

Effectively-Propositional Reasoning about Reachability in Linked Data Structures

Shachar Itzhaky	TAU
Anindya Banerjee	IMDEA
Neil Immerman	UMASS
Aleks Nanevski	IMDEA
Mooly Sagiv	TAU

<http://www.cs.tau.ac.il/~shachar/afwp.html>

Motivation

- Proving presence (absence) of pointer paths between memory allocated objects in a given program
 - Partial program correctness
 - Memory safety
 - Absence of memory leaks
 - Data structure invariants
 - Acyclicity, Sortedness
 - Total program correctness
 - Program equivalence

Program Termination

$\{x < n^* > y\}$

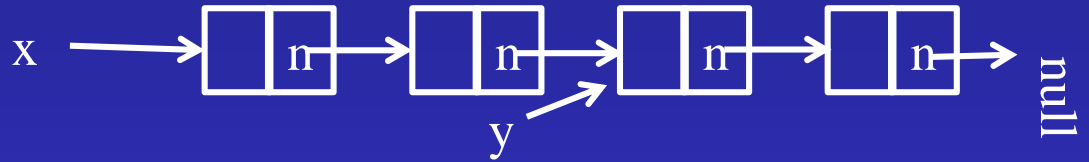
```
traverse(Node x, Node y) {
```

```
  for (t = x; t != y ; t = t.n) {
```

```
    ...
```

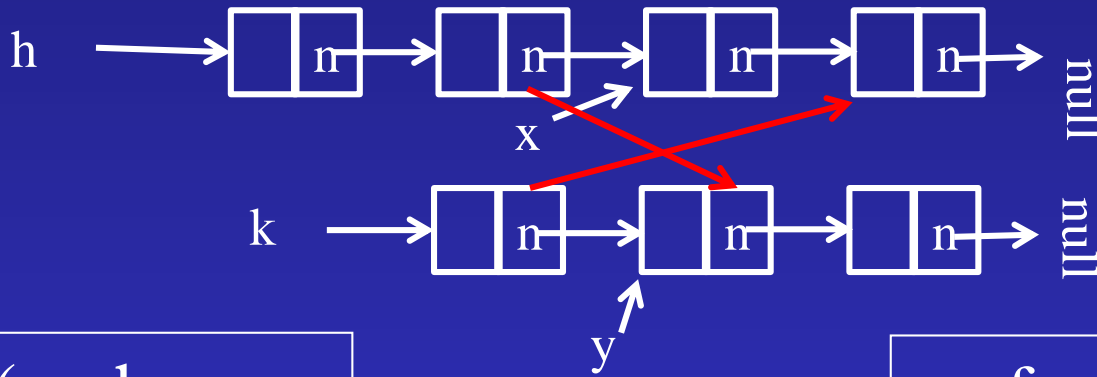
```
  }
```

```
}
```



Disjoint Parallelism

$$\{\forall \alpha: \alpha \neq \text{null} \rightarrow \neg(\text{h} \langle \text{n}^* \rangle \alpha \wedge \text{k} \langle \text{n}^* \rangle \alpha)\}$$



```
for (x = h;  
    x != null;  
    x = x.n) {  
    ...  
}
```

||

```
for (y = k;  
    y != null;  
    y = y.n) {  
    ...  
}
```

Challenges

- Complexity of reasoning about reachability assertions
 - Undecidability of reachability

"there is a mismatch between the simple intuitions about the way pointer operations work and the complexity of their axiomatic treatments"

O'Hearn, Reynolds, Yang [CSL 2001]

- [Inferring reachability properties from the code]

Link list manipulations are simple

- Simple to reason about correctness
 - Small counterexamples
- “Simple” invariants
 - Alternation Free + Reachability “ \subseteq ” $\exists^* \forall^*$

EA($\exists^* \forall^*$) formulas

Bernays-Schönfinkel-Ramsey

- $t ::= \text{var} \mid \text{constant}$ (Terms)
- $\text{ap} ::= t_1 = t_2 \mid r(t_1, t_2, \dots, t_n)$
- $\text{qf} ::= \text{ap} \mid \text{qf}_1 \wedge \text{qf}_2 \mid \text{qf}_1 \vee \text{qf}_2 \mid \neg \text{qf}$
- $\text{ea} ::= \exists \alpha_1, \alpha_2, \alpha_n: \forall \beta_1, \beta_2, \beta_m: \text{qf}$
- Effectively Propositional
 - Skolemization yields finite models
 - EQ-satisfiable to a propositional formula
 - Support from Z3

EA($\exists\forall$) formulas

Bernays-Schönfinkel-Ramsey

$$\exists\alpha_1, \alpha_2, : \forall \beta_1 : r(\alpha_1, \beta_1) \leftrightarrow r(\beta_1, \alpha_2)$$

$$=_{\text{sat}} \forall \beta_1 : r(c_1, \beta_1) \leftrightarrow r(\beta_1, c_2)$$

$$=_{\text{sat}} (r(c_1, c_1) \leftrightarrow r(c_1, c_2)) \wedge \\ (r(c_1, c_2) \leftrightarrow r(c_2, c_2))$$

$$=_{\text{sat}} (P_{11} \leftrightarrow P_{12}) \wedge (P_{12} \leftrightarrow P_{22})$$

Alternation Free Reachability (AF^R)

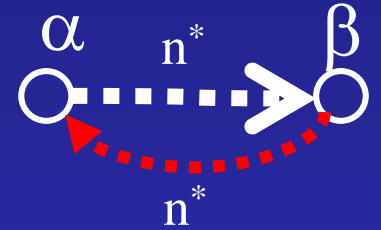
- “Extended subset” of EA
 - Closed under negation
- $t ::= \text{var} \mid \text{constant}$ (Terms)
- $\text{ap} ::= t_1 = t_2 \mid r(t_1, t_2, \dots, t_n)$
| $t_1 \langle f^* \rangle t_2$ (Reachability via sequences of f 's)
(exists $k: f^k(t_1) = t_2$)
- $\text{qf} ::= \text{qf} \mid \text{qf}_1 \wedge \text{qf}_2 \mid \text{qf}_1 \vee \text{qf}_2 \mid \neg \text{qf}$
- $e ::= \exists \alpha_1, \alpha_2, \dots, \alpha_n: \text{qf}$
 $a ::= \forall \beta_1, \beta_2, \dots, \beta_m: \text{qf}$
- $\text{af}^R ::= e \mid a \mid \text{af}^R_1 \wedge \text{af}^R_2 \mid \text{af}^R_1 \vee \text{af}^R_2$

AF^R Program Properties

- Acyclicity

- $\forall \alpha, \beta: \alpha \langle n^+ \rangle \beta \rightarrow \neg \beta \langle n^+ \rangle \alpha$

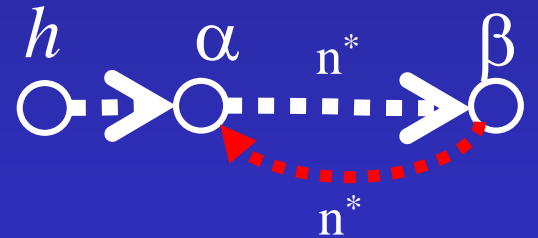
- $\forall \alpha, \beta: \alpha \langle n^* \rangle \beta \wedge \beta \langle n^* \rangle \alpha \rightarrow \alpha = \beta$



- Acyclic list with a head h

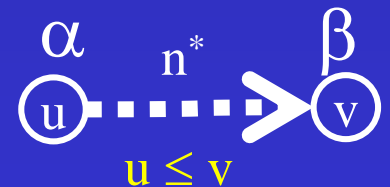
- $\forall \alpha, \beta: h \langle n^* \rangle \alpha \wedge h \langle n^* \rangle \beta \wedge$

- $\alpha \langle n^* \rangle \beta \wedge \beta \langle n^* \rangle \alpha \rightarrow \alpha = \beta$



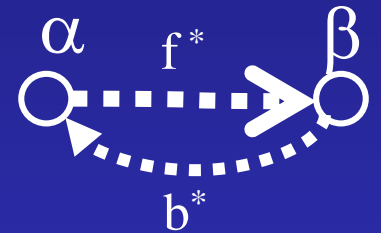
- Sorted segment

- $\forall \alpha, \beta: \alpha \langle n^* \rangle \beta \rightarrow \alpha \leq_{\text{data}} \beta$

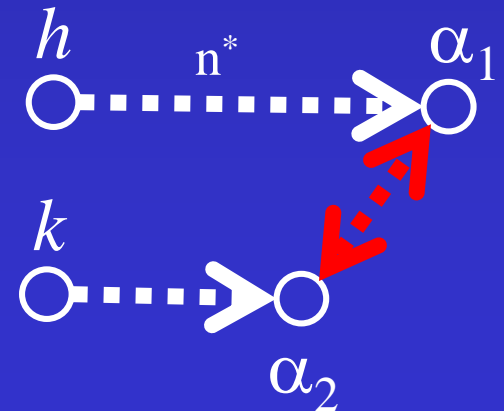


AF^R Program Properties

- Doubly linked lists
 - $\forall \alpha, \beta: \alpha \langle f^* \rangle \beta \leftrightarrow \beta \langle b^* \rangle \alpha$



- Disjoint lists with heads h and k
 - $\forall \alpha: \alpha \neq \text{null} \rightarrow \neg(h \langle n^* \rangle \alpha \wedge k \langle n^* \rangle \alpha)$



List Reversal (isolatd)

$\{ac [h] \wedge \forall \alpha: h < \underline{n^*} > \alpha\}$

```
Node reverse(Node h) {
```

```
  Node c = h;  Node d = null;
```

```
  while (c != null) {
```

```
    Node t = c.next;
```

```
    c.next = d;
```

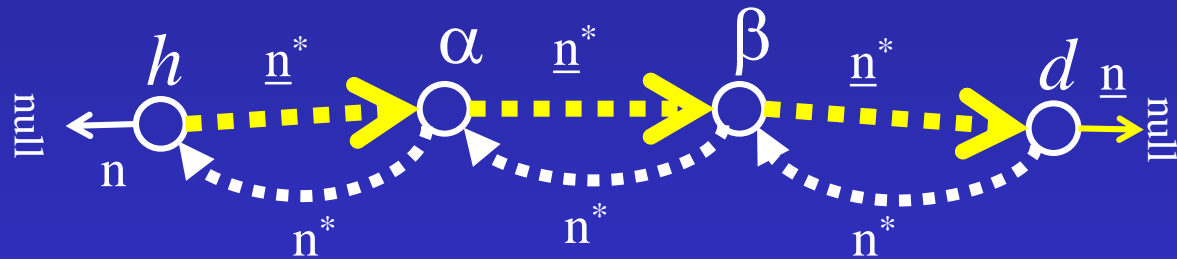
```
    d = c;
```

```
    c = t;
```

```
  }
```

```
  return d
```

```
}
```



$$\alpha < \underline{n^*} > \beta \leftrightarrow \beta < \underline{n^*} > \alpha$$

Invariant List Reversal (isolatd)

$\{ac [h] \wedge \forall \alpha: h <n^*> \alpha\}$

Node reverse(Node h) {

Node c = h; Node d = null;

while (c != null) {

Node t = c.next;

c.next = d;

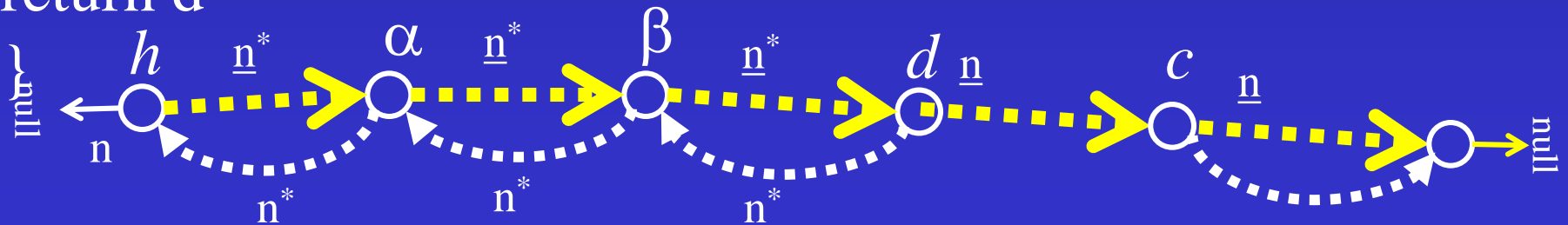
d = c;

c = t;

}

return d

$$I = \forall \alpha, \beta: \left\{ \begin{array}{l} \alpha <n^*> \beta \leftrightarrow \beta <\underline{n}^*> \alpha \quad d <n^*> \alpha \\ c <n^*> \alpha \wedge \quad \neg d <n^*> \alpha \\ (\alpha <n^*> \beta \leftrightarrow \alpha <\underline{n}^*> \beta) \end{array} \right.$$



List Reversal (isolated)

$\{ac [h] \wedge \forall \alpha: h \langle n^* \rangle \alpha\}$

Node reverse(Node h) {

Node c = h; Node d = null;

while **{I}** (c != null) {

Node t = c.next;

c.next = d;

d = c;

c = t;

}

return d

}

$$I = \forall \alpha, \beta: \left\{ \begin{array}{ll} \alpha \langle n^* \rangle \beta \leftrightarrow \beta \langle \underline{n}^* \rangle \alpha & d \langle n^* \rangle \alpha \\ c \langle n^* \rangle \alpha \wedge \neg d \langle n^* \rangle \alpha & \\ (\alpha \langle n^* \rangle \beta \leftrightarrow \alpha \langle \underline{n}^* \rangle \beta) & \end{array} \right.$$

$\{ac[d] \wedge \forall \alpha, \beta: \alpha \langle n^* \rangle \beta \leftrightarrow \beta \langle \underline{n}^* \rangle \alpha\}$

List Reversal (isolated)

$\{ac [h] \wedge \forall \alpha: h \langle n^* \rangle \alpha\}$

Node reverse(Node h) {

Node c = h; Node d = null;

while **{I}** (c != null) {

Node t = c.next;

c.next = d;

d = c;

c = t;

}

return d

}

$I = \forall \alpha, \beta: \left\{ \begin{array}{ll} \alpha \langle n^* \rangle \beta \leftrightarrow \beta \langle \underline{n}^* \rangle \alpha & d \langle n^* \rangle \alpha \\ c \langle n^* \rangle \alpha \wedge & \neg d \langle n^* \rangle \alpha \\ (\alpha \langle n^* \rangle \beta \leftrightarrow \alpha \langle \underline{n}^* \rangle \beta) & \end{array} \right.$

$\{ac[d] \wedge \forall \alpha, \beta: \alpha \langle n^* \rangle \beta \leftrightarrow \beta \langle \underline{n}^* \rangle \alpha \wedge \forall \alpha: d \langle n^* \rangle \alpha\}$

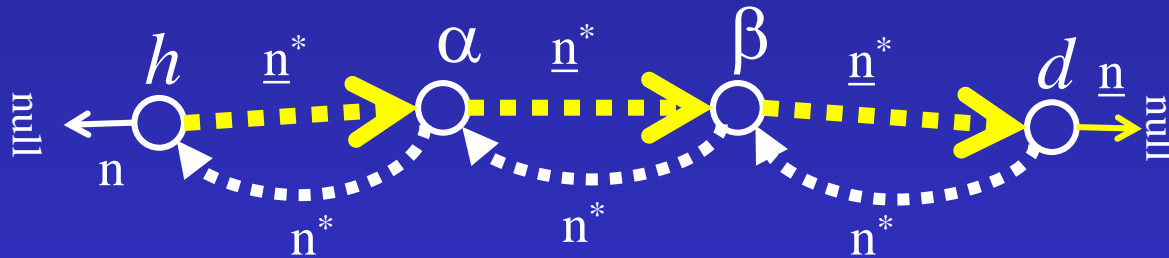
List Reversal (ownership)

$\{ \forall \alpha, \beta : h \langle n^* \rangle \alpha \wedge \beta \langle n^* \rangle \alpha \rightarrow h \langle n^* \rangle \beta \}$

```
Node reverse(Node h) {  
  Node c = h;  Node d = null;  
  while {I} (c != null) {  
    Node t = c.next;  
    c.next = d;  
    d = c;  
    c = t;  
  }  
  return d  
}
```


List Reversal (ownership)

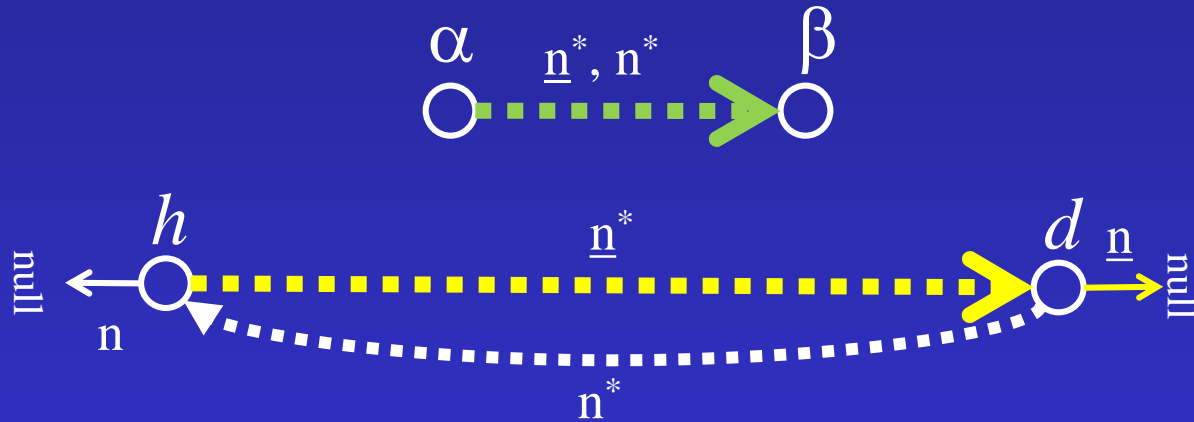
Case 1: $h \langle \underline{n}^* \rangle \alpha \wedge h \langle \underline{n}^* \rangle \beta$



$$\alpha \langle \underline{n}^* \rangle \beta \leftrightarrow \beta \langle \underline{n}^* \rangle \alpha$$

List Reversal (ownership)

Case 2: $\neg h<\underline{n}^*>\alpha \wedge \neg h<\underline{n}^*>\beta$

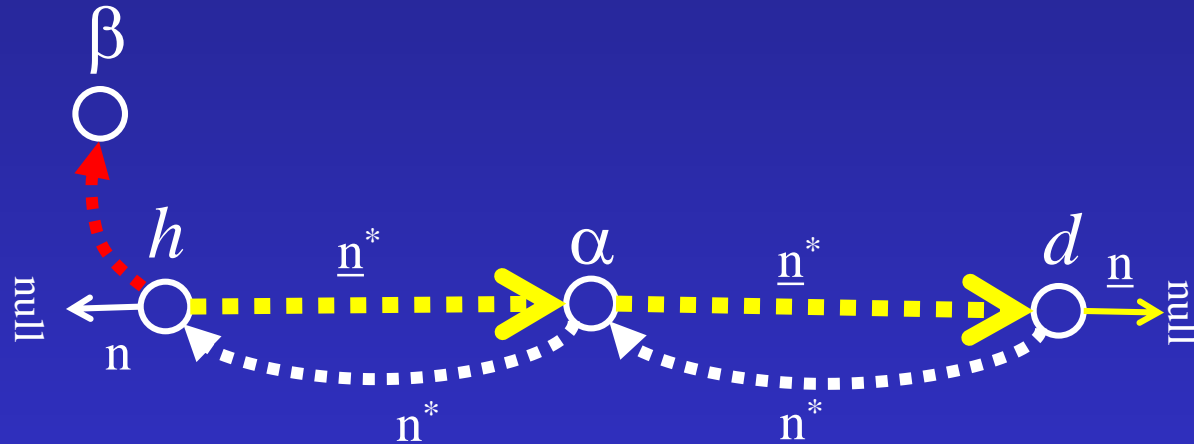


$$\alpha < n^* > \beta \leftrightarrow \alpha < \underline{n}^* > \beta$$

List Reversal (ownership)

Case 3:

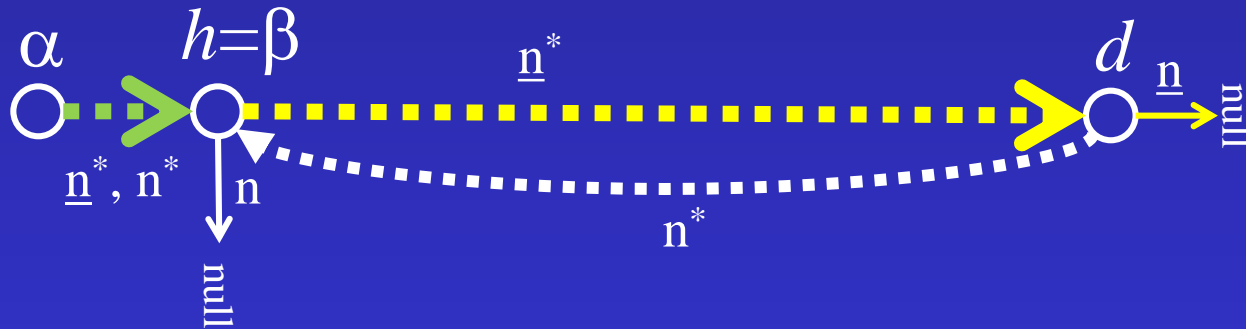
$$h \langle \underline{n}^* \rangle \alpha \wedge \neg h \langle \underline{n}^* \rangle \beta$$



$$\alpha \langle \underline{n}^* \rangle \beta \leftrightarrow \text{false}$$

List Reversal (ownership)

Case 4: $\neg h<\underline{n}^*>\alpha \wedge h<\underline{n}^*>\beta$



$$\alpha<n^*>\beta \leftrightarrow \alpha<\underline{n}^*>h=\beta$$

List Reversal (ownership)

$\{ ac [h] \wedge \forall \alpha, \beta : h \langle n^* \rangle \alpha \wedge \beta \langle n^* \rangle \alpha \rightarrow h \langle n^* \rangle \beta \}$

Node reverse(Node h) {

Node c = h; Node d = null;

while (c != null) {

Node t = c.next;

c.next = d;

d = c;

c = t;

}

return d;

}

$$\forall \alpha, \beta : \alpha \langle n^* \rangle \beta \leftrightarrow \begin{cases} \beta \langle \underline{n}^* \rangle \alpha & h \langle \underline{n}^* \rangle \alpha \wedge h \langle \underline{n}^* \rangle \beta \\ \alpha \langle \underline{n}^* \rangle \beta & \neg h \langle \underline{n}^* \rangle \alpha \wedge \neg h \langle \underline{n}^* \rangle \beta \\ \text{false} & h \langle \underline{n}^* \rangle \alpha \wedge \neg h \langle \underline{n}^* \rangle \beta \\ \alpha \langle \underline{n}^* \rangle h = \beta & \neg h \langle \underline{n}^* \rangle \alpha \wedge h \langle \underline{n}^* \rangle \beta \end{cases}$$

Why AF^R ?

- Represents the invariants of simple linked list manipulations
- Closed under $\vee, \wedge, \neg, \rightarrow$
- Finite model property
- Decidable for satisfiability/validity
- $AF^R \propto AF$
- Can be reduced to a propositional formula
 - SAT solver is complete for verification/falsification

$AF^R \propto AF$

- Introduce an auxiliary relation n^*
- $t[\alpha <n^*>\beta] = n^*(\alpha, \beta)$
- **Completely** axiomatize n^* by an AF formula
$$\Gamma_{\text{linOrd}} = \forall \alpha, \beta: n^*(\alpha, \beta) \wedge n^*(\beta, \alpha) \leftrightarrow \alpha = \beta \wedge$$
$$\forall \alpha, \beta, \gamma: n^*(\alpha, \beta) \wedge n^*(\beta, \gamma) \rightarrow n^*(\alpha, \gamma) \wedge$$
$$\forall \alpha, \beta, \gamma: n^*(\alpha, \beta) \wedge n^*(\alpha, \gamma) \rightarrow (n^*(\beta, \gamma) \vee n^*(\gamma, \beta))$$
- φ is satisfiable \leftrightarrow $(\Gamma_{\text{linOrd}} \wedge t[\varphi])$ is satisfiable
 - AF formulas have finite model

Inverting $n^* \Rightarrow n$

- Every finite model in which n^* satisfies the order requirements:

$$\Gamma_{\text{linOrd}} =$$

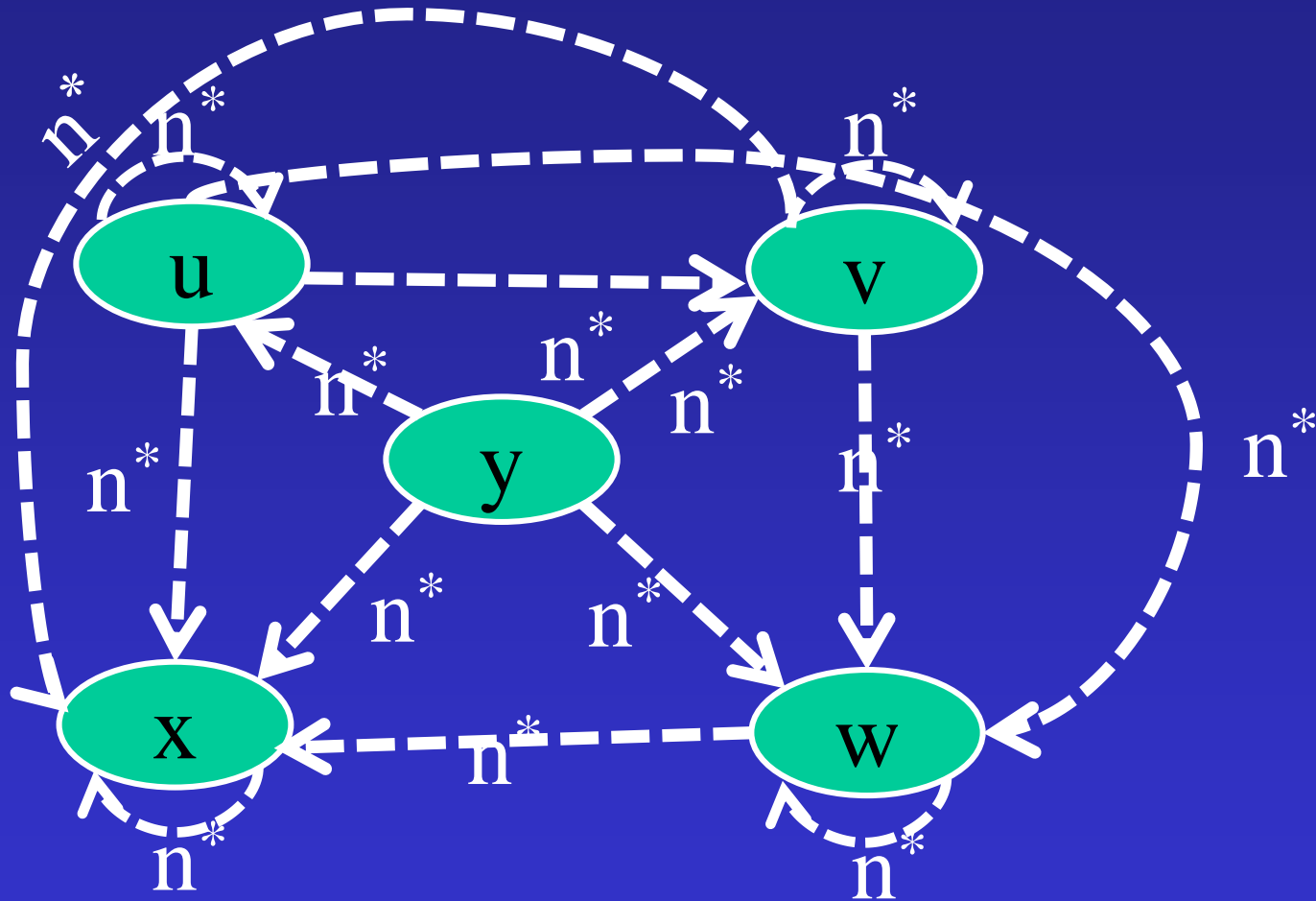
$$\forall \alpha, \beta: n^*(\alpha, \beta) \wedge n^*(\beta, \alpha) \leftrightarrow \alpha = \beta \wedge$$

$$\forall \alpha, \beta, \gamma: n^*(\alpha, \beta) \wedge n^*(\beta, \gamma) \rightarrow n^*(\alpha, \gamma) \wedge$$

$$\forall \alpha, \beta, \gamma: n^*(\alpha, \beta) \wedge n^*(\alpha, \gamma) \rightarrow (n^*(\beta, \gamma) \vee n^*(\gamma, \beta))$$

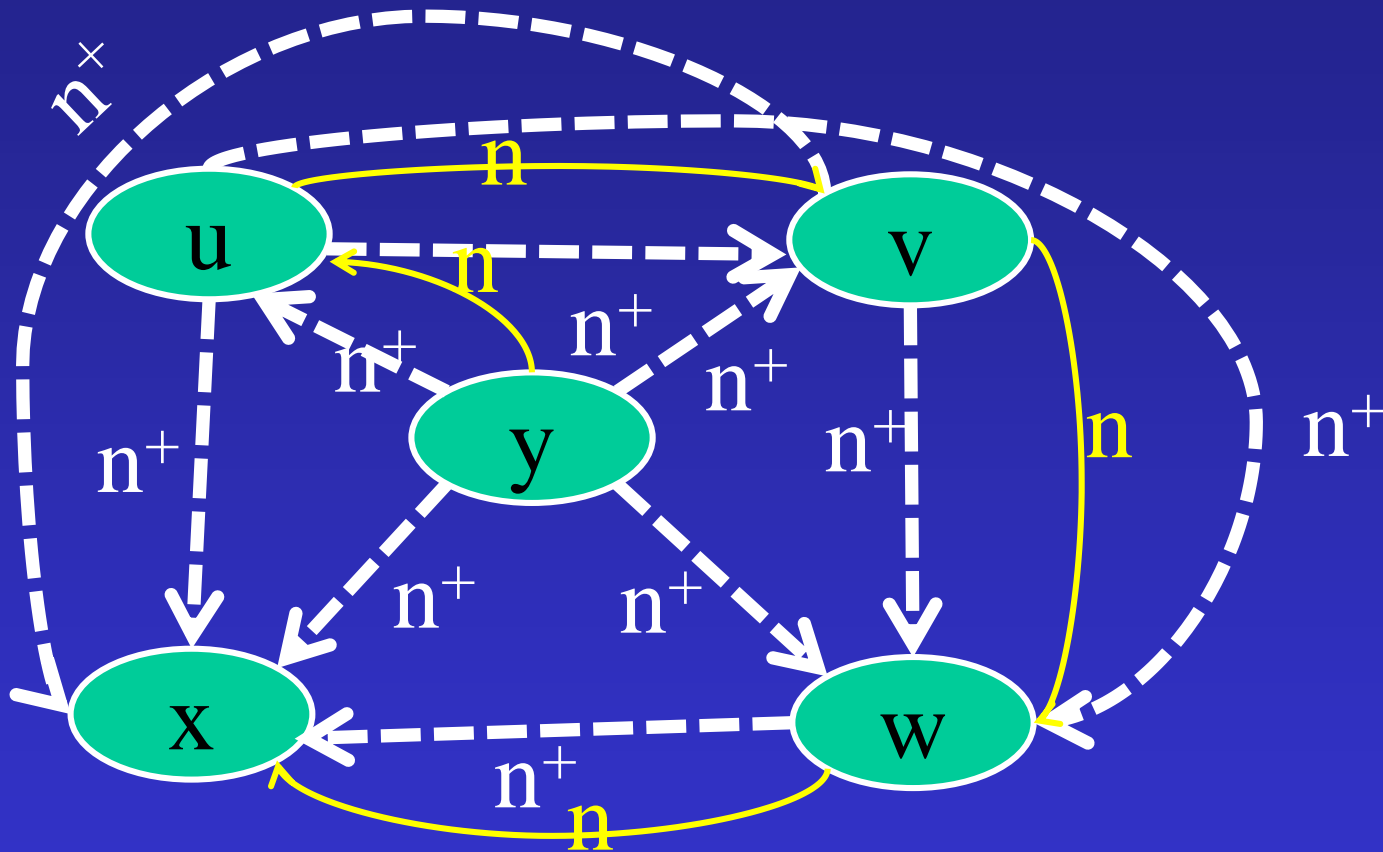
- n^* uniquely determines n

Inverting $n^* \Rightarrow n$



$$\alpha \langle n^+ \rangle \beta \leftrightarrow \alpha \langle n^* \rangle \beta \wedge \alpha \neq \beta$$

Inverting $n^* \Rightarrow n$



$$n(\alpha) = \beta \Leftrightarrow \alpha \langle n^+ \rangle \beta \wedge \forall \gamma: \alpha \langle n^+ \rangle \gamma \rightarrow \beta \langle n^* \rangle \gamma$$

Simple SAT Application

- Determine if two clients are identical
 - Produce isomorphic reachable stores
- $\text{reverse}(\text{reverse}(h)) = h$

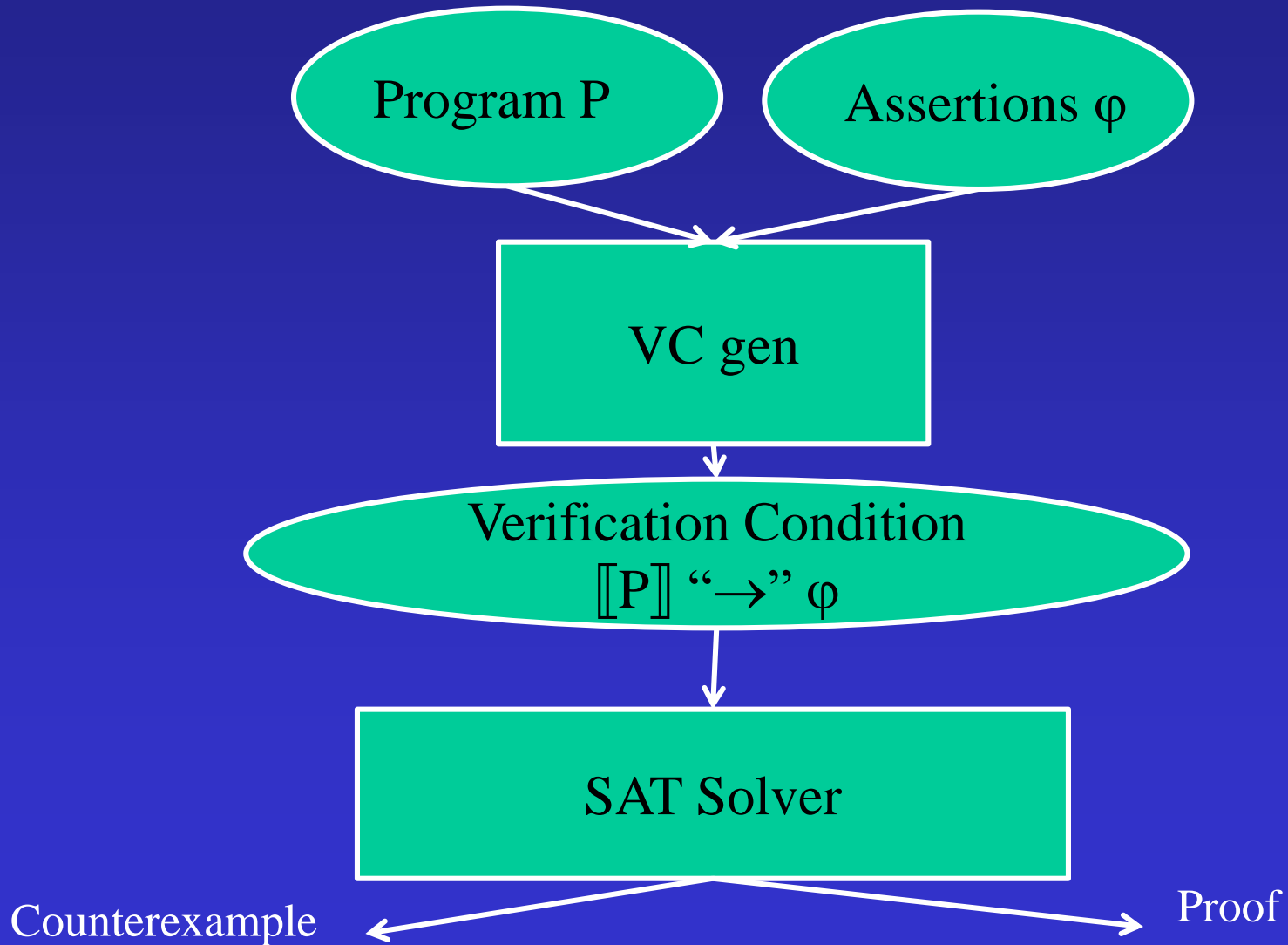
$$\forall \alpha, \beta: \alpha \langle n_1^* \rangle \beta \leftrightarrow \beta \langle n_0^* \rangle \alpha \wedge$$

$$\forall \alpha, \beta: \alpha \langle n_2^* \rangle \beta \leftrightarrow \beta \langle n_1^* \rangle \alpha$$



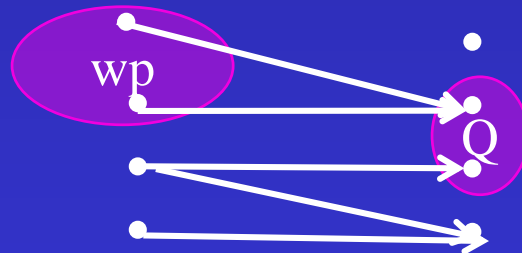
$$\forall \alpha, \beta: \alpha \langle n_0^* \rangle \beta \leftrightarrow \alpha \langle n_2^* \rangle \beta$$

Verification Process



Weakest Precondition

- $wp: \text{Stm} \rightarrow (\text{Ass} \rightarrow \text{Ass})$
- $wp \llbracket S \rrbracket (Q)$ – the weakest condition such that every terminating computation of S results in a state satisfying Q
- $\sigma \models wp \llbracket S \rrbracket (Q) \leftrightarrow \forall \sigma': \sigma \llbracket S \rrbracket \sigma' \rightarrow \sigma' \models Q$



- Can be used to compute verification conditions

Hoare Assignment Rule

- $\text{wp}[[x := e]](Q) = Q[e / x]$
- $\text{wp}[[x := 5]](x=5) = 5=5 \equiv \text{true}$
- $\text{wp}[[x := 5]](x=6) = 6=5 \equiv \text{false}$
- $\text{wp}[x := x + 1](x=7) = x+1 = 7 \equiv x = 6$

$$\text{wc} [[c := d]] \left[\begin{array}{l} \forall \alpha, \beta: \left\{ \begin{array}{l} \alpha \langle \underline{n}^* \rangle \beta \leftrightarrow \beta \langle \underline{n}^* \rangle \alpha \\ c \langle \underline{n}^* \rangle \alpha \wedge \\ \alpha \langle \underline{n}^* \rangle \beta \leftrightarrow \alpha \langle \underline{n}^* \rangle \beta \end{array} \right. \quad \begin{array}{l} d \langle \underline{n}^* \rangle \alpha \\ \neg d \langle \underline{n}^* \rangle \alpha \end{array} \end{array} \right] =$$

$$\forall \alpha, \beta: \left\{ \begin{array}{l} \alpha \langle \underline{n}^* \rangle \beta \leftrightarrow \beta \langle \underline{n}^* \rangle \alpha \\ d \langle \underline{n}^* \rangle \alpha \wedge \\ \alpha \langle \underline{n}^* \rangle \beta \leftrightarrow \alpha \langle \underline{n}^* \rangle \beta \end{array} \quad \begin{array}{l} d \langle \underline{n}^* \rangle \alpha \\ \neg d \langle \underline{n}^* \rangle \alpha \end{array} \right.$$

WP Compound statements

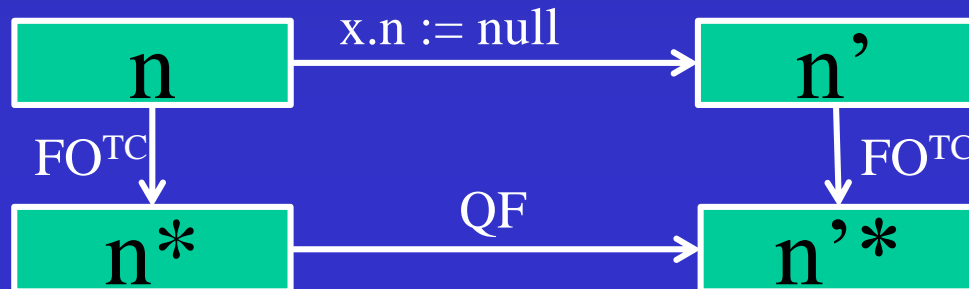
- $\text{wp} \llbracket \text{skip} \rrbracket (Q) = Q$
- $\text{wp} \llbracket x := e \rrbracket (Q) = Q[e / x]$
- $\text{wp} \llbracket S_1; S_2 \rrbracket (Q) = \text{wp} \llbracket S_1 \rrbracket (\text{wp} \llbracket S_2 \rrbracket (Q))$
- $\text{wp} \llbracket \text{if } B \text{ then } S_1 \text{ else } S_2 \rrbracket =$
 $(\llbracket B \rrbracket \wedge \text{wp} \llbracket S_1 \rrbracket (Q)) \vee (\neg \llbracket B \rrbracket \wedge \text{wp} \llbracket S_2 \rrbracket (Q))$
- $\text{wp} \llbracket \text{while } B \text{ do } \{I\} S \rrbracket = I$

VC rules

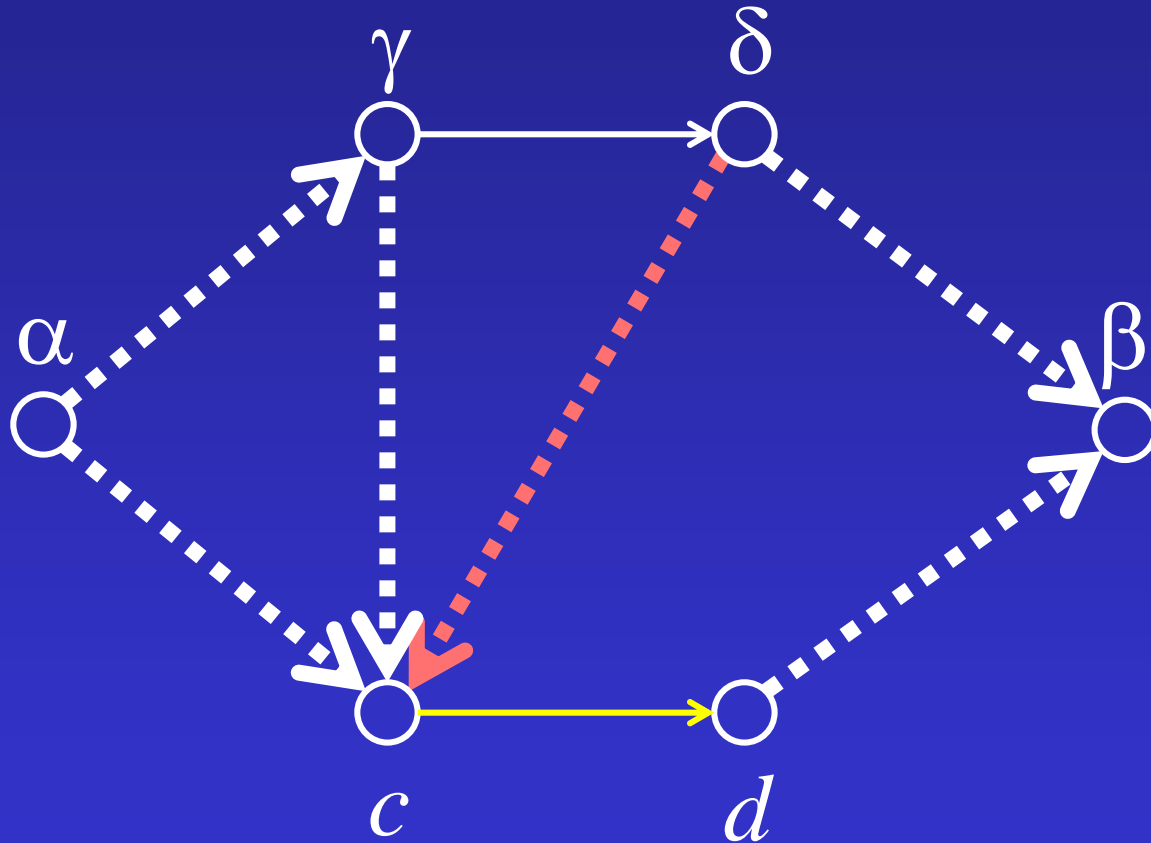
- $VC_{\text{gen}}(\{P\} S \{Q\}) = P \rightarrow \text{wp}[[S]](Q) \wedge \bigwedge VC_{\text{aux}}(S, Q)$
- $VC_{\text{aux}}(S, Q) = \{\}$ (for any atomic statement)
- $VC_{\text{aux}}(S_1; S_2, Q) =$
 $VC_{\text{aux}}(S_1, \text{wp}(S_2, Q)) \cup VC_{\text{aux}}(S_2, Q)$
- $VC_{\text{aux}}(\text{if } C \text{ then } S_1 \text{ else } S_2, Q) =$
 $VC_{\text{aux}}(S_1, Q) \cup VC_{\text{aux}}(S_2, Q)$
- $VC_{\text{aux}}(\text{while } B \text{ do } S, Q) = VC_{\text{aux}}(S, I) \cup$
 $\{I \wedge [[B]] \rightarrow \text{wp}[[S]](I)\} \cup$
 $\{I \wedge \neg [[B]] \rightarrow Q\}$

But how about heap mutations?

- McCarthy assignment rule does not work
- $\text{wp}[\![c.n := \text{null}]\!](Q) = Q[n[c \mapsto \text{null}] / n]$
 - Refers to n
 - Does not explicitly update reachability
 - Outside AF^R
- Employ incremental updates

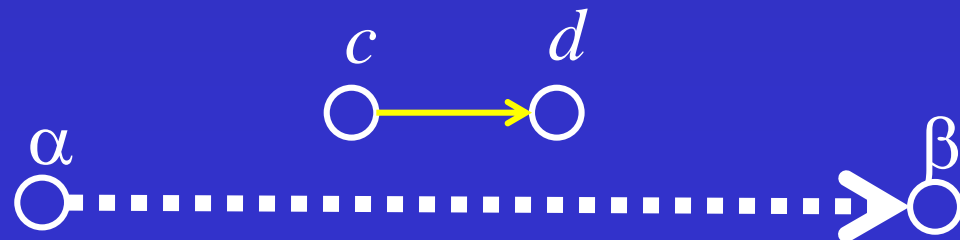


Dong & Su [SIGMOD'00] DAG



$$\boxed{\exists \gamma: \alpha \langle n^* \rangle \gamma \wedge \gamma \langle n^* \rangle c \wedge}$$
$$\boxed{n(\gamma) = \delta} \wedge \delta \langle n^* \rangle \beta \wedge \neg \delta \langle n^* \rangle c$$

Deterministic Graphs (function)



Mutating Single Linked Lists

- $\text{wp}[\![c.n := \text{null}]\!](Q) =$
 $Q[(\alpha \langle n^* \rangle \beta \wedge (\neg \alpha \langle n^* \rangle c \vee \beta \langle n^* \rangle c)) / \alpha \langle n^* \rangle \beta]$
- Can also enforce absence of null dereferences
 $c \neq \text{null}$

Circular Linked Lists

- Slightly more complex but Quantifier-Free
[Hesse'03,Reps, Lahiri&Quadeer POPL'08]
- wp remains in QF

Single Mutation $c.n := y$ (assuming $c.n = \text{null}$)

- Simple for general graphs
- AF^R for arbitrary data structures
- $\text{wp } \llbracket c.n := y \rrbracket (Q) =$
 $Q[(\alpha \langle n^* \rangle \beta \vee (\alpha \langle n^* \rangle c \wedge y \langle n^* \rangle \beta)) / \alpha \langle n^* \rangle \beta]$
- Can also enforce acyclicity $\neg y \langle n^* \rangle c$

But what about pointer traversals?

$x := x.n$

- Hoare assignment rule goes outside AF^R
- $wp[[x := y.n]](Q) = Q[n(y) / x]$
 - Outside AF^R
- Reason about list segments
- Coincides with complications in pointer and shape analysis

WP Compound statements

- $\text{wp} \llbracket \text{skip} \rrbracket (Q) = Q$
- $\text{wp} \llbracket x := e \rrbracket (Q) = Q[e / x]$
- $\text{wp} \llbracket S_1; S_2 \rrbracket (Q) = \text{wp} \llbracket S_1 \rrbracket (\text{wp} \llbracket S_2 \rrbracket (Q))$
- $\text{wp} \llbracket \text{if } B \text{ then } S_1 \text{ else } S_2 \rrbracket =$
 $(\llbracket B \rrbracket \wedge \text{wp} \llbracket S_1 \rrbracket (Q)) \vee (\neg \llbracket B \rrbracket \wedge \text{wp} \llbracket S_2 \rrbracket (Q))$
- $\text{wp} \llbracket \text{while } B \text{ do } \{I\} S \rrbracket = I$

VC rules

- $VC_{\text{gen}}(\{P\} S \{Q\}) = P \rightarrow \text{wp}[[S]](Q) \wedge \bigwedge VC_{\text{aux}}(S, Q)$
- $VC_{\text{aux}}(S, Q) = \{\}$ (for any atomic statement)
- $VC_{\text{aux}}(S_1; S_2, Q) =$
 $VC_{\text{aux}}(S_1, \text{wp}(S_2, Q)) \cup VC_{\text{aux}}(S_2, Q)$
- $VC_{\text{aux}}(\text{if } C \text{ then } S_1 \text{ else } S_2, Q) =$
 $VC_{\text{aux}}(S_1, Q) \cup VC_{\text{aux}}(S_2, Q)$
- $VC_{\text{aux}}(\text{while } B \text{ do } S, Q) = VC_{\text{aux}}(S, I) \cup$
 $\{I \wedge [[B]] \rightarrow \text{wp}[[S]](I)\} \cup$
 $\{I \wedge \neg [[B]] \rightarrow Q\}$

Pointer Traversals

- Observe that wp is only used positively in VCs (unlike invariants and preconditions)
- Allows EA formulas with reachability (AE^R)
- $wp \llbracket x := y.n \rrbracket (Q) = \forall \alpha: 'n(y)=\alpha' \rightarrow Q[\alpha/x]$
 - Replace n with n^* using reachability inversions
- Universal quantifications are also used for allocation $x := \text{new}()$

Backward Reasoning with WP

$\{a\langle n^* \rangle e \wedge c\langle n^* \rangle b \wedge \text{disjoint}(a,c)\}$

$d := e.n ;$

$\left\{ \begin{array}{l} \rightarrow (a\langle n^* \rangle b \wedge (\neg a\langle n^* \rangle d \vee b\langle n^* \rangle d)) \\ \rightarrow (a\langle n^* \rangle d \wedge (\neg a\langle n^* \rangle d \vee d\langle n^* \rangle d)) \wedge \\ \rightarrow c\langle n^* \rangle b \wedge (\neg c\langle n^* \rangle d \vee b\langle n^* \rangle d) \end{array} \right\} \equiv \text{true}$

$d.n := \text{null} ;$

$\{a\langle n^* \rangle b \vee (a\langle n^* \rangle d \wedge c\langle n^* \rangle b)\}$

$d.n := c ;$

$\{a\langle n^* \rangle b\}$

Backward Reasoning with WP

$$\left\{ \begin{array}{l} \{a\langle n^* \rangle e \wedge c\langle n^* \rangle b \wedge \text{disjoint}(a,c)\} \\ \forall \alpha: \text{“}n(e) = \alpha\text{”} \rightarrow \\ \quad (a\langle n^* \rangle b \wedge (\neg a\langle n^* \rangle \alpha \vee b\langle n^* \rangle \alpha)) \\ \quad \vee (a\langle n^* \rangle \alpha \wedge c\langle n^* \rangle b \wedge (\neg c\langle n^* \rangle \alpha \vee b\langle n^* \rangle \alpha)) \end{array} \right\}$$

$d := e.n$;

$$\left\{ \begin{array}{l} (a\langle n^* \rangle b \wedge (\neg a\langle n^* \rangle d \vee b\langle n^* \rangle d)) \\ \vee (a\langle n^* \rangle d \wedge c\langle n^* \rangle b \wedge (\neg c\langle n^* \rangle d \vee b\langle n^* \rangle d)) \end{array} \right\}$$

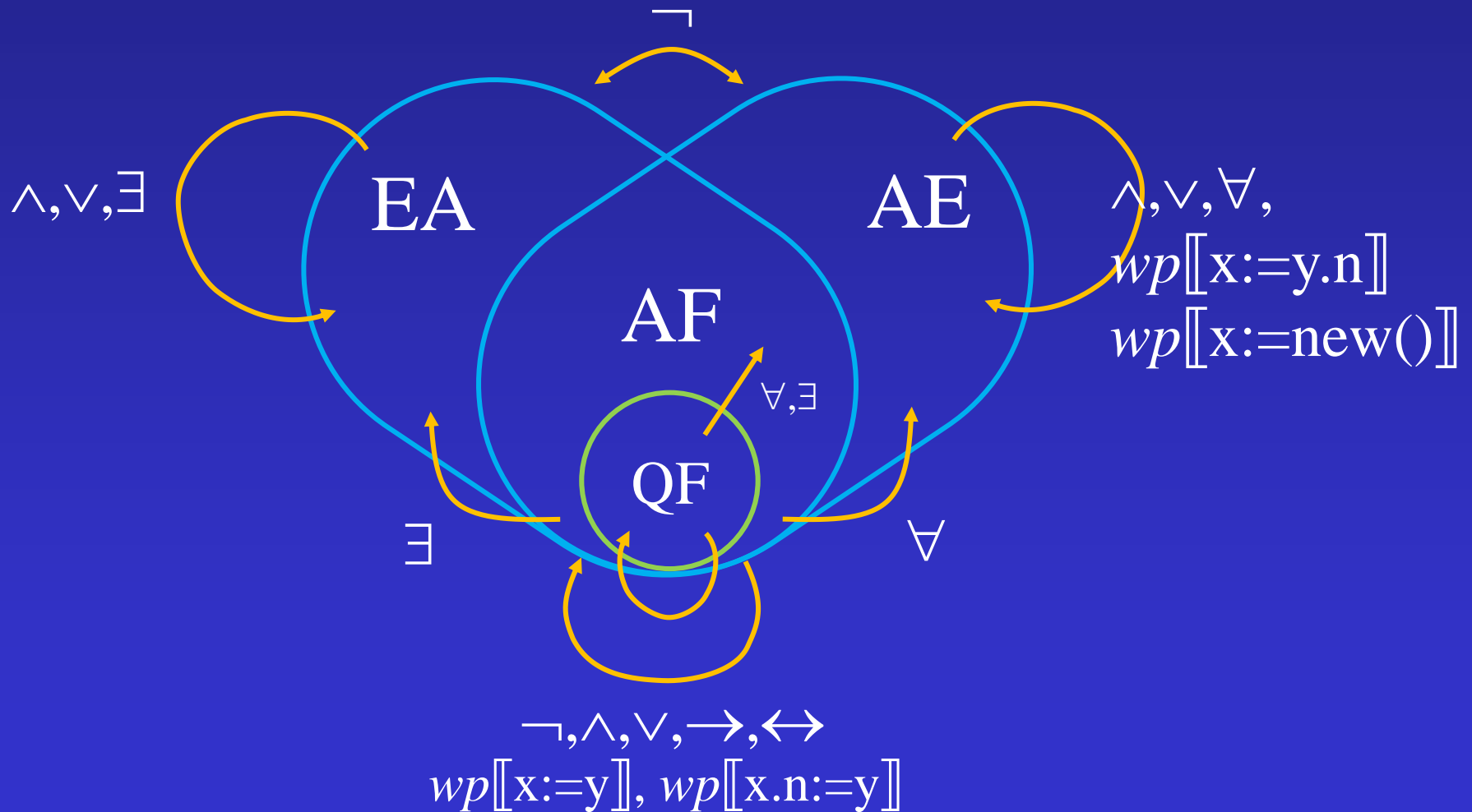
$d.n := \text{null}$;

$$\{a\langle n^* \rangle b \vee (a\langle n^* \rangle d \wedge c\langle n^* \rangle b)\}$$

$d.n := c$;

$$\{a\langle n^* \rangle b\}$$

Closure Properties



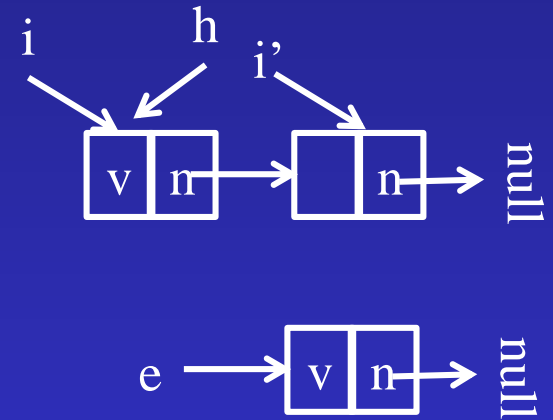
<i>Benchmark</i>	<i>Formula Size</i>						<i>Solving time (Z3)</i>
	P,Q		I		VC		
	#	\forall	#	\forall	#	\forall	
SLL: reverse	2	2	11	2	133	3	57ms
SLL: filter	5	1	14	1	280	4	39ms
SLL: create	1	0	1	0	36	3	13ms
SLL: delete	5	0	12	1	152	3	23ms
SLL: deleteAll	3	2	7	2	106	3	32ms
SLL: insert	8	1	6	1	178	3	17ms
SLL: find	7	1	7	1	64	3	15ms
SLL: last	3	0	5	0	74	3	15ms
SLL: merge	14	2	31	2	2255	3	226ms
SLL: rotate	6	1	-	-	73	3	22ms
SLL: swap	14	2	-	-	965	5	26ms
DLL: fix	5	2	11	2	121	3	32ms
DLL: splice	10	2	-	-	167	4	27ms

Disproving with SAT

Benchmark	Nature of defect	Formula Size						Solving time (Z3)	C.e. Size (vertices)
		P,Q		I		VC			
		#	\forall	#	\forall	#	\forall		
SLL: find	null pointer dereference	7	1	7	1	64	3	18ms	2
SLL: deleteAll	Loop invariant in annotation is too weak to prove the desired property	3	2	5	2	68	3	58ms	5
SLL: rotate	Transient cycle introduced during execution	6	1	-	-	109	3	25ms	3
SLL: insert	Unhandled corner case when an element with the same value already exists in the list --- ordering violated	8	1	6	1	178	3	33ms	4

Example Bug

```
Node insert(Node h, Node e) {  
  Node i = h, j = null;  
  while {I} (i != null && e.val >= i.val) {  
    j = i; i = i.n;  
  }  
  if (j != null) { j.n = e; e.n = i; }  
  else { e.n = h; h = e; }  
  return h;  
}
```



$$I = \forall \alpha: h \langle n^* \rangle \alpha \wedge \neg i \langle n^* \rangle \alpha \rightarrow e \langle_{\text{val}} \alpha$$

Data Structures outside AF^R

- Lists with the same lengths
- DAGs
- Grids
- ...

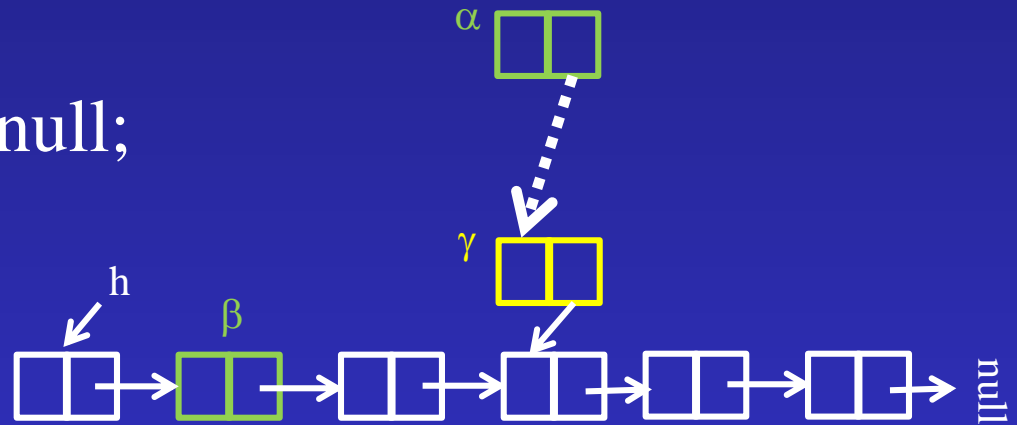
List Reversal (general)

$\{ ac [h] \wedge \forall \alpha, \beta : h \langle n^* \rangle \alpha \wedge \beta \langle n^* \rangle \alpha \rightarrow h \langle n^* \rangle \beta \}$

```

Node reverse(Node h) {
  Node c = h;  Node d = null;
  while {I} (c != null) {
    Node t = c.next;
    c.next = d;
    d = c;
    c = t;
  }
  return d
}

```



$\forall \alpha, \beta : \alpha \langle n^* \rangle \beta \leftrightarrow$
 $\begin{cases} \beta \langle n^* \rangle \alpha \\ \alpha \langle n^* \rangle \beta \\ \text{false} \\ \exists \gamma \alpha \langle n^* \rangle \gamma \wedge \neg h \langle n^* \rangle \gamma \wedge \beta \langle n^* \rangle n(\gamma) \end{cases}$

$\begin{matrix} h \langle n^* \rangle \alpha \wedge h \langle n^* \rangle \beta \\ \neg h \langle n^* \rangle \alpha \wedge \neg h \langle n^* \rangle \beta \\ h \langle n^* \rangle \alpha \wedge \neg h \langle n^* \rangle \beta \\ \neg h \langle n^* \rangle \alpha \wedge h \langle n^* \rangle \beta \end{matrix}$

Related Work

- **Axiomatizing Reachability**
 - [Nelson POPL'83] Useful axioms
 - [Lev-Ami'09] Useful axioms + completeness study
- **Descriptive Complexity** [Hesse'03, Reps'03, Lahiri&Qadeer POPL'08]
- **Decidable Logics** [Mona, STRAND, LRP]

Summary

- Reduction to SAT
- Works for many programs
- Principles
 - Restricted invariants
 - Inversion n^*
 - Incremental updates
 - Two logics