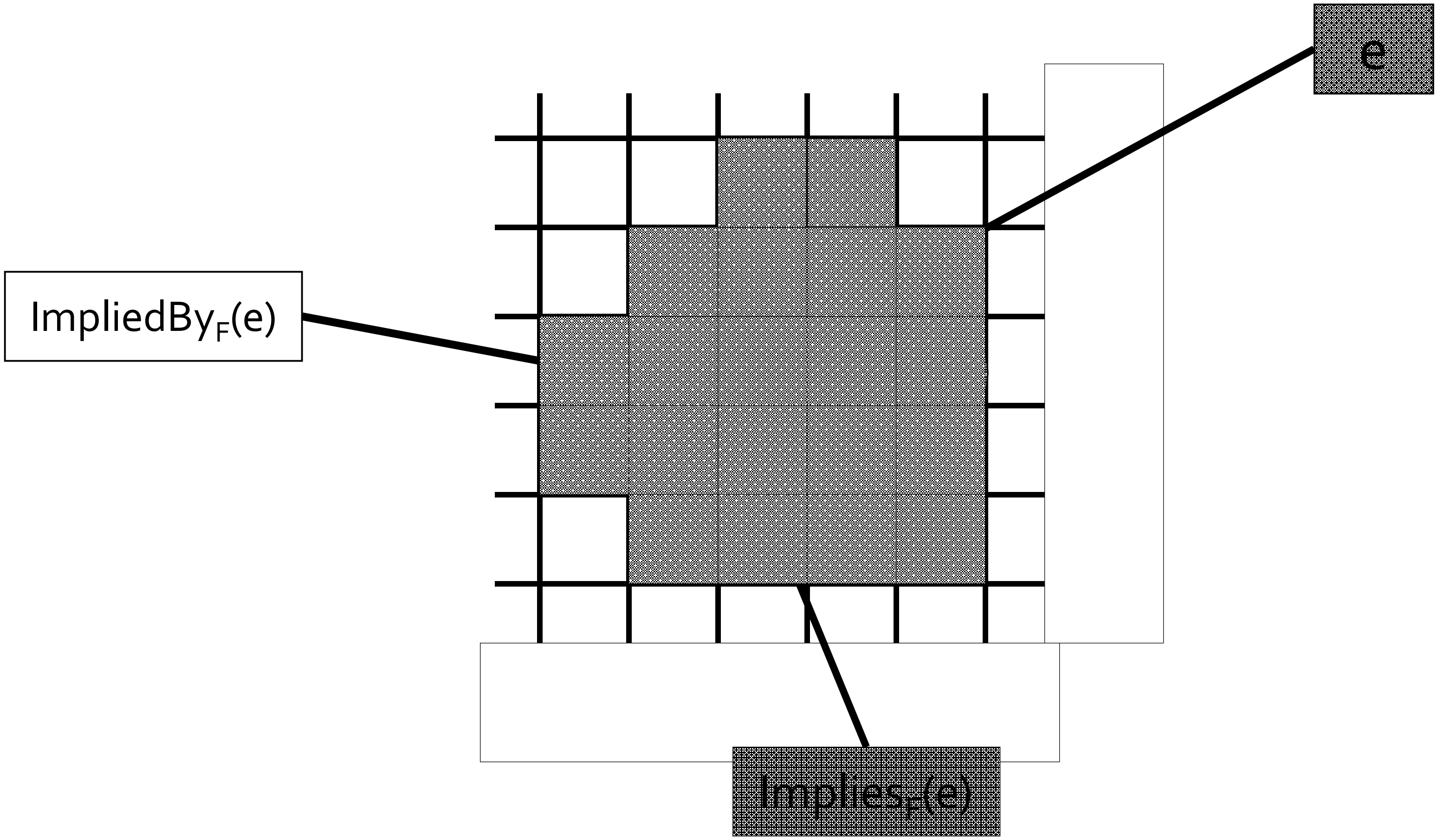
Abstracting Expressions via F

- $F = \{ e_1, \ldots, e_n \}$

- $\text{Implies}_F(e)$
  - *best* boolean function over F *that implies* e

- $\text{ImpliedBy}_F(e)$
  - *best* boolean function over F *implied by* e
  - $\text{ImpliedBy}_F(e) = !\text{Implies}_F(!e)$

$\texttt{Implies}_F\texttt{(e)}$ and $\texttt{ImpliedBy}_F\texttt{(e)}$



Computing $\mathtt{Implies}_F(\mathtt{e})$

- *minterm m* = d_1 && … && d_n
  - where $d_i = e_i$ or $d_i = !e_i$

- $\text{Implies}_F(e)$
  - disjunction of all minterms that imply e

- Naïve approach
  - generate all 2^n possible minterms
  - for each minterm m, **use decision procedure to check *validity* of each implication** $m \Rightarrow e$

- Many optimizations possible

Abstracting Assignments

- if $\text{Implies}_F(WP(s, e_i))$ is true before s then
  - e_i is true after s

- if $\text{Implies}_F(WP(s, !e_i))$ is true before s then
  - e_i is false after s

```
{e_i} = Implies_F(WP(s, e_i))    ?    true :
        Implies_F(WP(s, !e_i))   ?    false
                                 :    *;
```

Assignment Example

Statement in P:

y = y+1;

Predicates in E:

{x==y}

Weakest Precondition:

WP(y=y+1, x==y) = x==y+1

Implies_F(x==y+1) = false

Implies_F(x!=y+1) = x==y

Abstraction of assignment in B:

{x==y} = ({x==y} ? false : *);

Absracting Assumes

- WP(assume(e) , Q) = $e \Rightarrow Q$

- assume(e) is abstracted to:

  assume($\mathrm{ImpliedBy}_F$(e))

- Example:

  F = {x==2, x<5}

  assume(x < 2) is abstracted to:

  assume({x<5} && !{x==2})

Assignments + Pointers

Statement in P:

*p = 3

Predicates in E:

{x==5}

Weakest Precondition:

WP(*p=3 , x==5) = x==5

What if *p and x alias?

Correct Weakest Precondition:

(p==&x and 3==5) or (p!=&x and x==5)

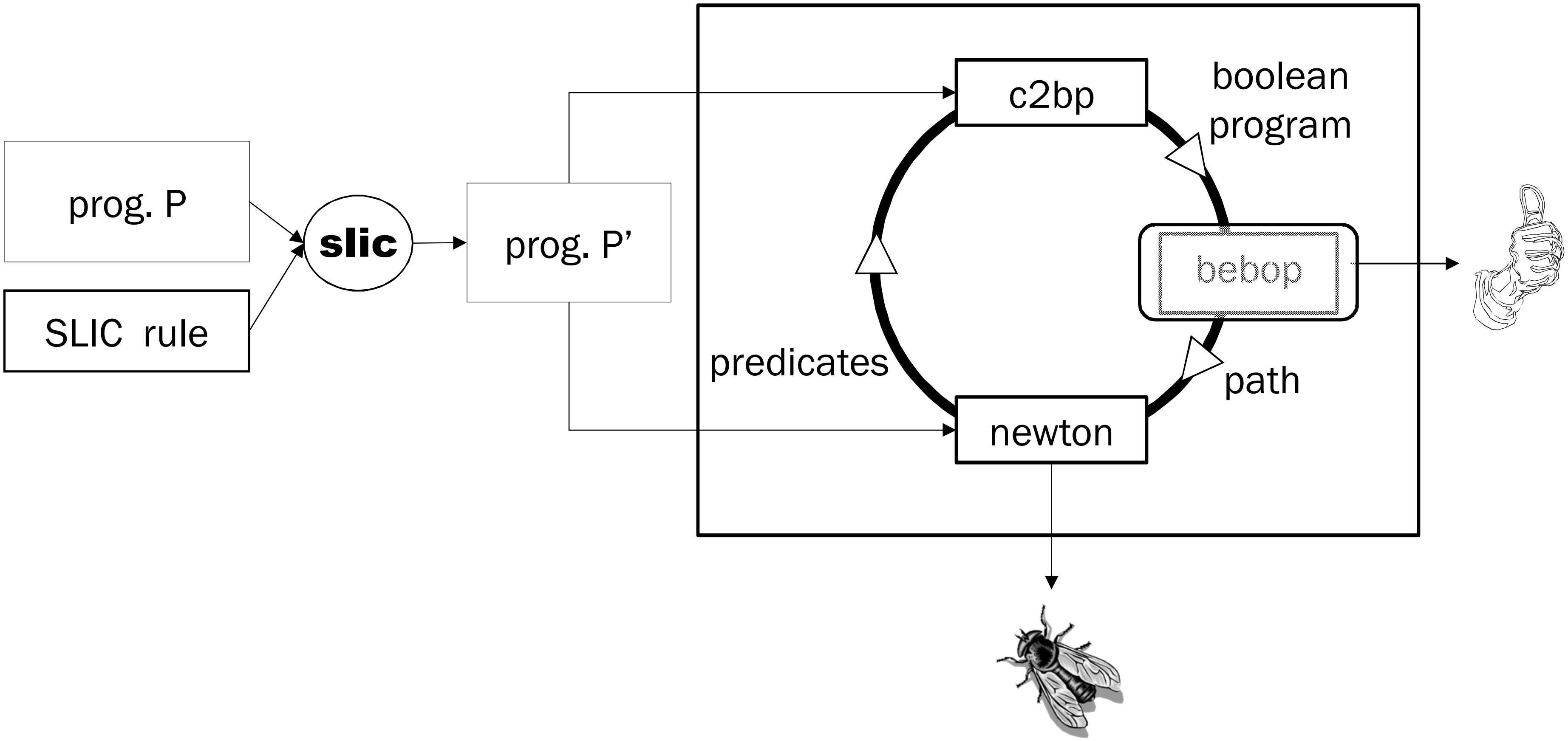
Precision

- For program P and E = $\{e_1,\ldots,e_n\}$, there exist two "ideal" abstractions:
  - Boolean(P,E): most precise abstraction
  - Cartesian(P,E): less precise abtraction, where each boolean variable is updated independently
  - [See Ball-Podelski-Rajamani, TACAS 00]

- Theory
  - with an "ideal" theorem prover, c2bp can compute Cartesian(P,E)

- Practice
  - c2bp computes a less precise abstraction than Cartesian(P,E)
  - we use Das/Dill's technique to incrementally improve precision
  - with an "ideal" theorem prover, the combination of c2bp + Das/Dill can compute Boolean(P,E)

The SLAM Process



prog. P

SLIC rule

slic

prog. P’

c2bp

boolean program

bebop

path

newton

predicates

Bebop

- Model checker for boolean programs

- Based on CFL reachability
    - [Sharir-Pnueli 81] [Reps-Sagiv-Horwitz 95]

- Iterative addition of edges to graph
    - “path edges”: <entry,d1> → <v,d2>
    - “summary edges”: <call,d1> → <ret,d2>

Symbolic CFL reachability

- Partition path edges by their “target”
  - PE(v) = { <d1,d2> | <entry,d1> → <v,d2> }

- What is <d1,d2> for boolean programs?
  - A bit-vector!

- What is PE(v)?
  - A set of bit-vectors

- Use a BDD (attached to v) to represent PE(v)

BDDs

- Canonical representation of
  - boolean functions
  - set of (fixed-length) bitvectors
  - binary relations over finite domains
- Efficient algorithms for common dataflow operations
  - transfer function
  - join/meet
  - subsumption test

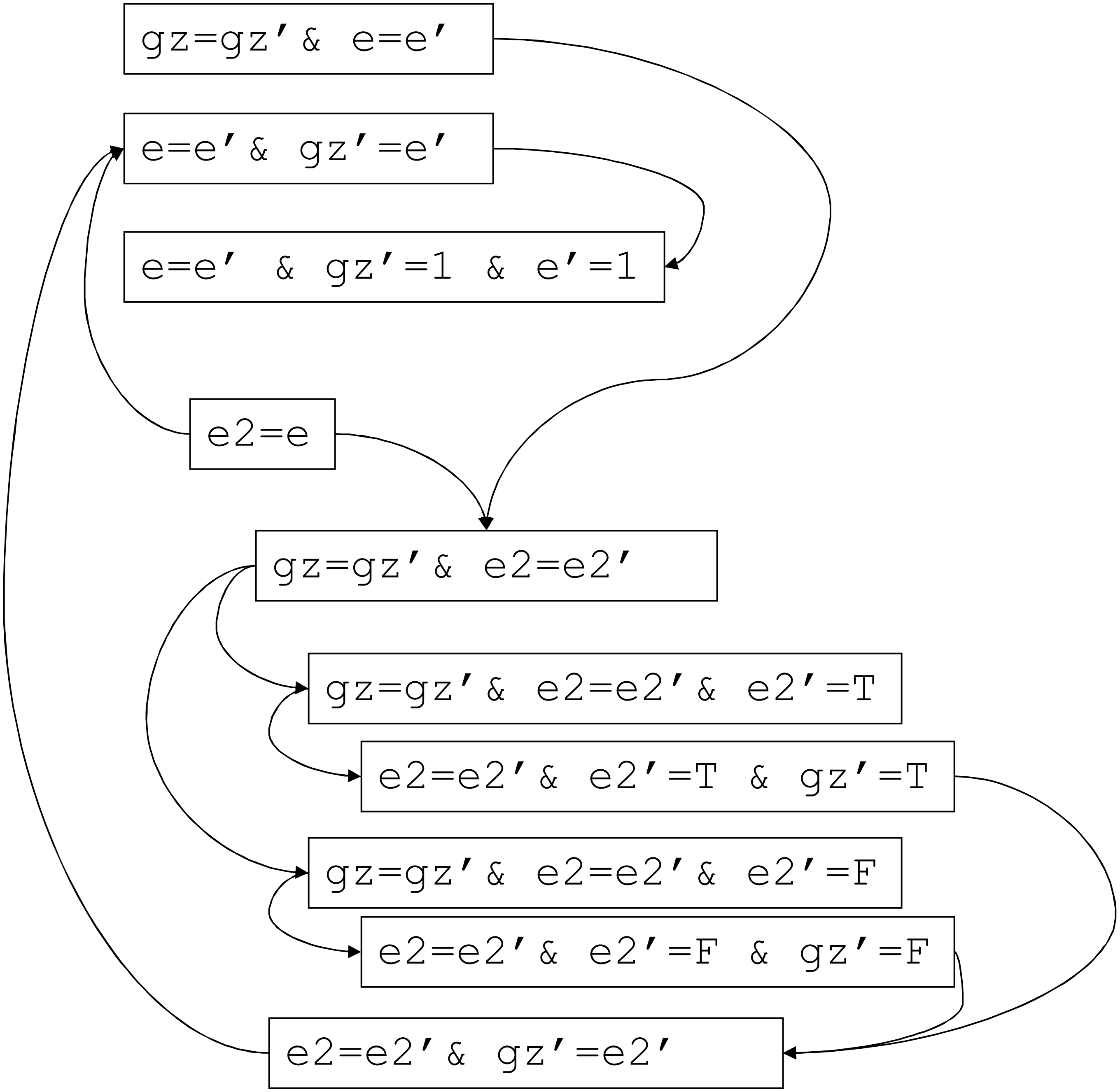
```
void cmp ( e2 ) {
[5]Goto L1, L2
  [6]L1: assume( e2 );
  [7] gz = T; goto L3;

  [8]L2: assume( !e2 );
  [9]gz = F; goto L3

[10] L3: return;
}
```

BDD at line [10] of cmp:

```
e2=e2’ & gz’=e2’
```

Read: “cmp leaves e2 unchanged and sets gz to have the same final value as e2”

```
   decl gz ;
   main( e ) {

    [1] cmp( e );

    [2] assume( gz );

    [3] assume( !e );

    [4] assert(F);
   }

void cmp ( e2 ) {
[5]Goto L1, L2

  [6]L1: assume( e2 );

  [7] gz = T; goto L3;

  [8]L2: assume( !e2 );

  [9]gz = F; goto L3

[10] L3: return;
}
```



gz=gz’& e=e’

e=e’& gz’=e’

e=e’ & gz’=1 & e’=1

e2=e

gz=gz’& e2=e2’

gz=gz’& e2=e2’& e2’=T

e2=e2’& e2’=T & gz’=T

gz=gz’& e2=e2’& e2’=F

e2=e2’& e2’=F & gz’=F

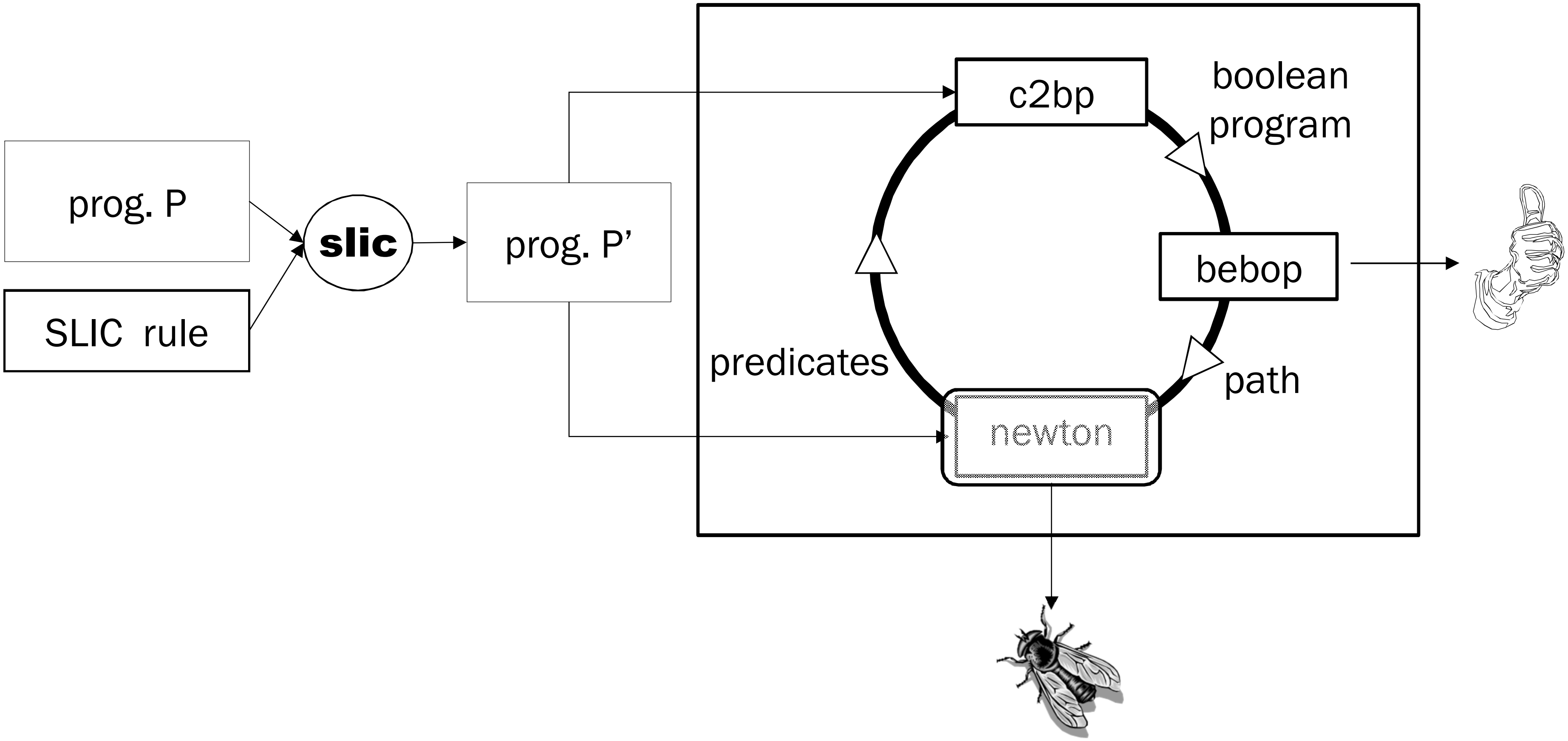
e2=e2’& gz’=e2’

Bebop: summary

- Explicit representation of CFG
- Implicit representation of path edges and summary edges
- Generation of hierarchical error traces

- Complexity: $O(E * 2^{O(N)})$
  - E is the size of the CFG
  - N is the max. number of variables in scope

The SLAM Process



prog. P

SLIC rule

slic

prog. P'

c2bp

boolean program

bebop

path

newton

predicates

Newton

- Given an error path p in boolean program B
    - is p a feasible path of the corresponding C program?
        - Yes: found an error
        - No: find predicates that explain the infeasibility
    - uses the same interfaces to the theorem provers as c2bp.

Newton

- Execute path symbolically
- Check conditions for inconsistency using theorem prover (satisfiability)
- After detecting inconsistency
  - minimize inconsistent conditions
  - traverse dependencies
  - obtain predicates

Symbolic simulation for C--

Domains

- variables: names in the program
- values: constants + symbols

State of the simulator has 3 components:

- store: map from variables to values
- conditions: predicates over symbols
- history: past valuations of the store

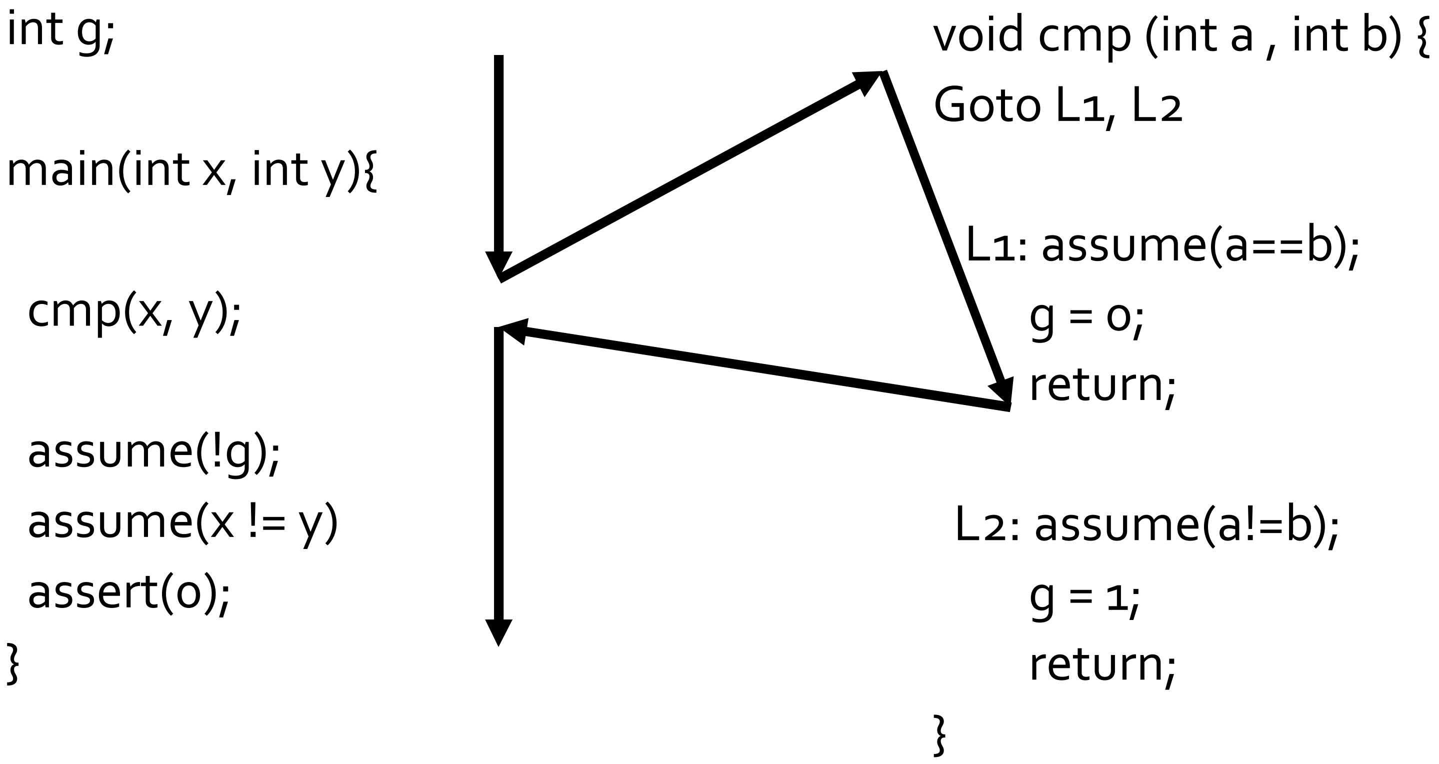
Symbolic simulation Algorithm

Input: path ***p***

```
For each statement s in p do
    match s with
        Assign(x,e):
                let val = Eval(e) in
                if (Store[x]) is defined then
                        History[x] := History[x] ⊕ Store[x]
                Store[x] := val

        Assume(e):
                let val = Eval(e) in
                        Cond := Cond and val
                        let result = CheckConsistency(Cond) in
                        if (result  == “inconsistent”) then
                                GenerateInconsistentPredicates()
End

Say “Path p is feasible”
```

Symbolic Simulation: Caveats

- Procedure calls
  - add a stack to the simulator
  - push and pop stack frames on calls and returns
  - implement mappings to keep values “in scope” at calls and returns
- Dependencies
  - for each condition or store, keep track of which values were used to generate this value
  - traverse dependency during predicate generation



```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```

```
void cmp (int a , int b) {
Goto L1, L2

  L1: assume(a==b);
        g = 0;
        return;

  L2: assume(a!=b);
        g = 1;
        return;
}
```

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```

Global:

main:

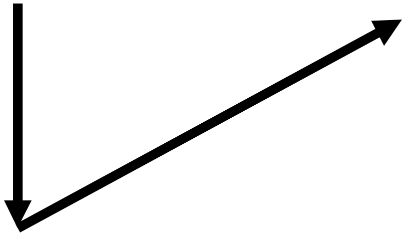
(1) x: X

(2) y: Y

```
void cmp (int a , int b) {
Goto L1, L2

  L1: assume(a==b);
       g = 0;
       return;

 L2: assume(a!=b);
       g = 1;
       return;
}
```

Conditions:

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```



```
void cmp (int a , int b) {
Goto L1, L2

  L1: assume(a==b);
       g = 0;
       return;

  L2: assume(a!=b);
       g = 1;
       return;
}
```

Global:

main:

(1) x: X

(2) y: Y

cmp:

(3) a: A

(4) b: B

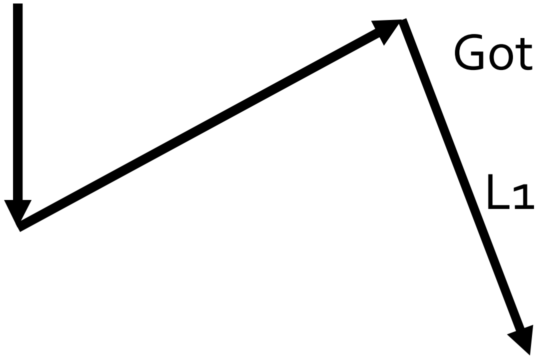
Map:

$X \rightarrow A$

$Y \rightarrow B$

Conditions:

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```



```
void cmp (int a , int b) {
Goto L1, L2

 L1: assume(a==b);
     g = 0;
     return;

 L2: assume(a!=b);
     g = 1;
     return;
}
```

Global:

(6) g: 0

main:

(1) x: X

(2) y: Y

cmp:

(3) a: A

(4) b: B

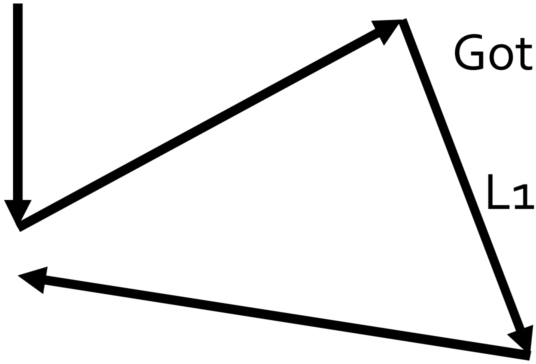
Map:

$X \rightarrow A$

$Y \rightarrow B$

Conditions:

(5) (A == B) [3, 4]



```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```

```
void cmp (int a , int b) {
Goto L1, L2

 L1: assume(a==b);
      g = 0;
      return;

 L2: assume(a!=b);
      g = 1;
      return;
}
```

Global:

(6) g: 0

main:

(1) x: X

(2) y: Y

cmp:

(3) a: A

(4) b: B

Map:

X → A

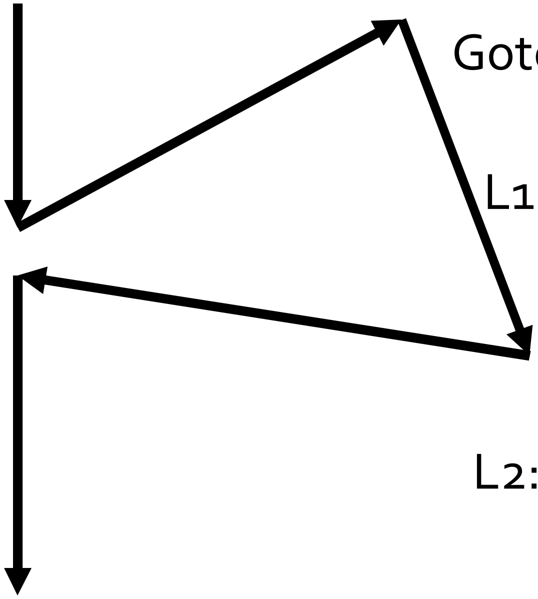
Y → B

Conditions:

(5) (A == B) [3, 4]

(6) (X == Y) [5]

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```



```
void cmp (int a , int b) {
Goto L1, L2

  L1: assume(a==b);
        g = 0;
        return;

  L2: assume(a!=b);
        g = 1;
        return;
}
```

Global:

(6) g: 0

main:

(1) x: X

(2) y: Y

cmp:

(3) a: A

(4) b: B

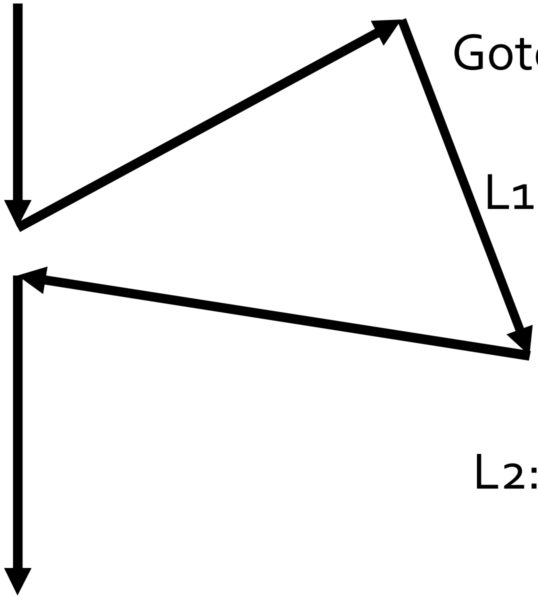
Conditions:

(5) (A == B) [3, 4]

(6) (X == Y) [5]

(7) (X != Y) [1, 2]

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```



```
void cmp (int a , int b) {
Goto L1, L2

  L1: assume(a==b);
      g = 0;
      return;

  L2: assume(a!=b);
      g = 1;
      return;
}
```

Global:

(6) g: 0

main:

(1) x: X

(2) y: Y

cmp:

(3) a: A

(4) b: B

Conditions:

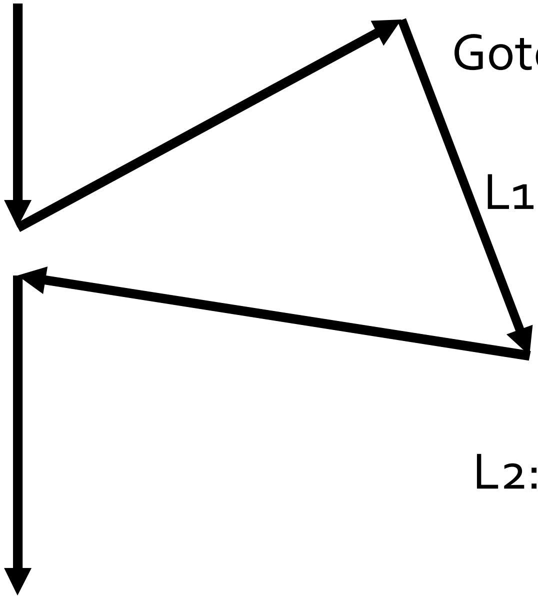
(5) (A == B) [3, 4]

(6) (X == Y) [5]

(7) (X != Y) [1, 2]

Contradictory!

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```



```
void cmp (int a , int b) {
Goto L1, L2

  L1: assume(a==b);
      g = 0;
      return;

  L2: assume(a!=b);
      g = 1;
      return;
}
```

Global:

(6) g: 0

main:

(1) x: X

(2) y: Y

cmp:

(3) a: A

(4) b: B

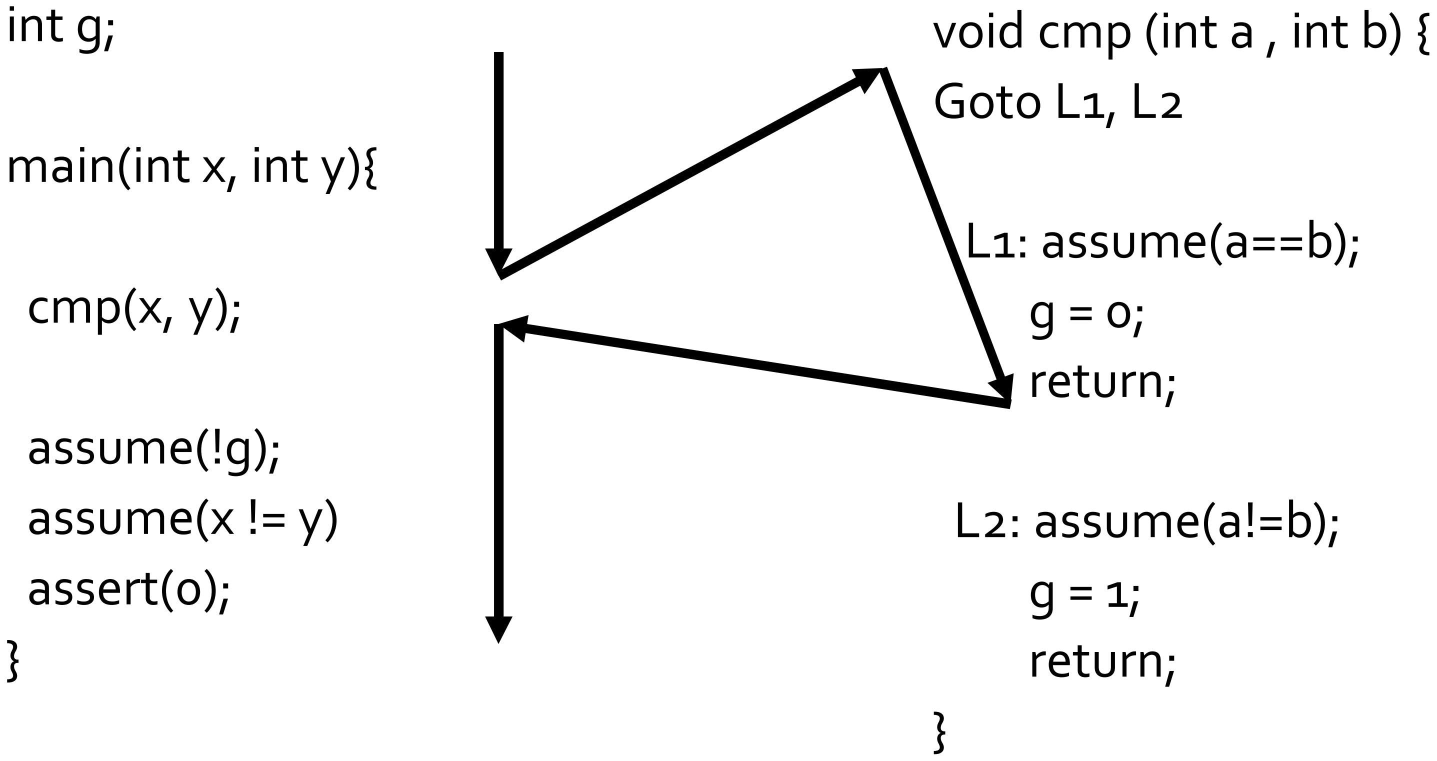
Conditions:

(5) (A == B) [3, 4]

(6) (X == Y) [5]

(7) (X != Y) [1, 2]

Contradictory!



```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```

```
void cmp (int a , int b) {
Goto L1, L2

  L1: assume(a==b);
        g = 0;
        return;

  L2: assume(a!=b);
        g = 1;
        return;
}
```

Predicates after simplification:

{ x == y, a == b }

Recap

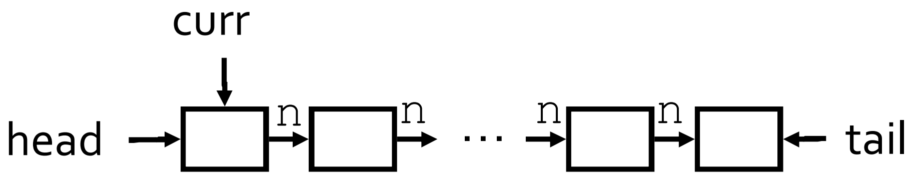
- Predicate Abstraction (via SLAM)
- CEGAR (without the details)

PREDICATE ABSTRACTION AND CANONICAL ABSTRACTION FOR SINGLY-LINKED LISTS

[Roman Manevich et. al. VMCAI'05]

Motivating Example 1: CEGAR

```
curr = head;
while (curr != tail) {
    assert (curr != null);
    curr = curr.n;
}
```

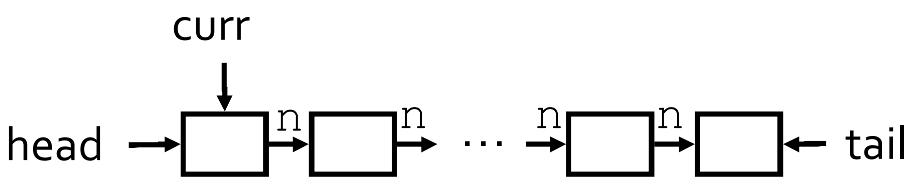


CEGAR generates following predicates:

curr.n ≠ null ,
curr.n.n ≠ null ,
… after i refinement steps:
$curr(.n)^i \neq null$

Motivating Example 1: CEGAR

```
curr = head;
while (curr != tail) {
    assert (curr != null);
    curr = curr.n;
}
```



In general, problem is undecidable:
V. Chakaravathy [POPL 2003]

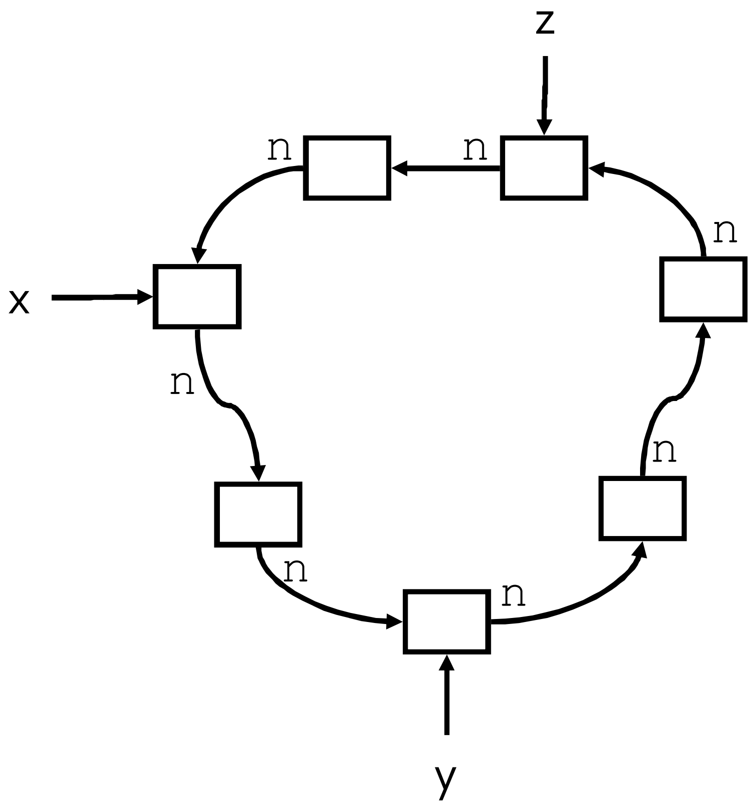
State-of-the-art canonical abstractions can prove assertion

Motivating Example 2

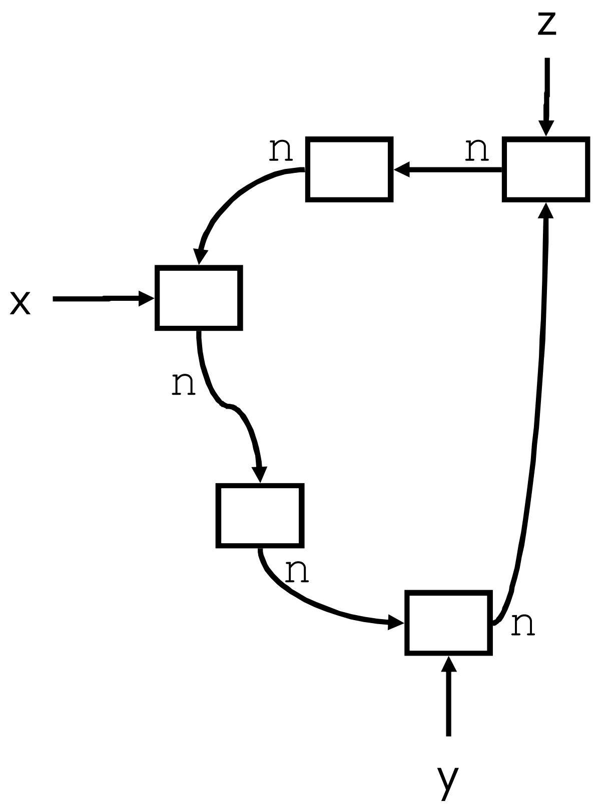
```
// @pre cyclic(x)
t = null;
y = x;
while (t != x && y.data < low) {
    t = y.n; y = t;
}
z = y;
while (z != x && z.data < high) {
    t = z.n; z = t;
}
t = null;
if (y != z) {
    y.n = null;
    y.n = z;
}
// @post cyclic(x)
```

Motivating Example 2

`@pre cyclic(x)`

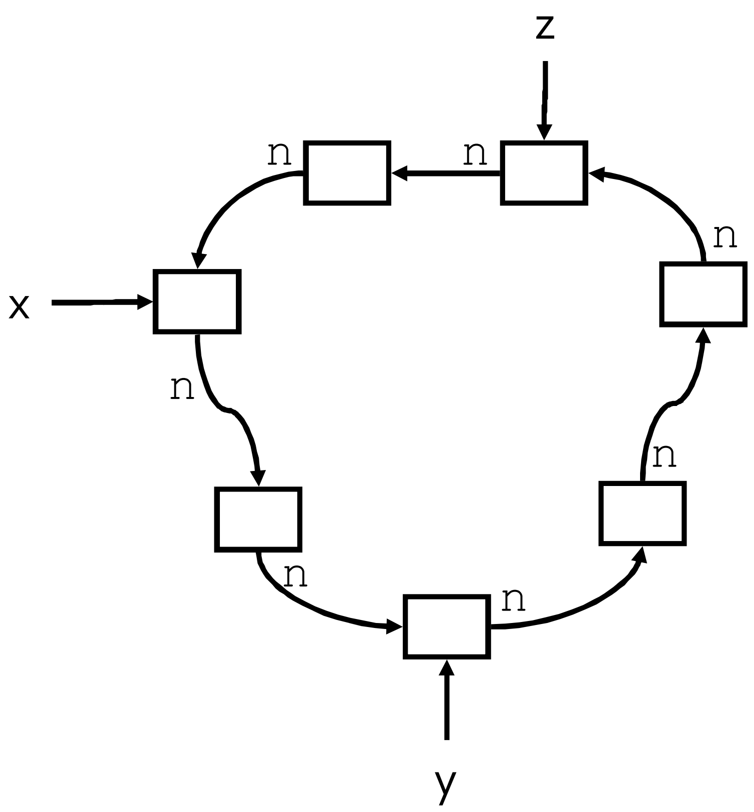


`@post cyclic(x)`

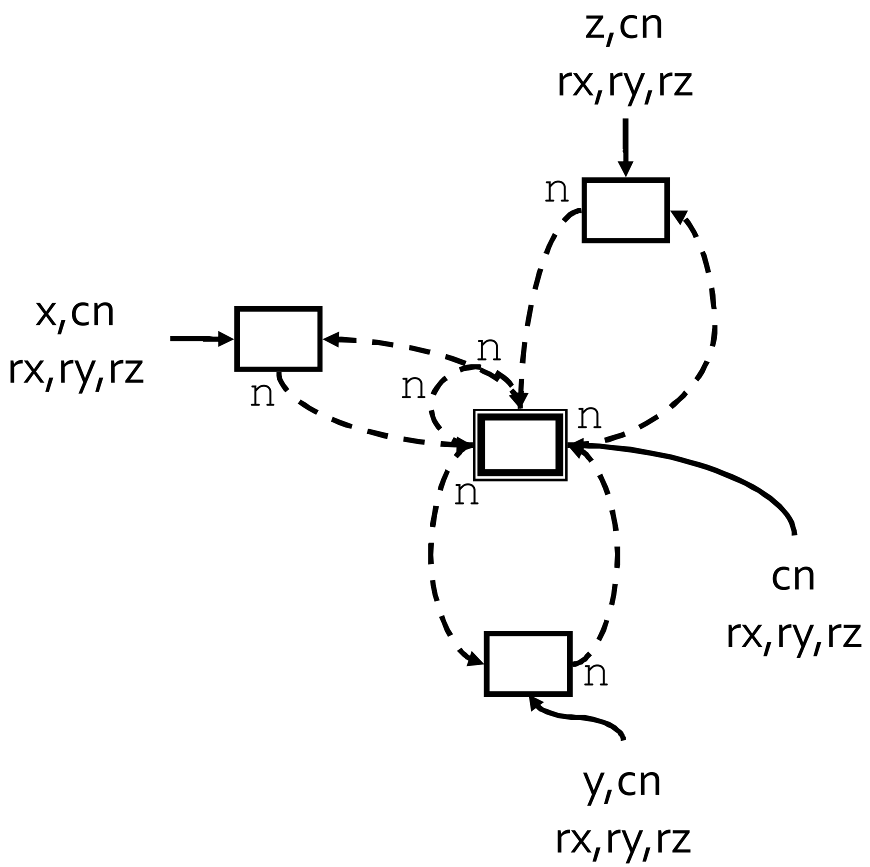


Existing Canonical Abstraction

concrete



abstract



order between variables lost!

cannot establish **`@post cyclic(x)`**

Overview and Main Results

- Current predicate abstraction refinement methods not adequate for analyzing heaps
- Predicate abstraction can simulate arbitrary finite abstract domains
  - Often requires too many predicates
- New family of abstractions for lists
  - Bounded number of sharing patterns
  - Handles cycles more precisely than existing canonical abstractions
- Encode abstraction with two methods
  - Canonical abstraction
  - **Polynomial** predicate abstraction

Outline

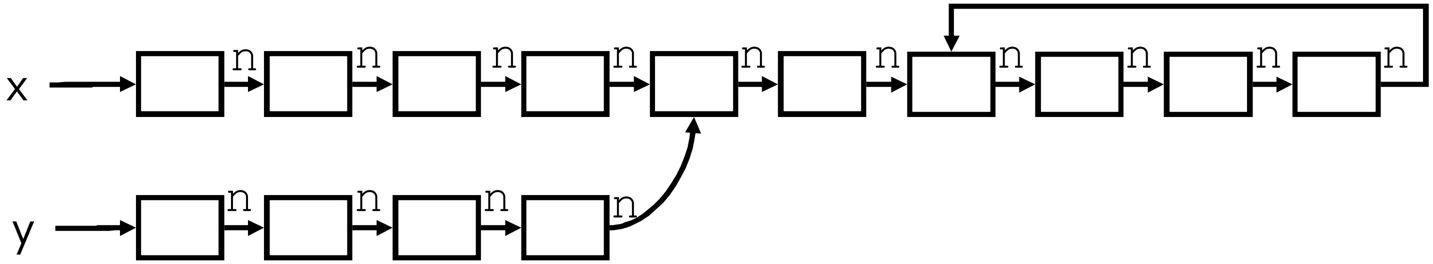
- New abstractions for lists
    - Observations on concrete shapes
    - Static naming scheme
    - Encoding via predicate abstraction
    - Encoding via canonical abstraction
    - Controlling the number of predicates via heap-sharing depth parameter
- Conclusion

Concrete Shapes

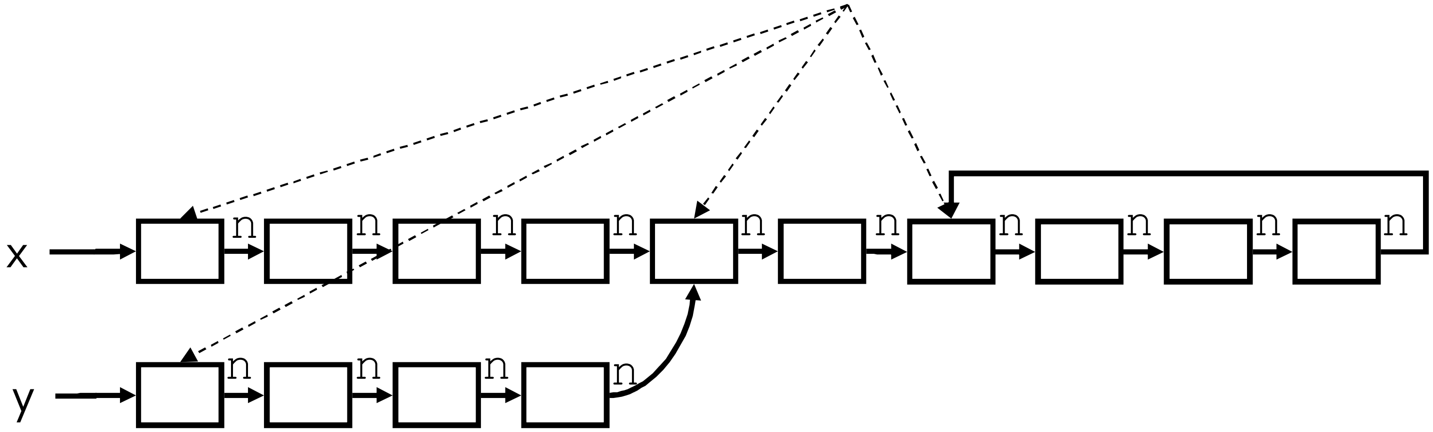
- Assume the following class of (list-) heaps
  - Heap contains only singly-linked lists
  - No garbage (easy to handle)
- A heap can be decomposed into
  - Basic shape (sharing pattern)
  - List lengths

Concrete Shapes

```
class SLL {
  Object value;
  SLL n;
}
```



Interrupting Nodes

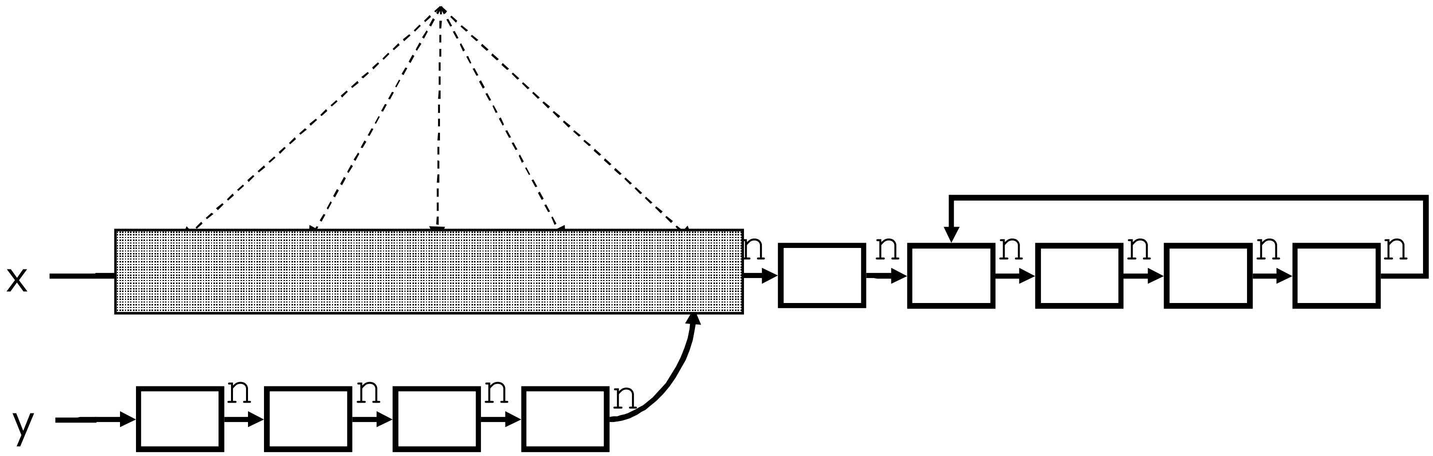


Interruption:
node pointed-to by a variable
or shared by **`n`** fields

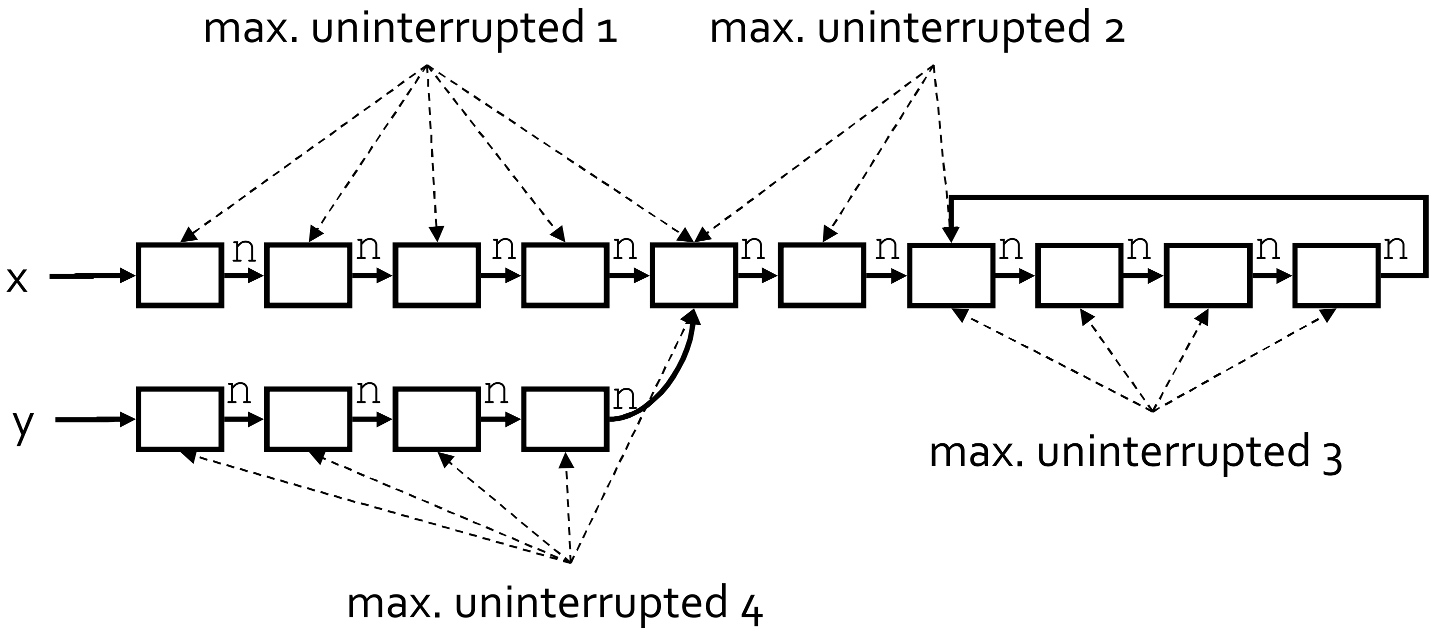
#interruptions ≤ 2 · #variables
(bounded number of sharing patterns)

Maximal Uninterrupted Lists

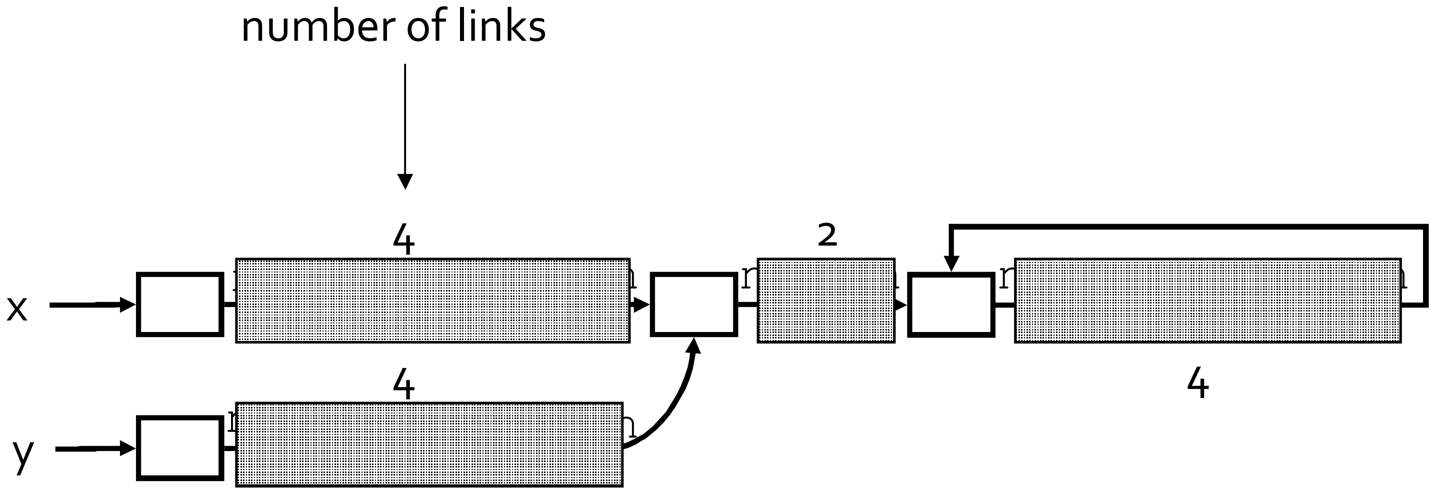
Maximal uninterrupted list:
maximal list segment between two interruptions
not containing interruptions in-between



Maximal Uninterrupted Lists

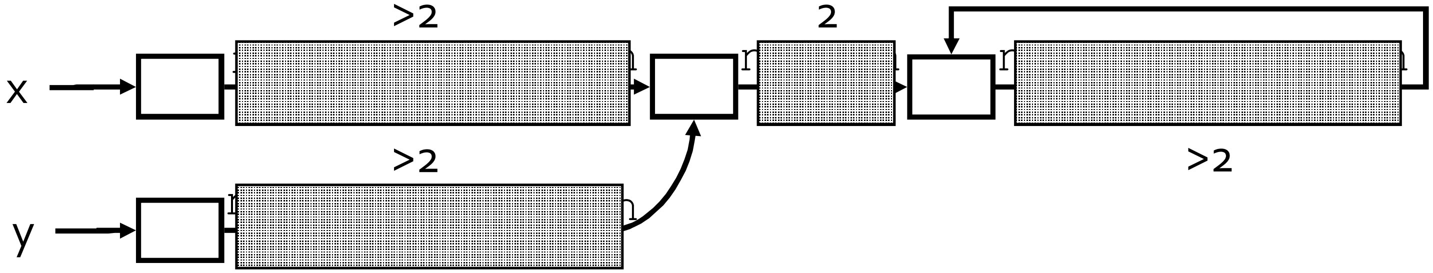


Maximal Uninterrupted Lists



Maximal Uninterrupted Lists

Abstract lengths: {1,2,>2}



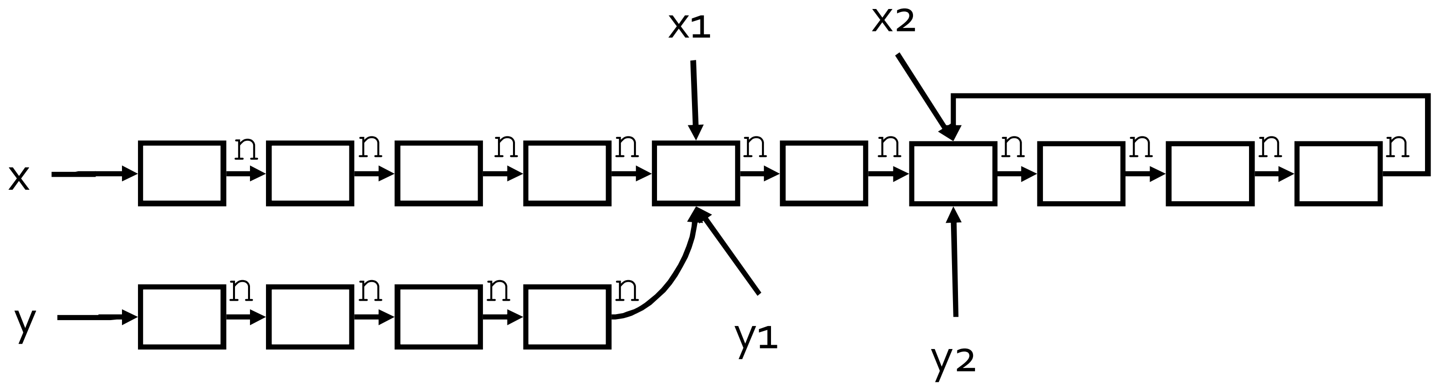
Using Static Names

- Goal: name all sharing patterns
- Prepare static names for interruptions
    - Derive predicates for canonical abstraction
- Prepare static names for max. uninterrupted lists
    - Derive predicates for predicate abstraction
- All names expressed by FO^{TC} formulae

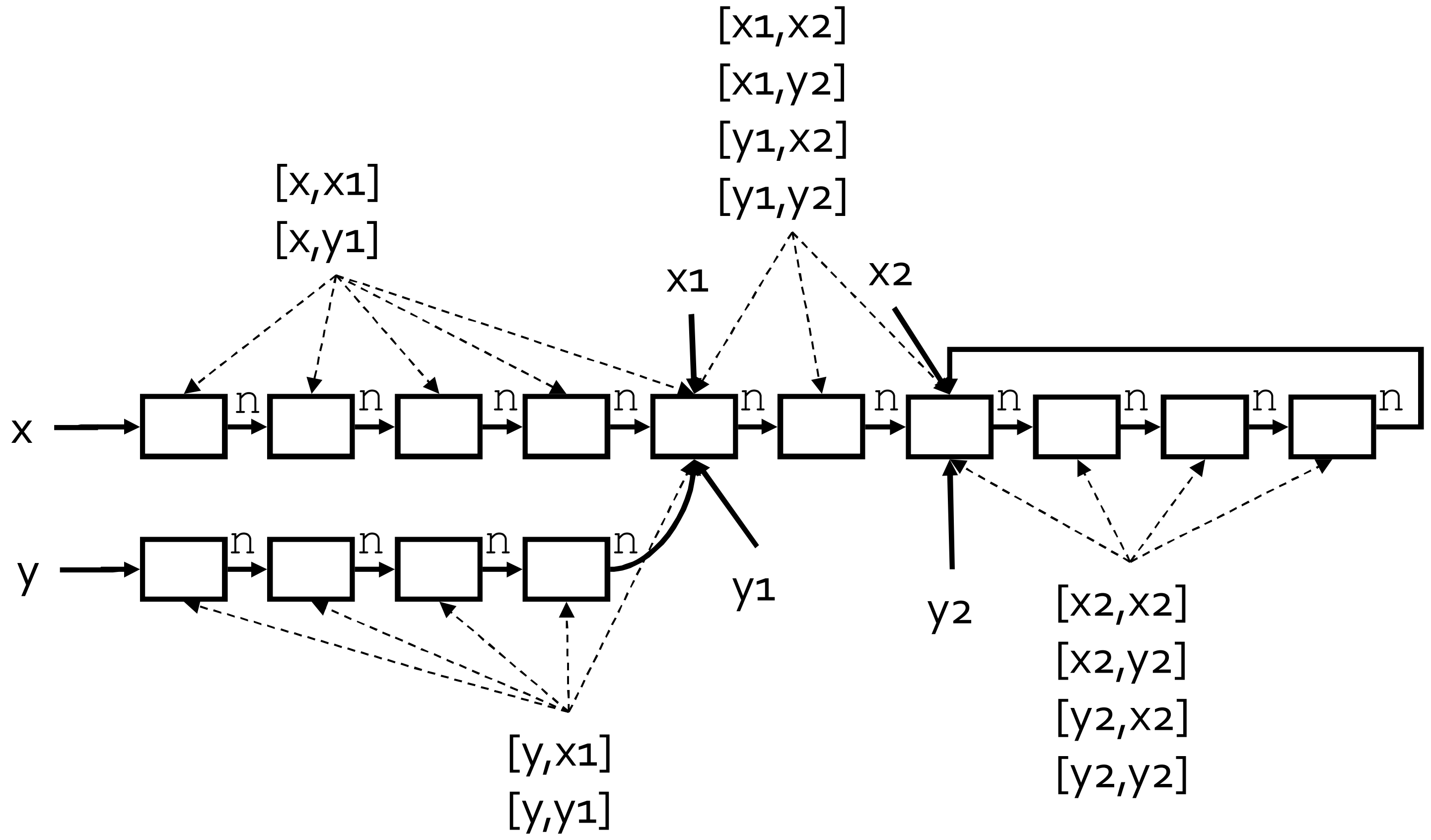
Naming Interruptions

We name interruptions by adding auxiliary variables

For every variable x : x1,…,xk (k=#variables)



Naming Max. Uninterrupted Lists

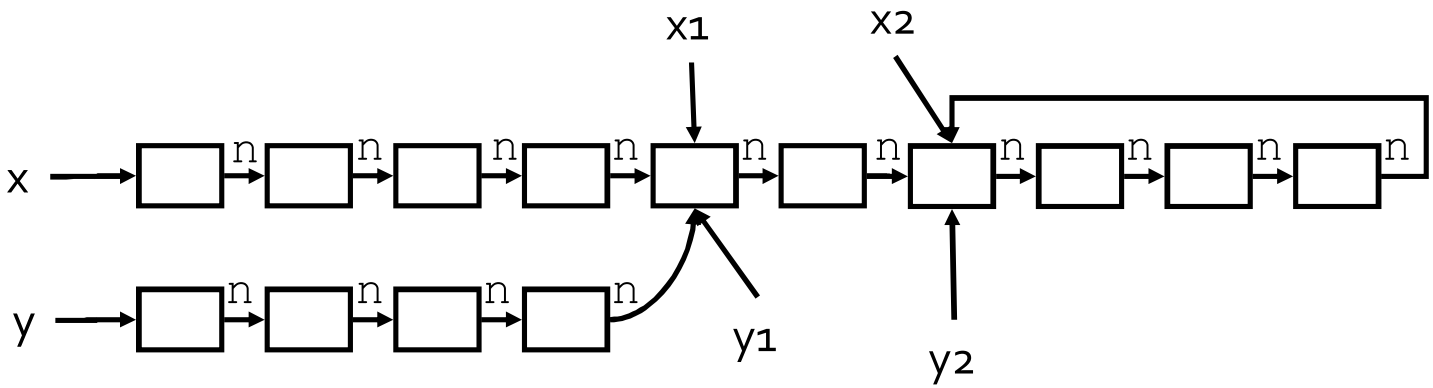


A Predicate Abstraction

- For every pair of variables x,y (regular and auxiliary)
  - **Aliased[x,y]** = x and y point to same node
  - **UList1[x,y]** = max. uninterrupted list of length 1
  - **UList2[x,y]** = max. uninterrupted list of length 2
  - **UList[x,y]** = max. uninterrupted list of any length
- For every variable x (regular and auxiliary)
  - **UList1[x,null]** = max. uninterrupted list of length 1
  - **UList2[x,null]** = max. uninterrupted list of length 2
  - **UList[x,null]** = max. uninterrupted list of any length
- Predicates expressed by FO^{TC} formulae

Predicate Abstraction Example

concrete



abstract

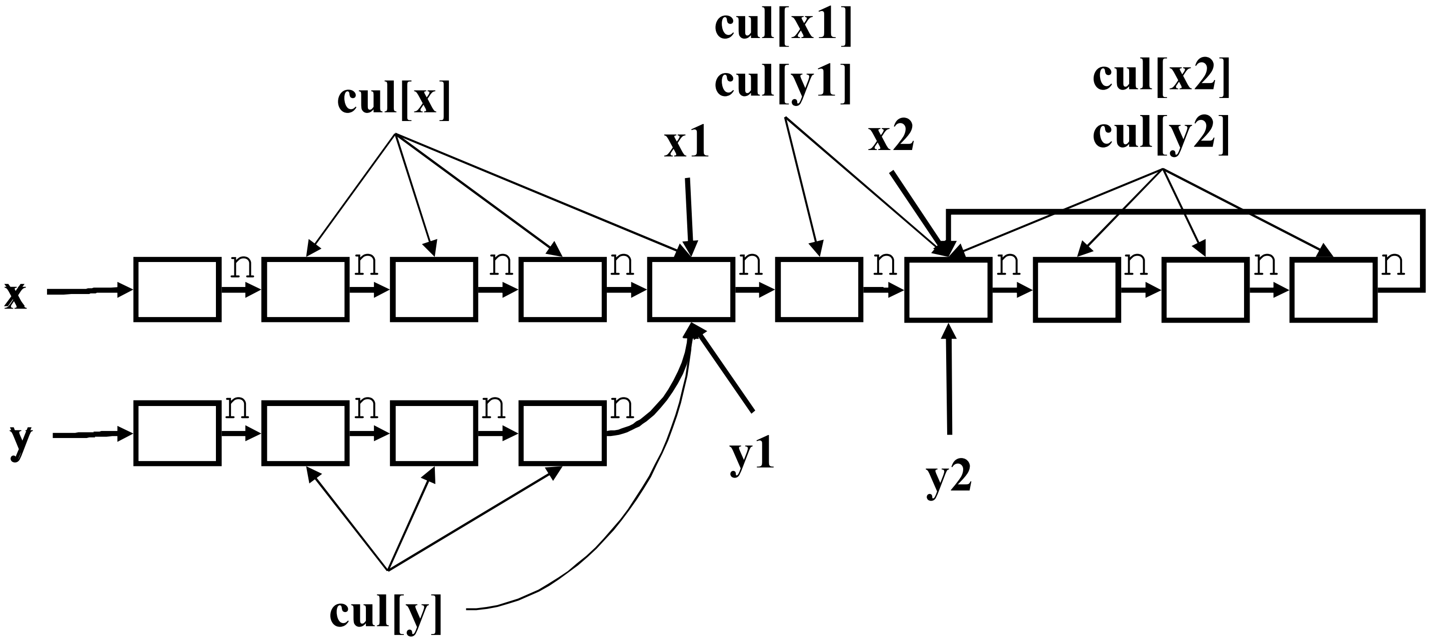
Aliased[x,x]
Aliased[y,y]
Aliased[x1,x1]
Aliased[x2,x2]
Aliased[y1,y1]
Aliased[y2,y2]
Aliased[x1,y1]
Aliased[y1,x1]
Aliased[x2,y2]
Aliased[y2,x2]
UList[x,x1]
UList[x,y1]
UList2[x1,x2]
UList2[x1,y2]
UList2[y1,x2]
UList2[y1,y2]
UList[x1,x2]
UList[x1,y2]
UList[y1,x2]
UList[y1,y2]
UList[y,x1]
UList[y,y1]
UList[x2,x2]
UList[x2,y2]
UList[y2,x2]
UList[y2,y2]

A Canonical Abstraction

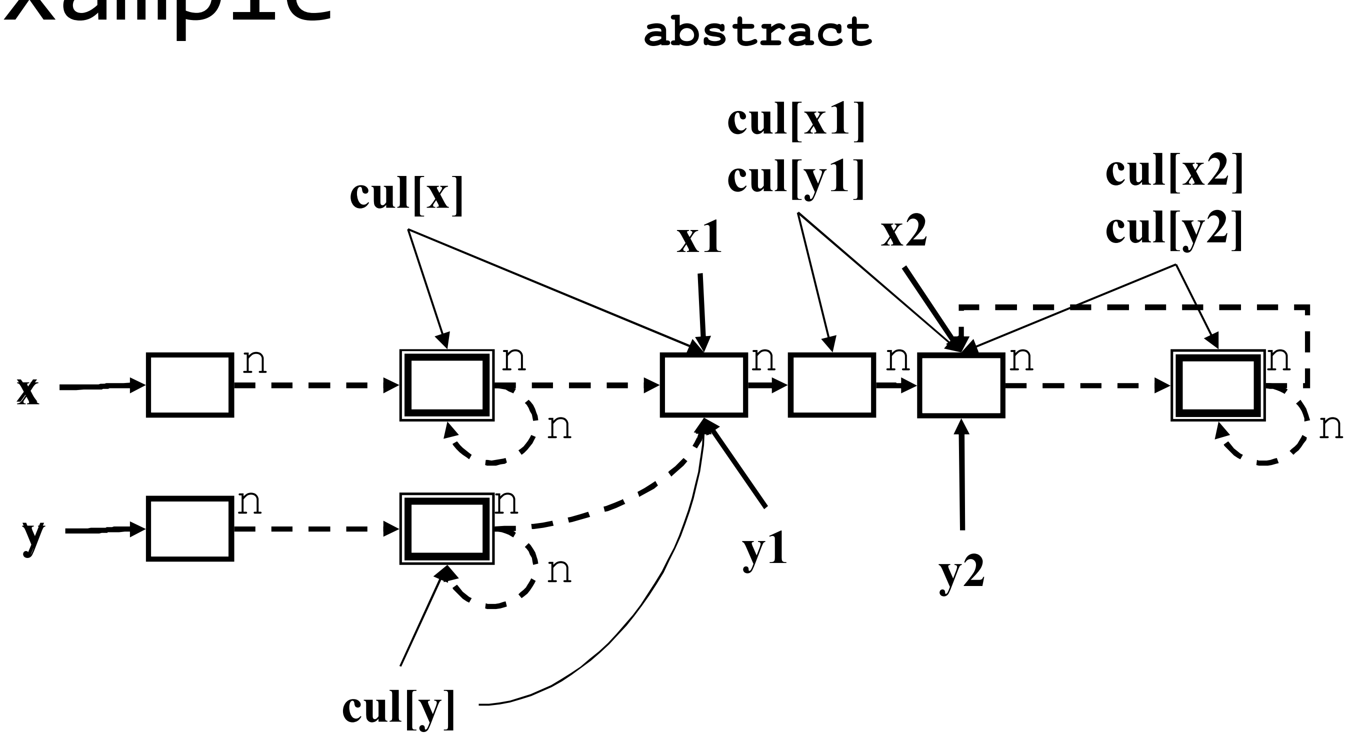
- For every variable x (regular and auxiliary)
  - **x(v)** = v is pointed-to by x
  - **cul[x](v)** = uninterrupted list from node pointed-to by x to v
- Predicates expressed by FO^{TC} formulae

Canonical Abstraction Example

concrete

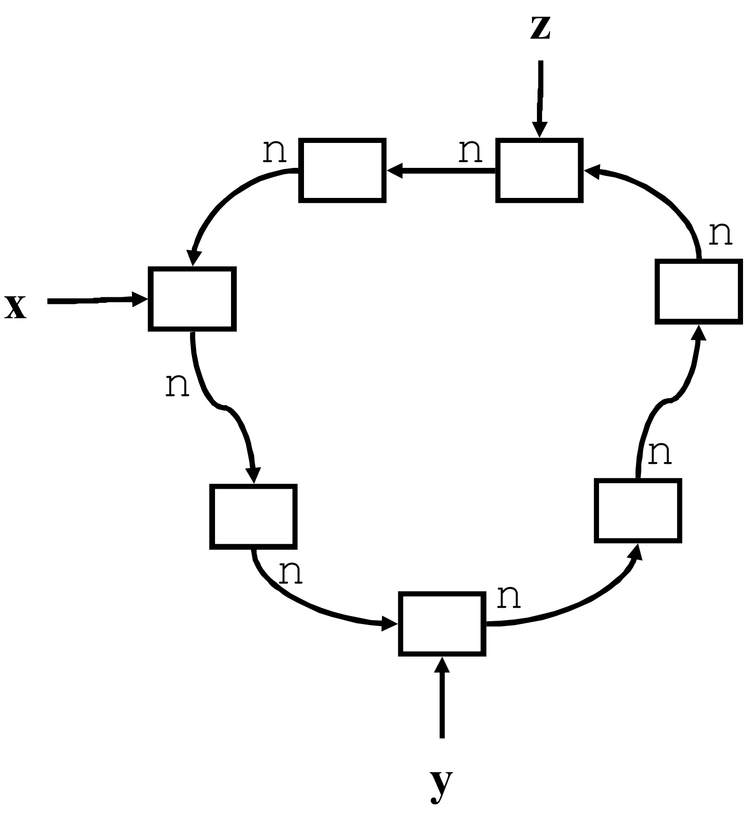


Canonical Abstraction Example

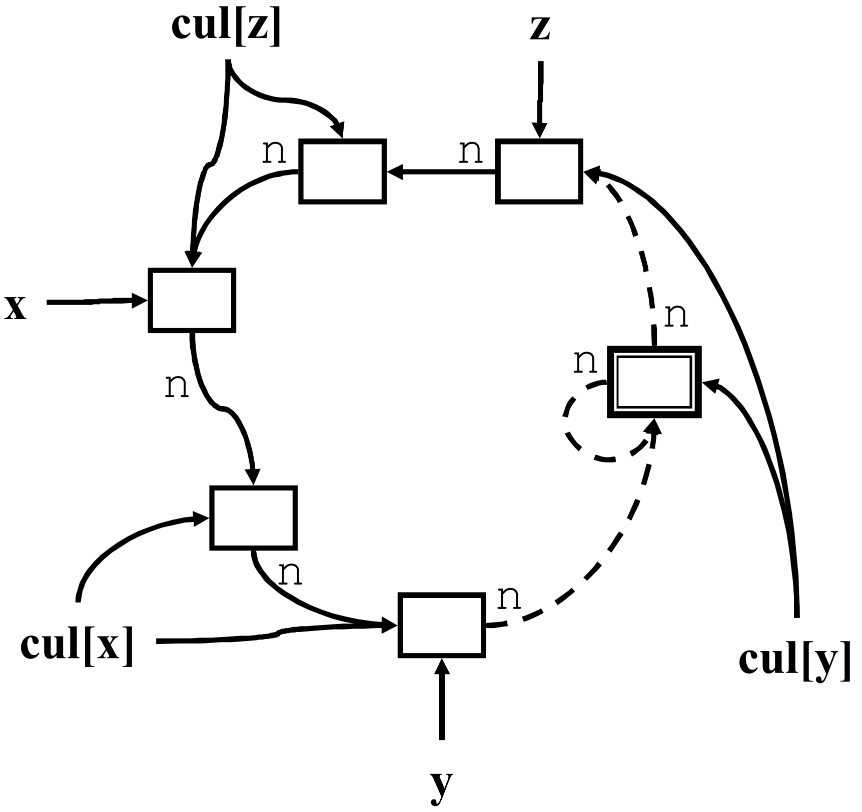


Canonical Abstraction of Cyclic List

concrete

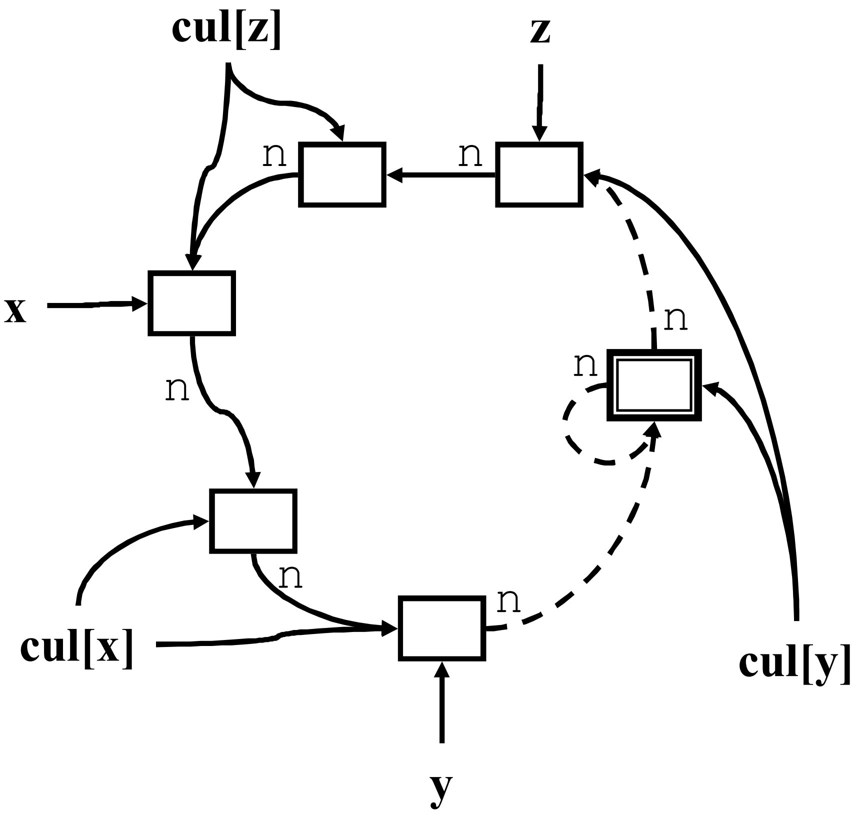


abstract

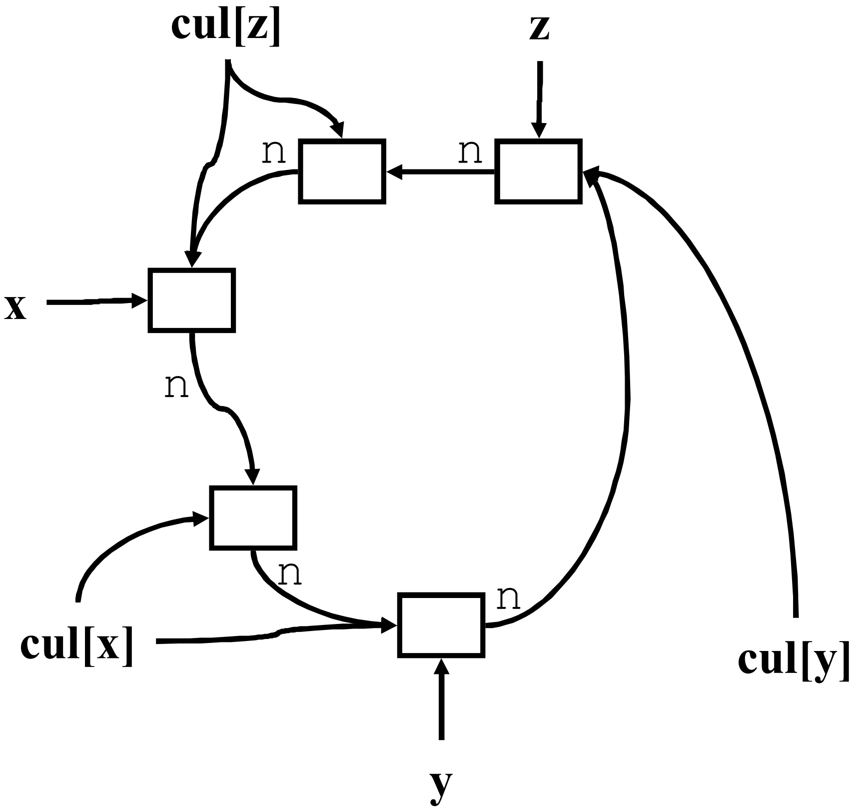


Canonical Abstraction of Cyclic List

abstract pre



abstract post



Related Work

- Dor, Rode and Sagiv [SAS ’00]
  - Checking cleanness in lists
- Sagiv, Reps and Wilhelm [TOPLAS ‘02]
  - General framework + abstractions for lists
- Dams and Namjoshi [VMCAI ’03]
  - Semi-automatic predicate abstraction for shape analysis
- Balaban, Pnueli and Zuck [VMCAI ’05]
  - Predicate abstraction for shapes via small models
- Deutsch [PLDI ’94]
  - Symbolic access paths with lengths

Conclusion

- New abstractions for lists
  - Observations about concrete shapes
  - Precise for programs containing heaps with sharing and cycles, ignoring list lengths
  - Parametric in sharing-depth d:[1…k]
- Encoded new abstractions via
  - Canonical abstraction $O(d \cdot k)$
  - **Polynomial** predicate abstraction $O(d^2 \cdot k^2)$
  - d=1 sufficient for all examples