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

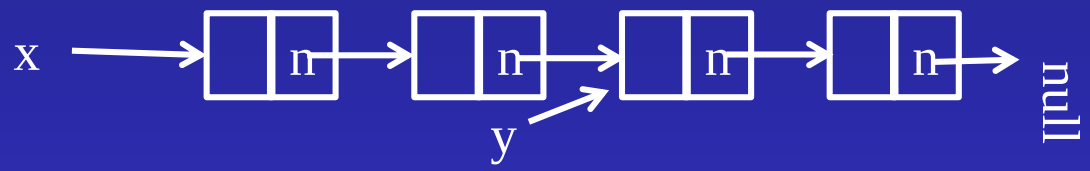
Motivation

- 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
      - Acylicity, Sortedness
  - Total program correctness
  - Program equivalence

Program Termination

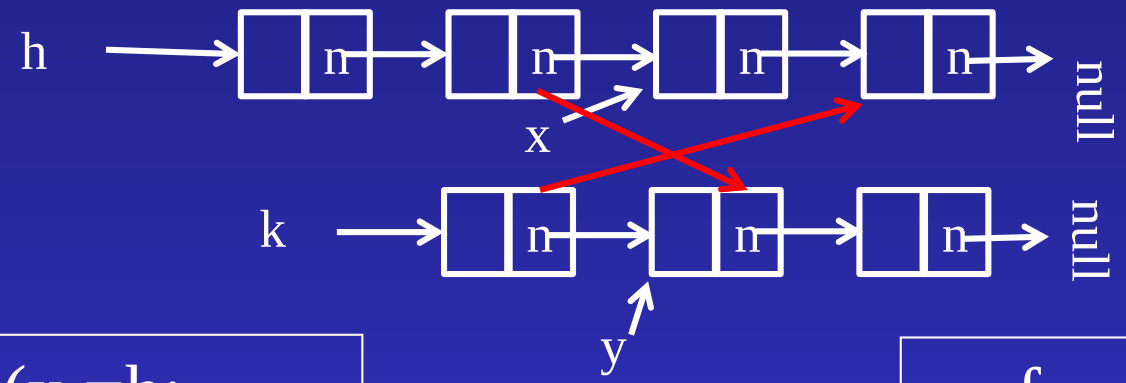
{x <n*> y}

```
traverse(Node x, Node y) {
    for (t =x; t != y ; t = t.n) {
        …
        }
}
```



Disjoint Parallelism

$\{\forall\alpha:\ \alpha \neq null \rightarrow \neg(h<n^*>\alpha \wedge k<n^*>\alpha)\}$




```
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 ::= var | constant (Terms)
- ap ::= $t_1 = t_2$ | $r(t_1, t_2, \ldots, t_n)$
- qf ::= ap | $qf_1 \wedge qf_2$ | $qf_1 \vee qf_2$ | $\neg qf$
- ea ::= $\exists \alpha_1, \alpha_2, \alpha_n: \forall \beta_1, \beta_2, \beta_m: qf$
- Effectively Propositional
  - Skolimization yields finite models
  - EQ-satisfiable to a propositional formula
  - Support from Z3

EA($\exists\forall$) formulas
Bernays-Schönfinkel-Ramsey

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

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

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

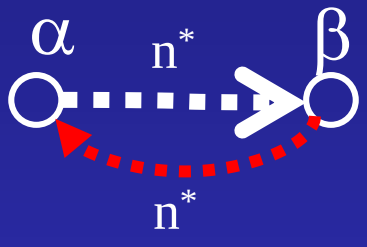
$$=_{sat} (P_{11} \leftrightarrow P_{12}) \wedge (P_{12} \leftrightarrow P_{22})$$

Alternation Free Reachability (AF^R)

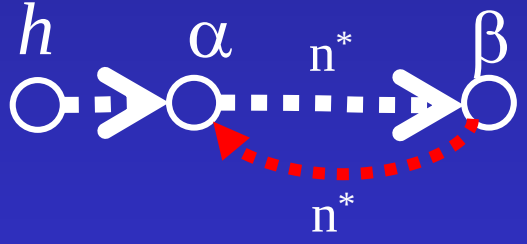
- "Extended subset" of EA
  - Closed under negation
- t ::= var | constant (Terms)
- ap ::= $t_1 = t_2$ | $r(t_1,t_2, \ldots, t_n)$
  | $t_1 <f^*> t_2$ (Reachability via sequences of f's)
  (exists k: $f^k(t_1)=t_2$)
- qf ::= qf | $qf_1 \wedge qf_2$ | $qf_1 \vee qf_2$ | $\neg qf$
- e ::= $\exists \alpha_1, \alpha_2, \ldots, \alpha_n$: qf
  a: ::= $\forall \beta_1, \beta_2, \ldots, \beta_m$: qf
- af^R ::= e | a | $af^R_1 \wedge af^R_2$ | $af^R_1 \vee af^R_2$

AF^R Program Properties

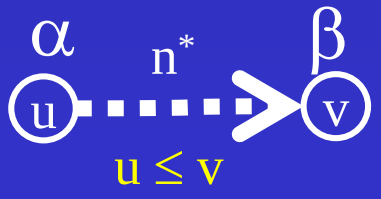
- Acylicity
  - $\forall \alpha, \beta: \alpha <n^+> \beta \rightarrow \neg \beta <n^+> \alpha$
  - $\forall \alpha, \beta: \alpha <n^*> \beta \wedge \beta <n^*> \alpha \rightarrow \alpha = \beta$



- Acyclic list with a head h
  - $\forall \alpha, \beta: h <n^*> \alpha \wedge h <n^*> \beta \wedge \alpha <n^*> \beta \wedge \beta <n^*> \alpha \rightarrow \alpha = \beta$

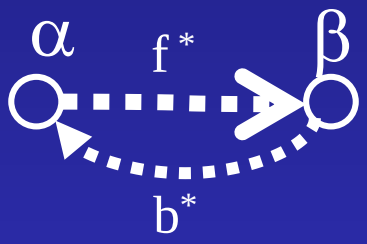


- Sorted segment
  - $\forall \alpha, \beta: \alpha <n^*> \beta \rightarrow \alpha \leq_{data} \beta$

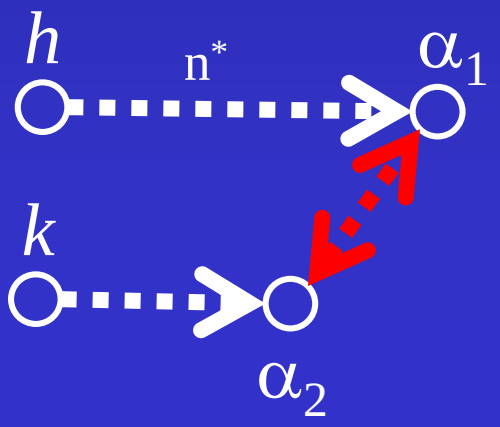


AF^R Program Properties

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



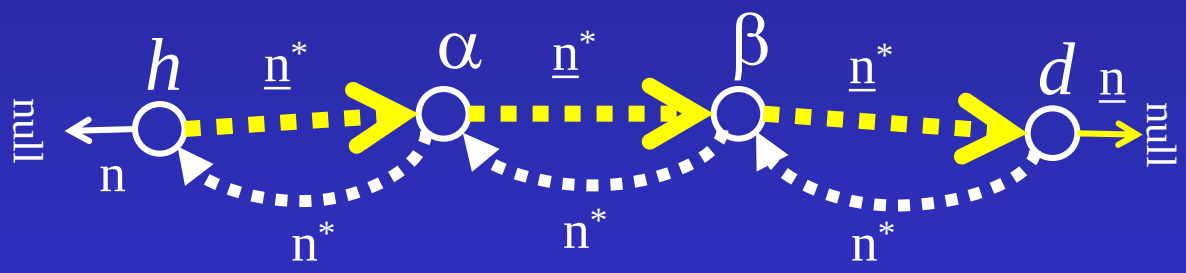
- Disjoint lists with heads h and k
  - $\forall \alpha: \ \alpha \neq \text{null} \rightarrow \neg(h<n^*>\alpha \wedge k<n^*> \alpha)$



List Reversal (isolatd)

{ac [h]∧∀α: h <n*>α}

```
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
}
```



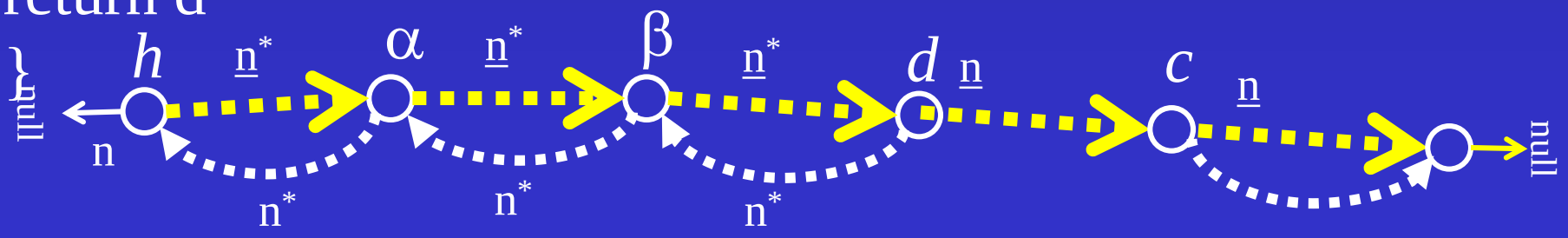
$\alpha<n^*>\beta \leftrightarrow \beta<\underline{n}^*>\alpha$

Invariant List Reversal (isolatd)

{ac [h]∧∀α: h <n*>α}

```
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
}
```

$$I = \forall \alpha, \beta: \begin{cases} \alpha\langle n^* \rangle \beta \leftrightarrow \beta\langle \underline{n}^* \rangle \alpha & d\langle n^* \rangle \alpha \\ c\langle n^* \rangle \alpha \wedge & \neg d\langle n^* \rangle \alpha \\ \quad (\alpha\langle n^* \rangle \beta \leftrightarrow \alpha\langle \underline{n}^* \rangle \beta) & \end{cases}$$



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
  }
```

$I = \forall\alpha, \beta:$ $\begin{cases} \alpha\langle n^* \rangle\beta \leftrightarrow \beta\langle \underline{n}^* \rangle\alpha \\ c\,\langle n^* \rangle\alpha \wedge (\alpha\langle n^* \rangle\beta \leftrightarrow \alpha\langle \underline{n}^* \rangle\beta) \end{cases}$ $\begin{matrix} d\langle n^* \rangle\alpha \\ \neg d\langle n^* \rangle\alpha \end{matrix}$

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

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
  }
```

$I = \forall \alpha, \beta:$ $\begin{cases} \alpha\langle n^* \rangle \beta \leftrightarrow \beta\langle \underline{n}^* \rangle \alpha \\ c\ \langle n^* \rangle \alpha \wedge \\ \quad (\alpha\langle n^* \rangle \beta \leftrightarrow \alpha\langle \underline{n}^* \rangle \beta) \end{cases}$ $\begin{matrix} d\langle n^* \rangle \alpha \\ \neg d\langle n^* \rangle \alpha \end{matrix}$

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

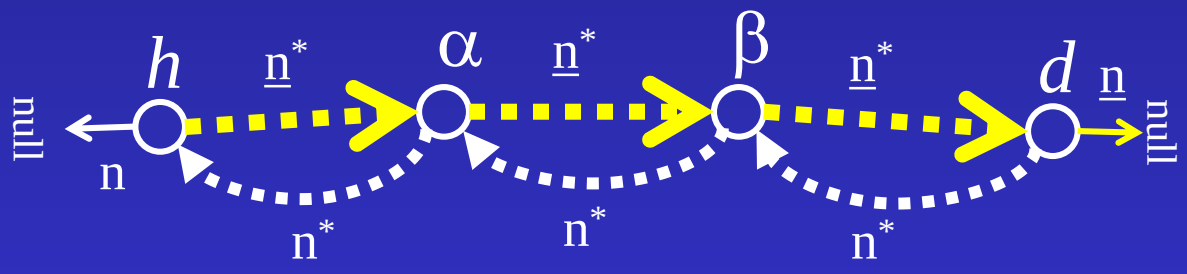
List Reversal (ownership)

$\{\forall \alpha, \beta : h <n^*>\alpha \wedge \beta <n^*>\alpha \rightarrow h <n^*>\beta \}$

```
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)

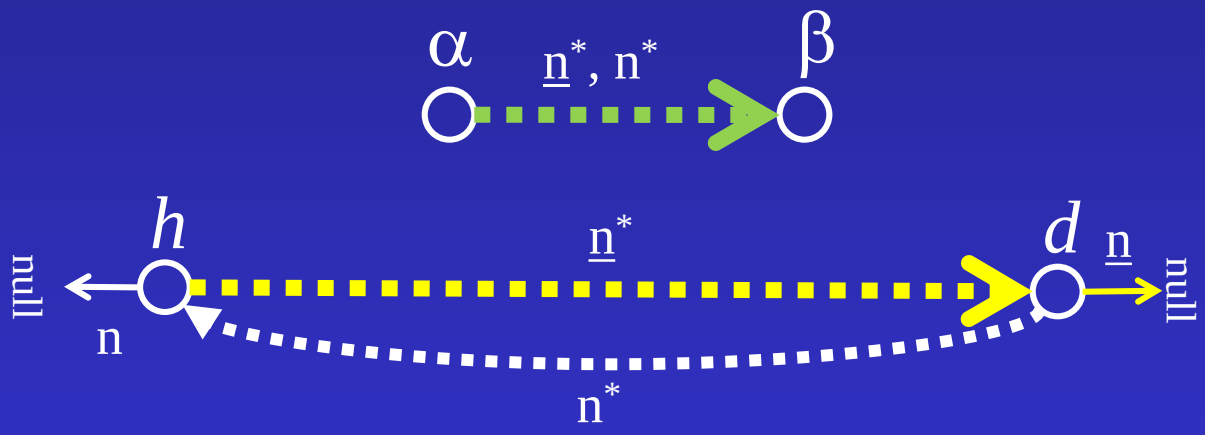
Case 1: $h\langle \underline{n}^* \rangle \alpha \wedge h\langle \underline{n}^* \rangle \beta$



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

List Reversal (ownership)

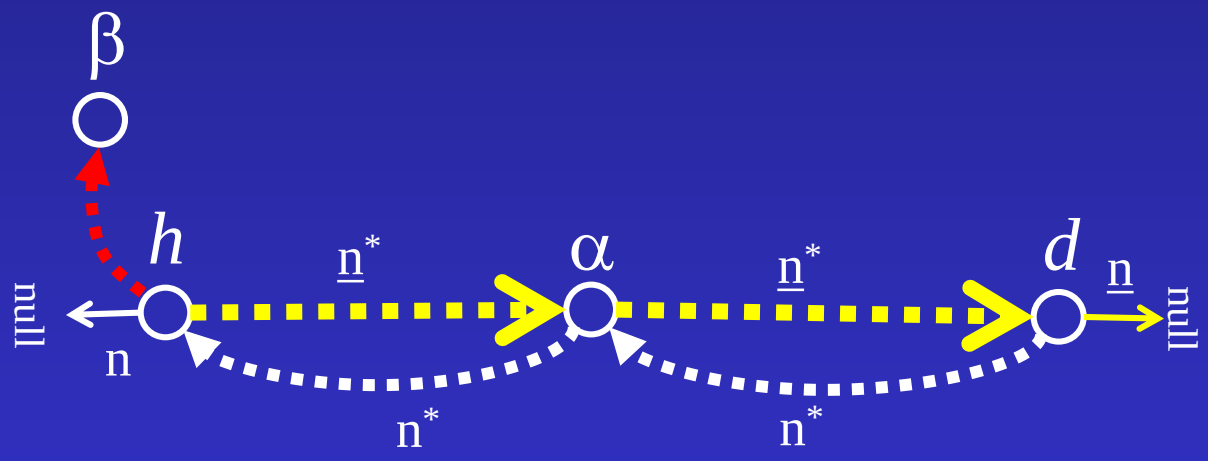
Case 2: $\neg h\langle \underline{n}^* \rangle \alpha \wedge \neg h\langle \underline{n}^* \rangle \beta$



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

List Reversal (ownership)

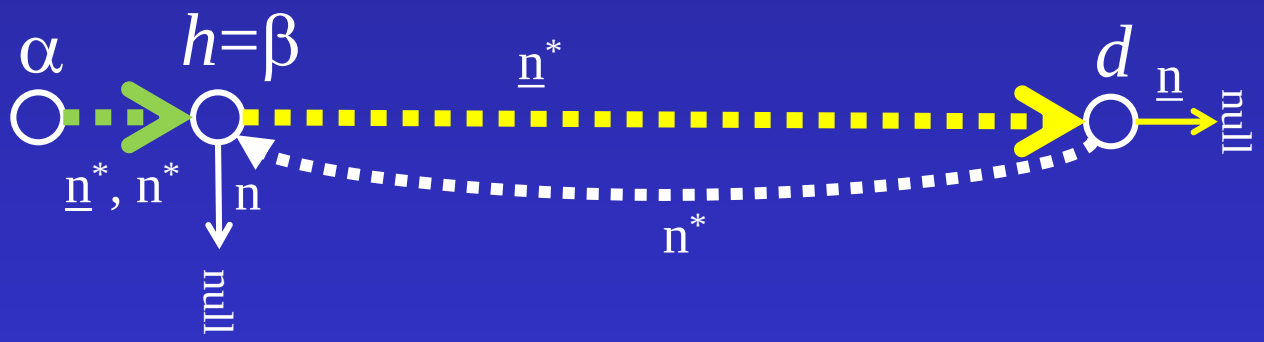
Case 3: $h\langle \underline{n}^* \rangle \alpha \wedge \neg h\langle \underline{n}^* \rangle \beta$



$\alpha\langle n^* \rangle \beta \leftrightarrow$ false

List Reversal (ownership)

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



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

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

  $\Gamma_{linOrd} = \forall\alpha, \beta: n^*(\alpha, \beta) \wedge n^*(\beta, \alpha) \leftrightarrow \alpha = \beta \wedge$

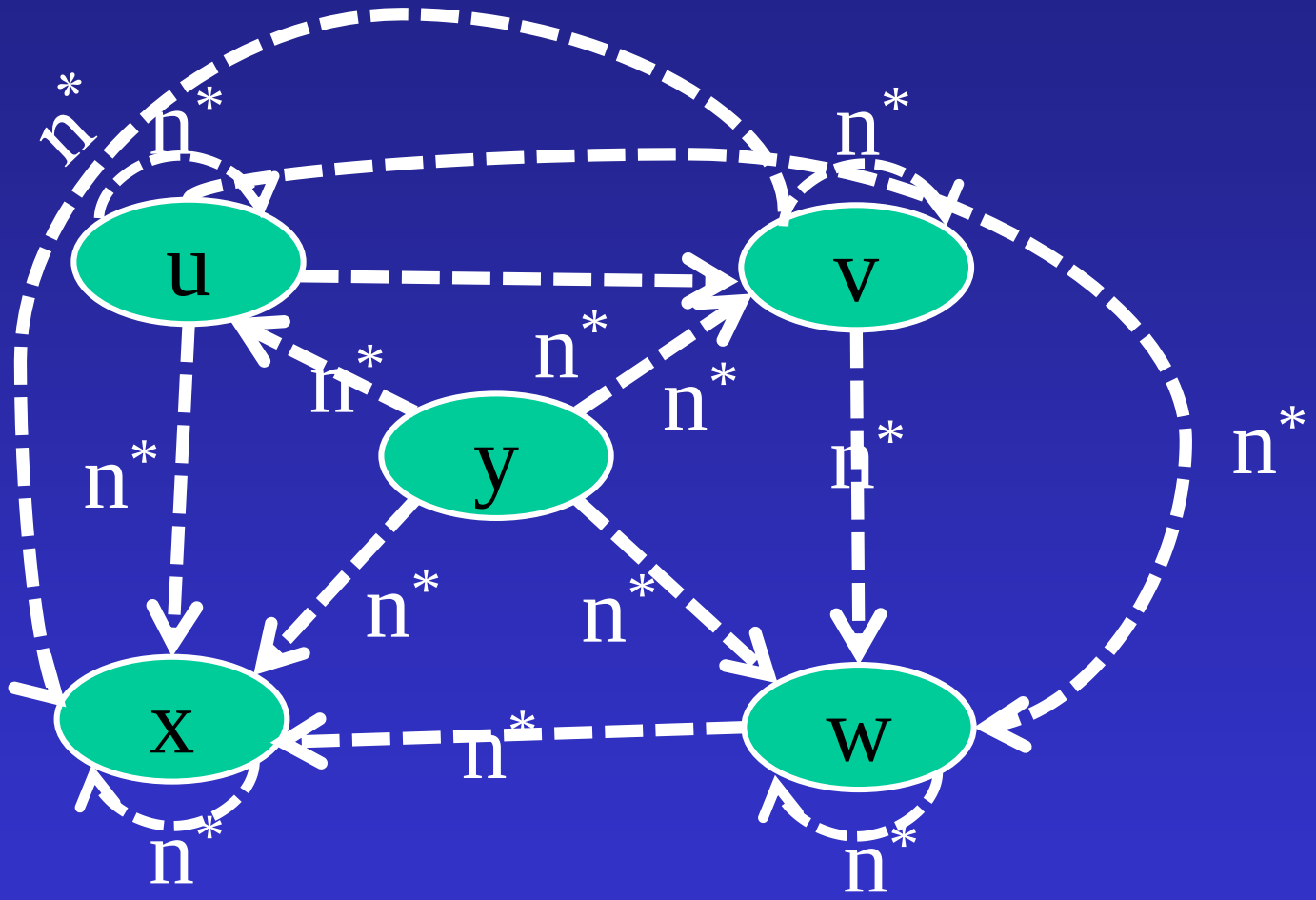
  $\forall\alpha, \beta, \gamma: n^*(\alpha, \beta) \wedge n^*(\beta, \gamma) \rightarrow n^*(\alpha, \gamma) \wedge$

  $\forall\alpha, \beta, \gamma: n^*(\alpha, \beta) \wedge n^*(\alpha, \gamma) \rightarrow (n^*(\beta, \gamma) \vee n^*(\gamma, \beta))$
- φ is satisfiable $\leftrightarrow$ $(\Gamma_{linOrd} \wedge t[\varphi])$ is satisfiable
  - AF formulas have finite model

Inverting $n^* \Rightarrow n$

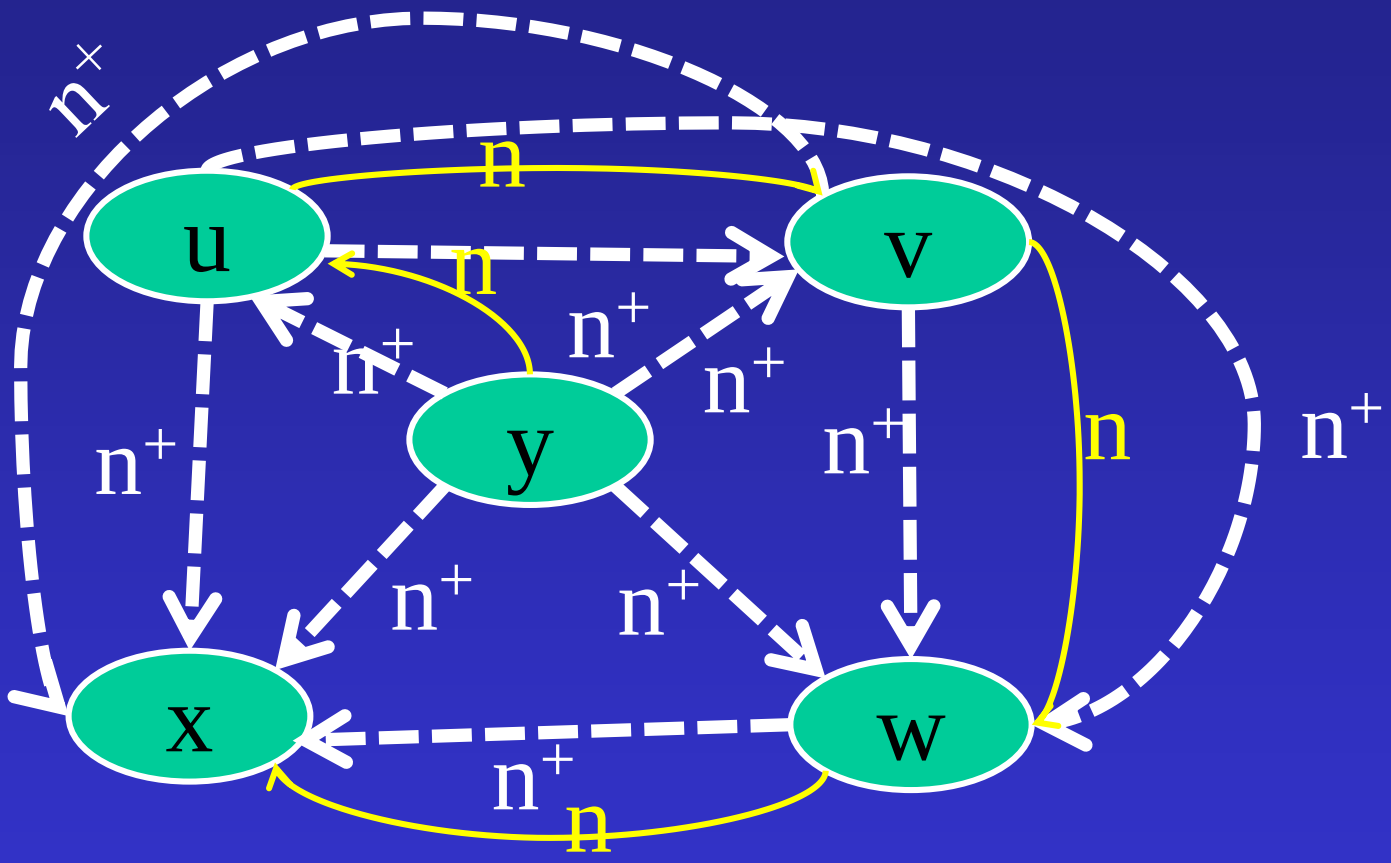
- Every finite model in which n^* satisfies the order requirements:
  $\Gamma_{linOrd}=$
  $\forall \alpha, \beta: n^*(\alpha, \beta) \wedge n^*(\beta, \alpha) \leftrightarrow \alpha = \beta \wedge$
  $\forall \alpha, \beta, \gamma: n^*(\alpha, \beta) \wedge n^*(\beta, \gamma) \rightarrow n^*(\alpha, \gamma) \wedge$
  $\forall \alpha, \beta, \gamma: n^*(\alpha, \beta) \wedge n^*(\alpha, \gamma) \rightarrow (n^*(\beta, \gamma) \vee n^*(\gamma, \beta))$
- n^* uniquely determines n

Inverting n*⇒ n



$\alpha <n^{+}>\beta \leftrightarrow \alpha <n^{*}>\beta \wedge \alpha \neq \beta$

Inverting n*⇒ n



$n(\alpha) = \beta \leftrightarrow \alpha <n^+>\beta \wedge \forall\gamma: \alpha <n^+>\gamma \rightarrow \beta<n^*>\gamma$

Simple SAT Application

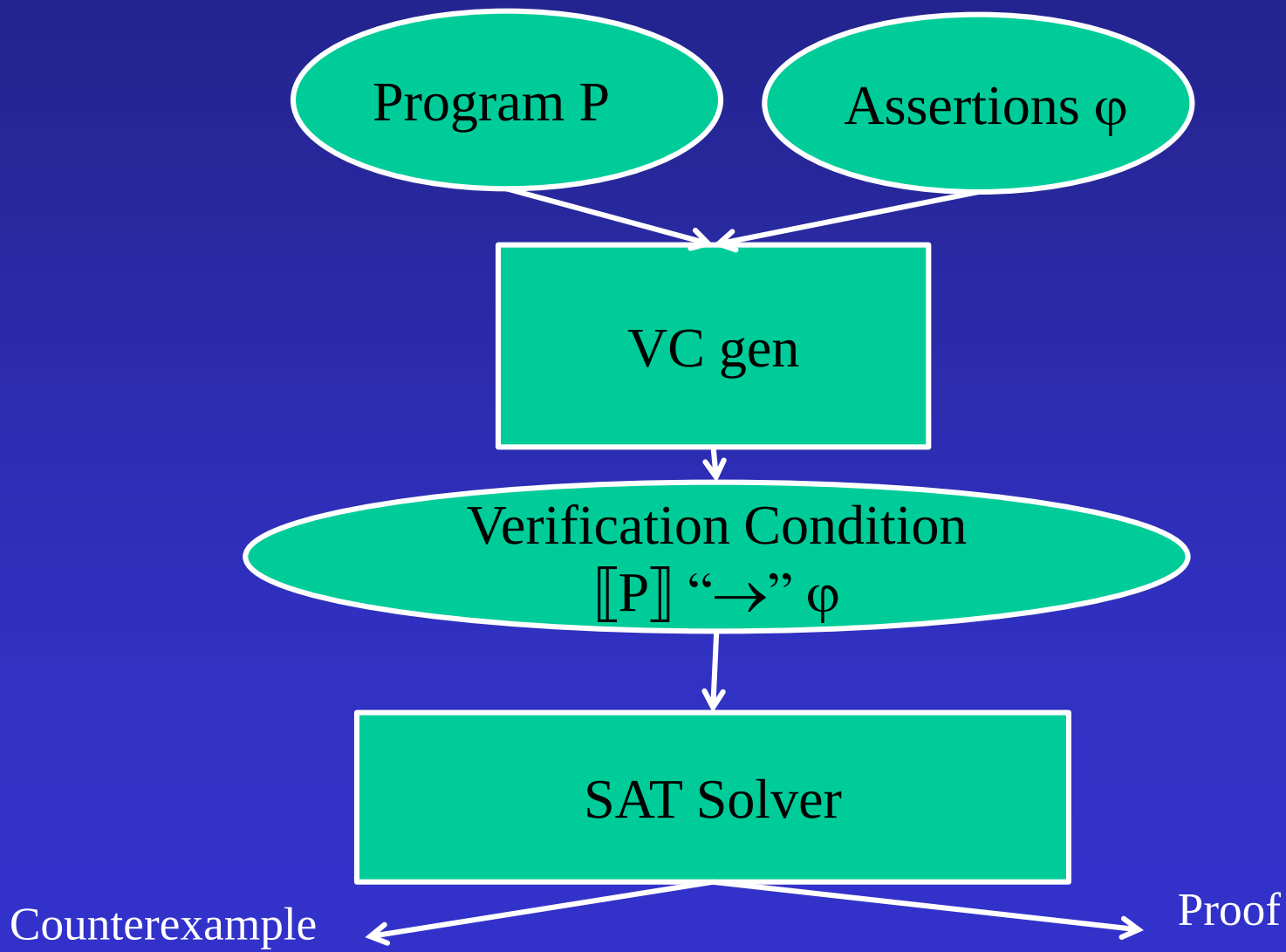
- Determine if two clients are identical
  - Produce isomorphic reachable stores
- reverse(reverse(h)) = h

$$\forall \alpha, \beta: \alpha\langle n_1^* \rangle \beta \leftrightarrow \beta\langle n_0^* \rangle \alpha \wedge$$
$$\forall \alpha, \beta: \alpha\langle n_2^* \rangle \beta \leftrightarrow \beta\langle n_1^* \rangle \alpha$$

$$\Downarrow$$

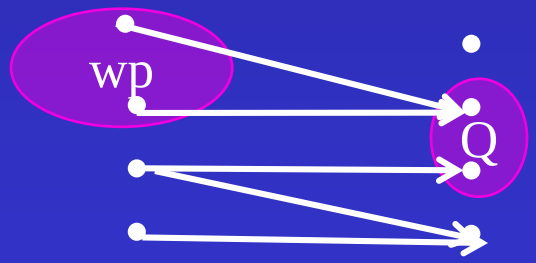
$$\forall \alpha, \beta: \alpha\langle n_0^* \rangle \beta \leftrightarrow \alpha \langle n_2^* \rangle \beta$$

Verification Process



Weakest Precondition

- wp: Stm $\rightarrow$(Ass$\rightarrow$Ass)
- wp $[\![S]\!](Q)$ – the weakest condition such that every terminating computation of S results in a state satisfying Q
- $\sigma \vDash wp\ [\![S]\!](Q) \leftrightarrow \quad \forall \sigma' : \sigma\ [\![S]\!]\ \sigma' \rightarrow \sigma' \vDash Q$



- Can be used to compute verification conditions

Hoare Assignment Rule

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

$$wc\ [\![c := d]\!] \left[\forall \alpha, \beta: \begin{cases} \alpha\langle n^* \rangle \beta \leftrightarrow \beta\langle \underline{n}^* \rangle \alpha & d\langle \underline{n}^* \rangle \alpha \\ c\ \langle n^* \rangle \alpha \wedge & \neg d\langle \underline{n}^* \rangle \alpha \\ \alpha\langle n^* \rangle \beta \leftrightarrow \alpha\langle \underline{n}^* \rangle \beta \end{cases} \right] =$$

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

WP Compound statements

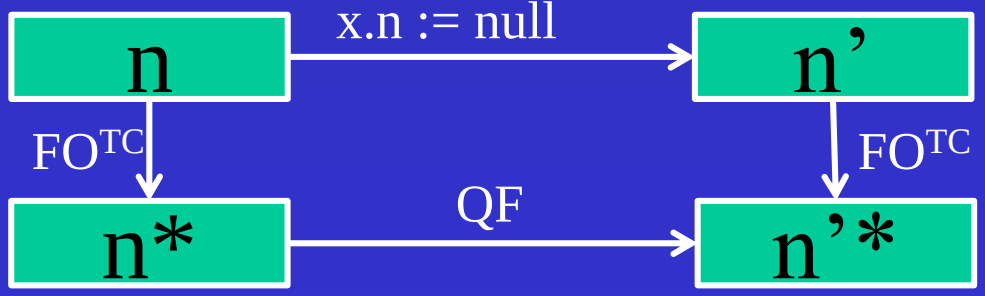
- $wp\ [\![skip]\!](Q) = Q$
- $wp[\![x := e]\!](Q) = Q[e / x]$
- $wp[\![S_1; S_2]\!](Q) = wp[\![S_1]\!](wp[\![S_2]\!](Q))$
- $wp[\![\text{if } B \text{ then } S_1 \text{ else } S_2]\!] = ([\![B]\!] \wedge wp[\![S_1]\!](Q)) \vee (\neg[\![B]\!] \wedge wp[\![S_2]\!](Q))$
- $wp[\![\text{while } B \text{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

VC rules

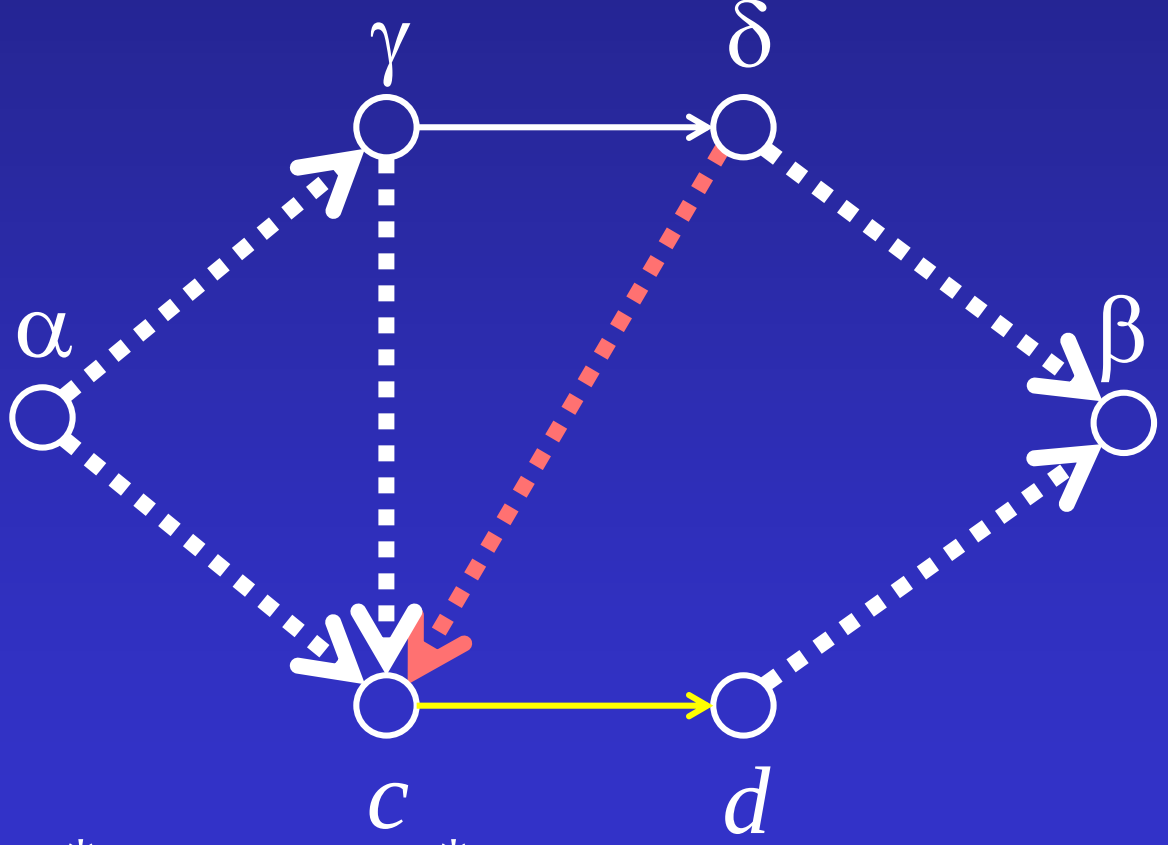
- $VC_{gen}(\{P\}\ S\ \{Q\}) = P \rightarrow wp[\![S]\!](Q) \wedge \bigwedge VC_{aux}(S, Q)$
- $VC_{aux}(S, Q) = \{\}$ (for any atomic statement)
- $VC_{aux}(S_1; S_2, Q) =$
  $VC_{aux}(S_1, wp(S_2, Q)) \cup VC_{aux}(S_2, Q)$
- $VC_{aux}(\text{if C then } S_1 \text{ else } S_2, Q) =$
  $VC_{aux}(S_1,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

But how about heap mutations?

- McCarthy assignment rule does not work
- wp⟦c.n := null⟧(Q) = Q[n[c↦null] / n]
  - Refers to n
  - Does not explicitly update reachability
  - Outside AF^R
- Employ incremental updates



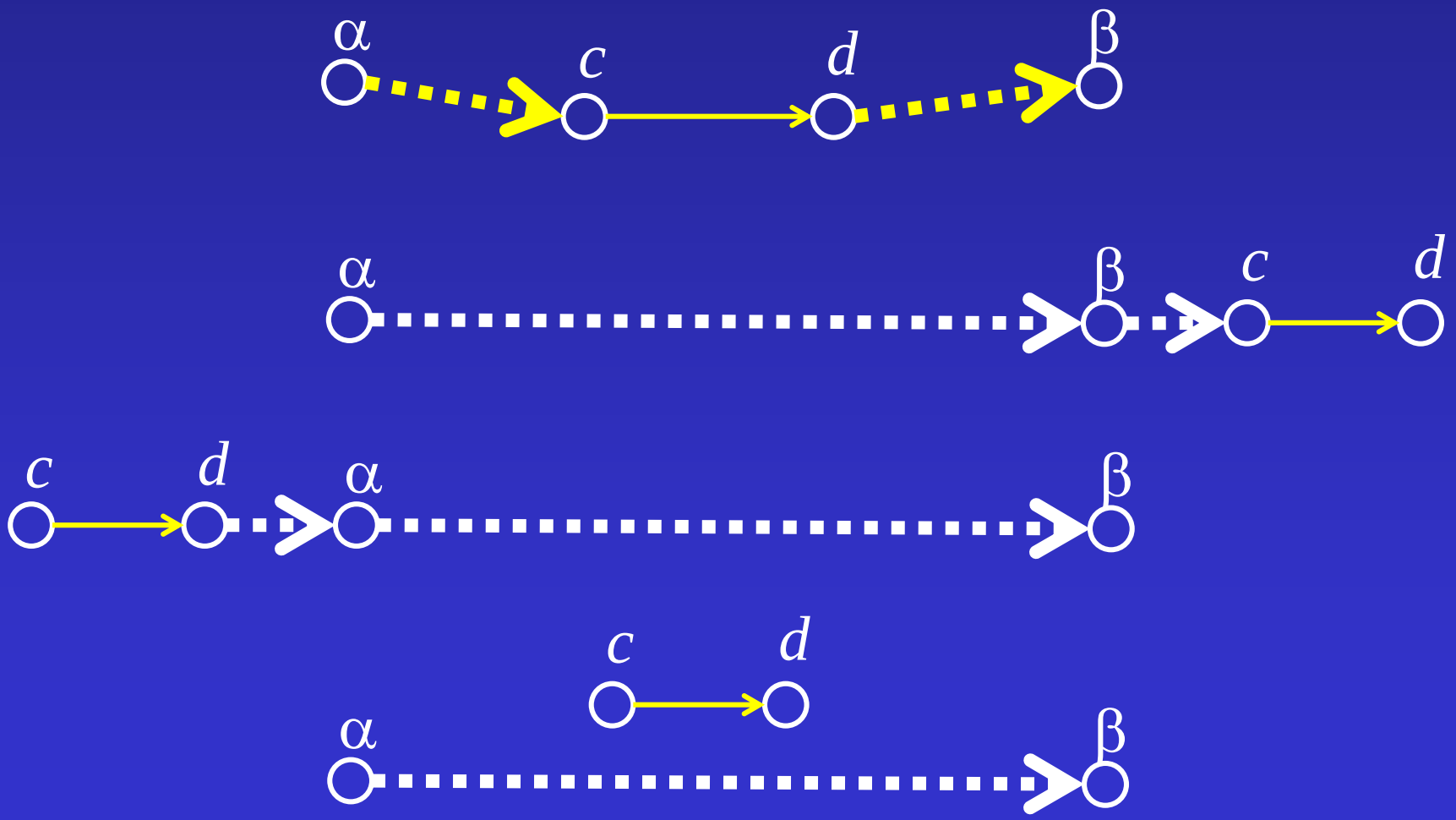
Dong & Su [SIGMOD'00] DAG



$\exists\gamma$: $\alpha<n^*>\gamma \wedge \gamma<n^*>c \wedge$

$n(\gamma)=\delta \wedge \delta<n^*>\beta \wedge \neg\delta<n^*>c$

Deterministic Graphs (function)



Mutating Single Linked Lists

- $wp[\![c.n := null]\!](Q) =$

  $Q[(\alpha<n^*>\beta \wedge (\neg\alpha<n^*>c \vee \beta<n^*>c)) \,/\, \alpha<n^*>\beta]$
- Can also enforce absence of null dereferences

  $c \neq null$

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 =null)

- Simple for general graphs
- AF^R for arbitrary data structures
- wp $[\![c.n := y]\!](Q) = Q[(\alpha<n^*>\beta \vee (\alpha<n^*>c \wedge y<n^*>\beta))/ \alpha<n^*>\beta]$
- Can also enforce acyclicity $\neg y<n^*>c$

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

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

VC rules

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

Pointer Traversals

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

Backward Reasoning with WP

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

d := e.n ;

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

d.n := null ;

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

d.n := c ;

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

Backward Reasoning with WP

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

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

d := e.n ;

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

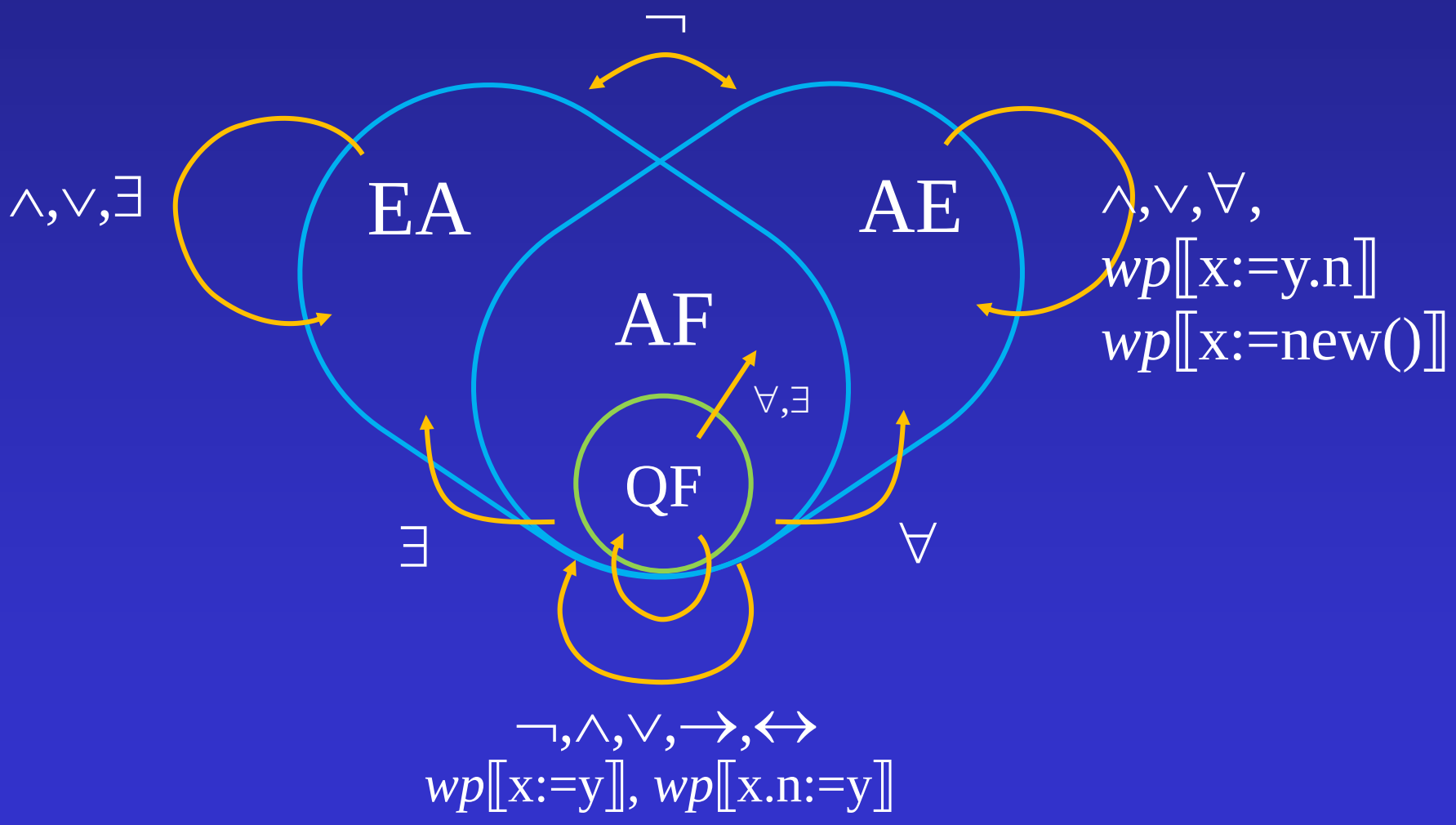
d.n := null ;

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

d.n := c ;

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

Closure Properties



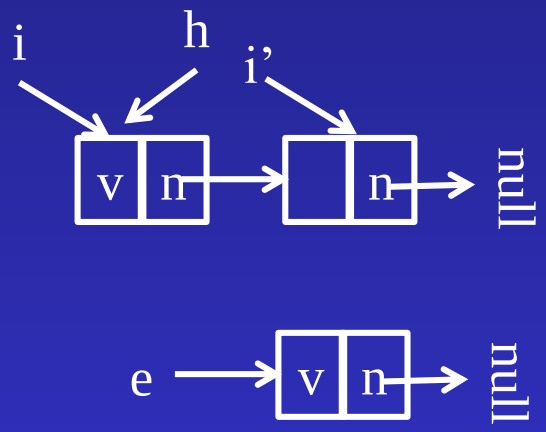
Benchmark	*Formula Size*						*Solving time* (Z3)
	P,Q		I		VC		
	#	∀	#	∀	#	∀	
SLL: reverse	2	2	11	2	133	3	57ms
SLL: filter	5	1	14	1	280	4	39ms
SLL: create	1	0	1	0	36	3	13ms
SLL: delete	5	0	12	1	152	3	23ms
SLL: deleteAll	3	2	7	2	106	3	32ms
SLL: insert	8	1	6	1	178	3	17ms
SLL: find	7	1	7	1	64	3	15ms
SLL: 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			
		#	∀	#	∀	#	∀		
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;
}
```



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

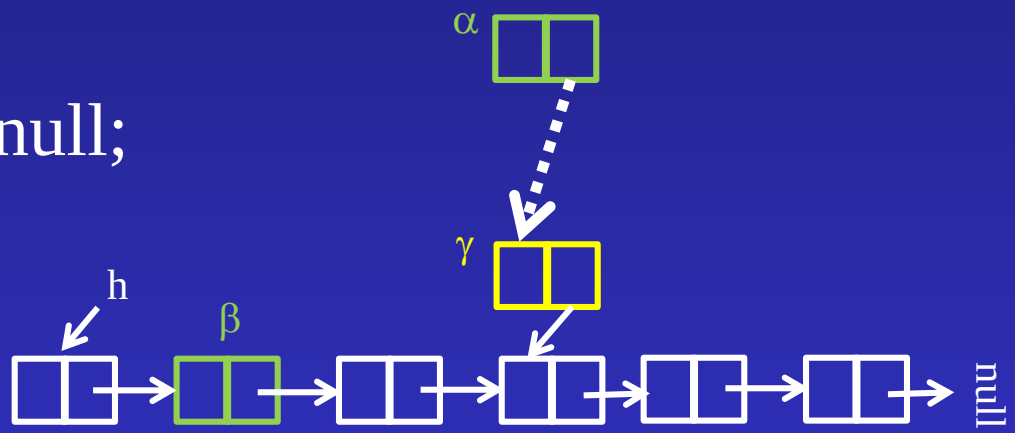
Data Structures outside AF^R

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

List Reversal (general)

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

```
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
}
```



$\forall \alpha, \beta : \alpha <n^*> \beta \leftrightarrow$

$$\begin{cases} \beta <\underline{n}^*> \alpha \\ \alpha <\underline{n}^*> \beta \\ \text{false} \\ \exists \gamma : \alpha <\underline{n}^*> \gamma \wedge \neg h <\underline{n}^*> \gamma \wedge \beta <\underline{n}^*> n(\gamma) \end{cases}$$

$$\begin{aligned} & h <\underline{n}^*> \alpha \wedge h <\underline{n}^*> \beta \\ \neg & h <\underline{n}^*> \alpha \wedge \neg\, h <\underline{n}^*> \beta \\ & h <\underline{n}^*> \alpha \wedge \neg\, h <\underline{n}^*> \beta \\ \neg & h <\underline{n}^*> \alpha \wedge h <\underline{n}^*> \beta \end{aligned}$$

Related Work

- Axiomatizing Rechability
  - [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