Effectively-Propositional Modular Reasoning about Reachability in Linked Data Structures
CAV'13, POPL'14

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

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

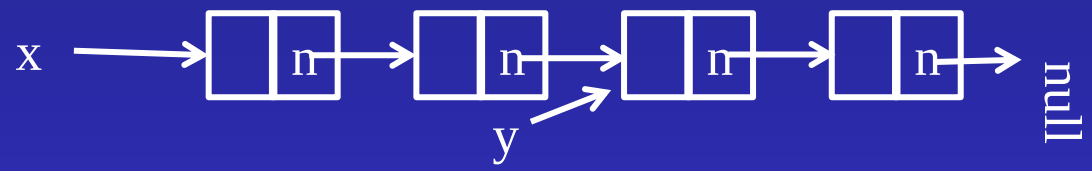
Motivation

- 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

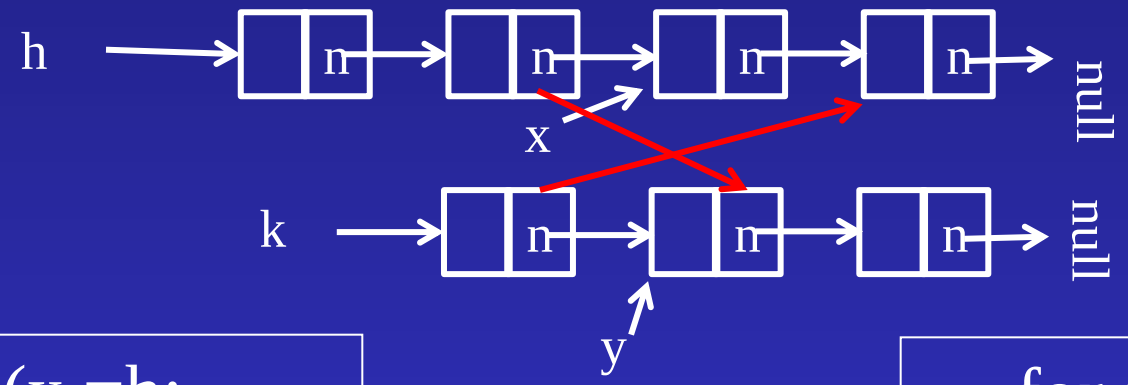
$\{x <n^*> y\}$

```
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 (not even RE)
- Modularity
  - How to specify the behavior on reachability independent of the call
- [Inferring reachability properties from the code and some assertions]

Link list manipulations are simple

- Simple to reason about correctness
  - Small counterexamples
- Even for doubly/circular/nested lists
- “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

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

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

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

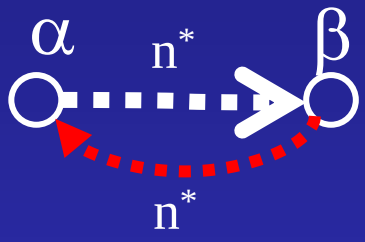
$$=_{sat} (P_{11} \leftrightarrow P_{12}) \wedge (P_{12} \leftrightarrow P_{22})$$

Alternation Free Reachability (AF^R)

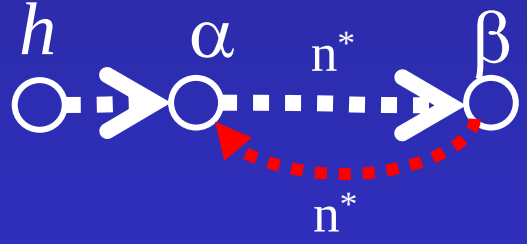
- "Extended subset" of EA
  - Closed under negation
- t ::= var | constant (Terms)
- ap ::= $t_1 = t_2$ | $r(t_1,t_2, \ldots, t_n)$
  | $t_1 <f^*> t_2$ (Reachability via sequences of f's)
  (exists k: $f^k(t_1)=t_2$)
- qf ::= qf | $qf_1 \wedge qf_2$ | $qf_1 \vee qf_2$ | $\neg qf$
- e ::= $\exists \alpha_1,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

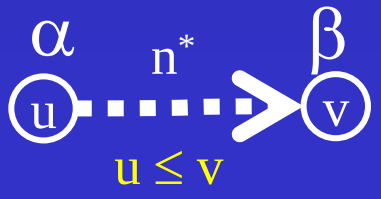
- Acylicity (>2)
  - $\forall \alpha, \beta: \alpha<n^*>\beta \wedge \beta<n^*>\alpha \rightarrow \alpha=\beta$



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

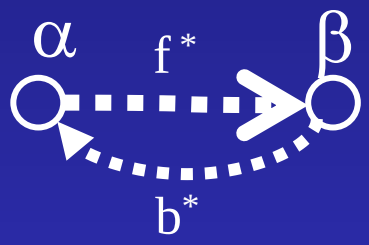


- Sorted segment
  - $\forall \alpha,\beta: \alpha<n^*>\beta \rightarrow \alpha \leq_{data} \beta$

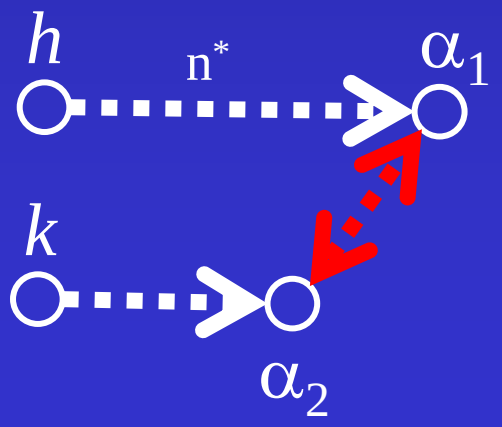


AF^R Program Properties

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



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



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
  }
```

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

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

Why AF^R?

- Represent invariants of simple linked list manipulations
- Closed under $\vee$, $\wedge$, $\neg$, $\rightarrow$, wp$[\![$x.n :=y$]\!]$
- 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

$AF^R \propto AF$

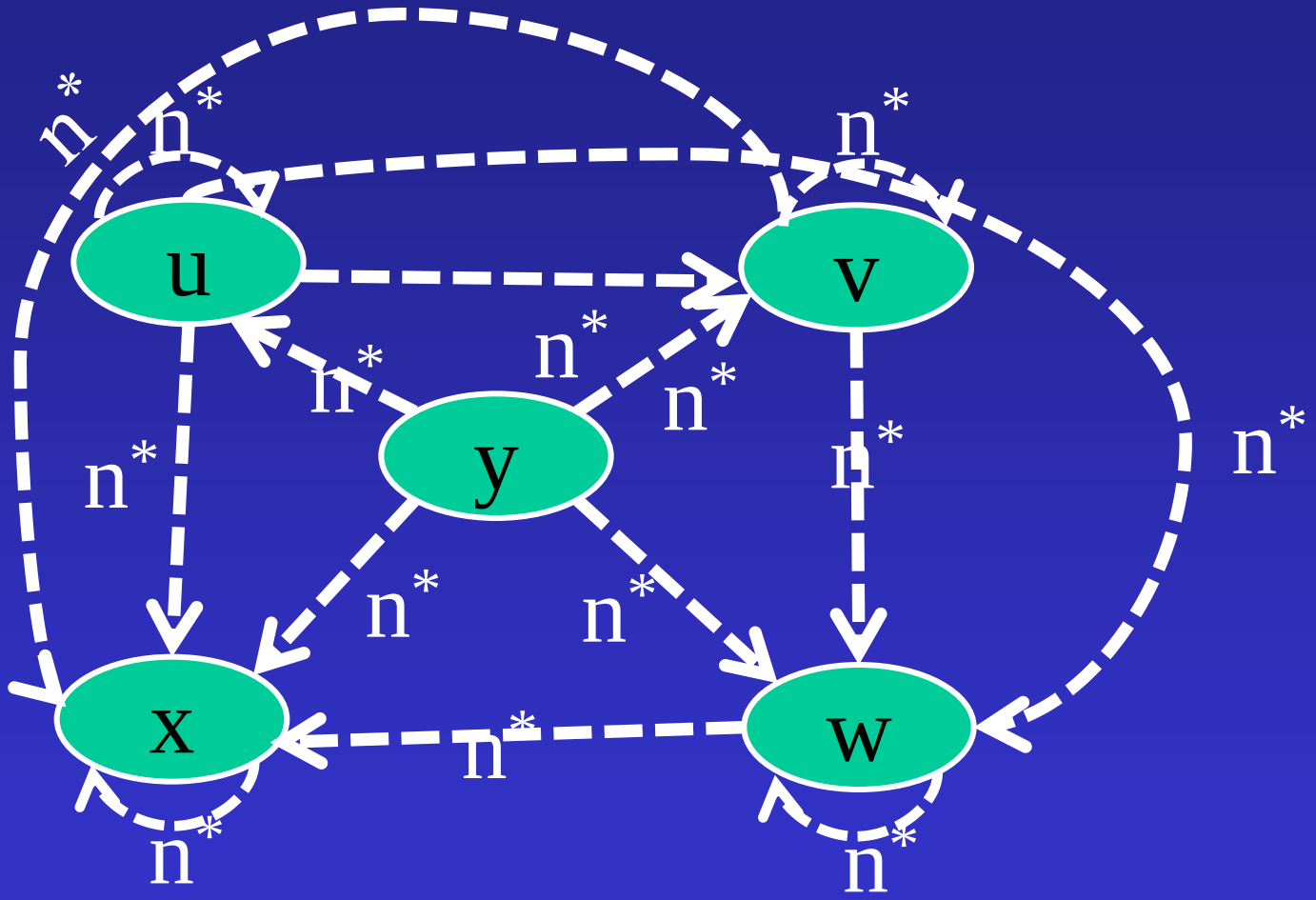
- Introduce an auxiliary relation n*
- $t[\alpha <n^*>\beta] = n^*(\alpha, \beta)$
- Axiomatize n* by an AF formula $\Gamma_{linOrd} = \forall \alpha, \beta:$
  $n^*(\alpha, \beta) \wedge n^*(\beta, \alpha) \rightarrow \alpha = \beta \wedge$
  $\forall \alpha: n^*(\alpha, \alpha)$
  $\forall \alpha, \beta, \gamma: n^*(\alpha, \beta) \wedge n^*(\beta, \gamma) \rightarrow n^*(\alpha, \gamma) \wedge$
  $\forall \alpha, \beta, \gamma: n^*(\alpha, \beta) \wedge n^*(\alpha, \gamma) \rightarrow (n^*(\beta, \gamma) \vee n^*(\gamma, \beta))$
- Completeness

  φ is satisfiable $\leftrightarrow$ $(\Gamma_{linOrd} \wedge t[\varphi])$ is satisfiable
  - AF formulas have finite model

Inverting $n^* \Rightarrow n$

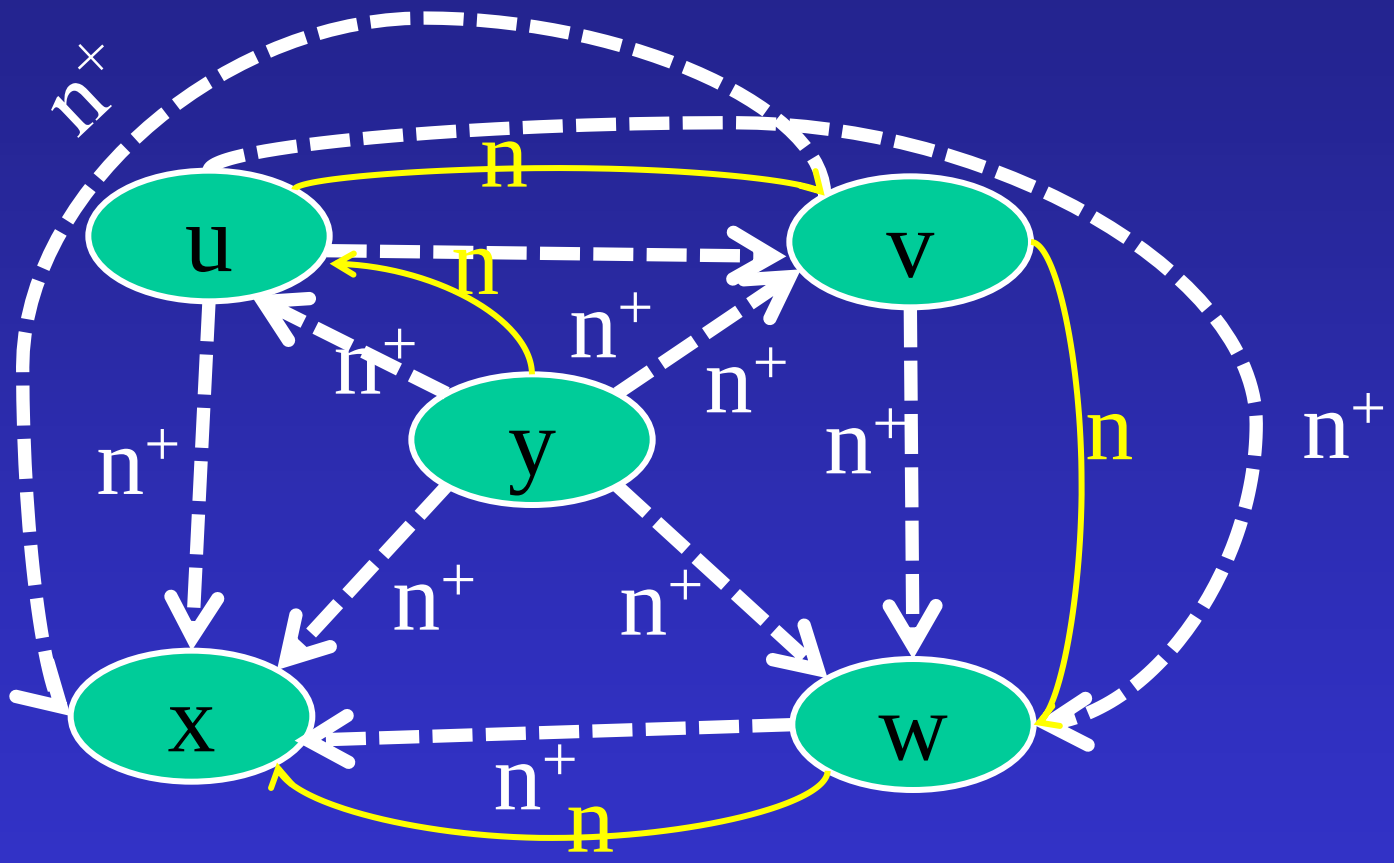
- Every finite model in which n^* satisfies the order requirements:
  $\forall \alpha, \beta: n^*(\alpha, \beta) \wedge n^*(\beta, \alpha) \rightarrow \alpha = \beta \ \wedge$
  $\forall \alpha: n^*(\alpha, \alpha) \wedge$
  $\forall \alpha, \beta, \gamma: n^*(\alpha, \beta) \wedge n^*(\beta, \gamma)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

Inverting n*⇒ n



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

Inverting n*⇒ n



‘n(α)=β’ ↔α <n+>β ∧∀γ: α <n+>γ→β<n*>γ

Simple SAT Application

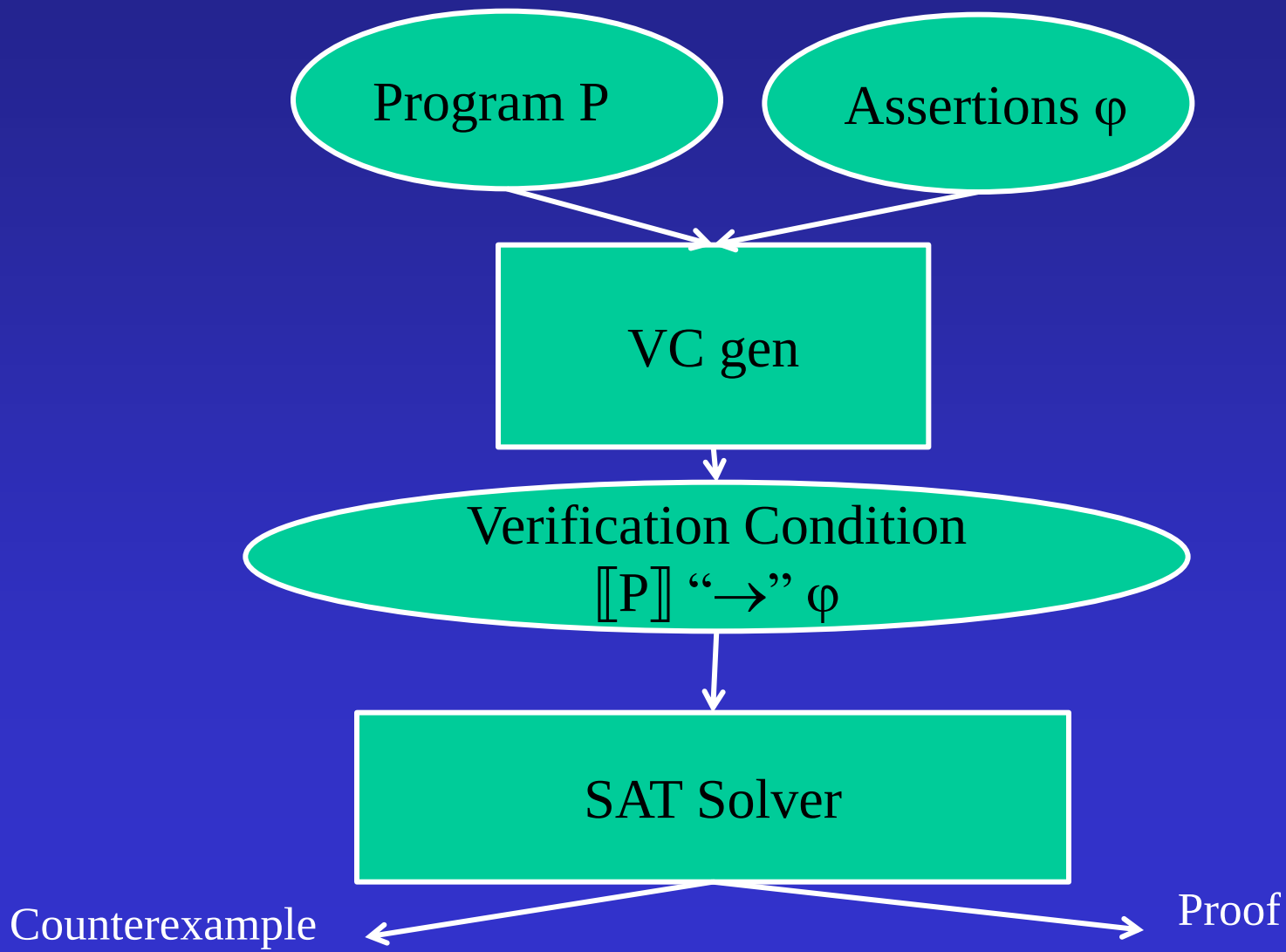
- Determine if two clients are identical
  - Produce isomorphic reachable stores
- reverse(reverse(h)) = h

$$\forall \alpha, \beta: \alpha <n_1^*> \beta \leftrightarrow \beta <n_0^*> \alpha \wedge$$
$$\forall \alpha, \beta: \alpha <n_2^*> \beta \leftrightarrow \beta <n_1^*> \alpha$$

$$\Downarrow$$

$$\forall \alpha, \beta: \alpha <n_0^*> \beta \leftrightarrow \alpha <n_2^*> \beta$$

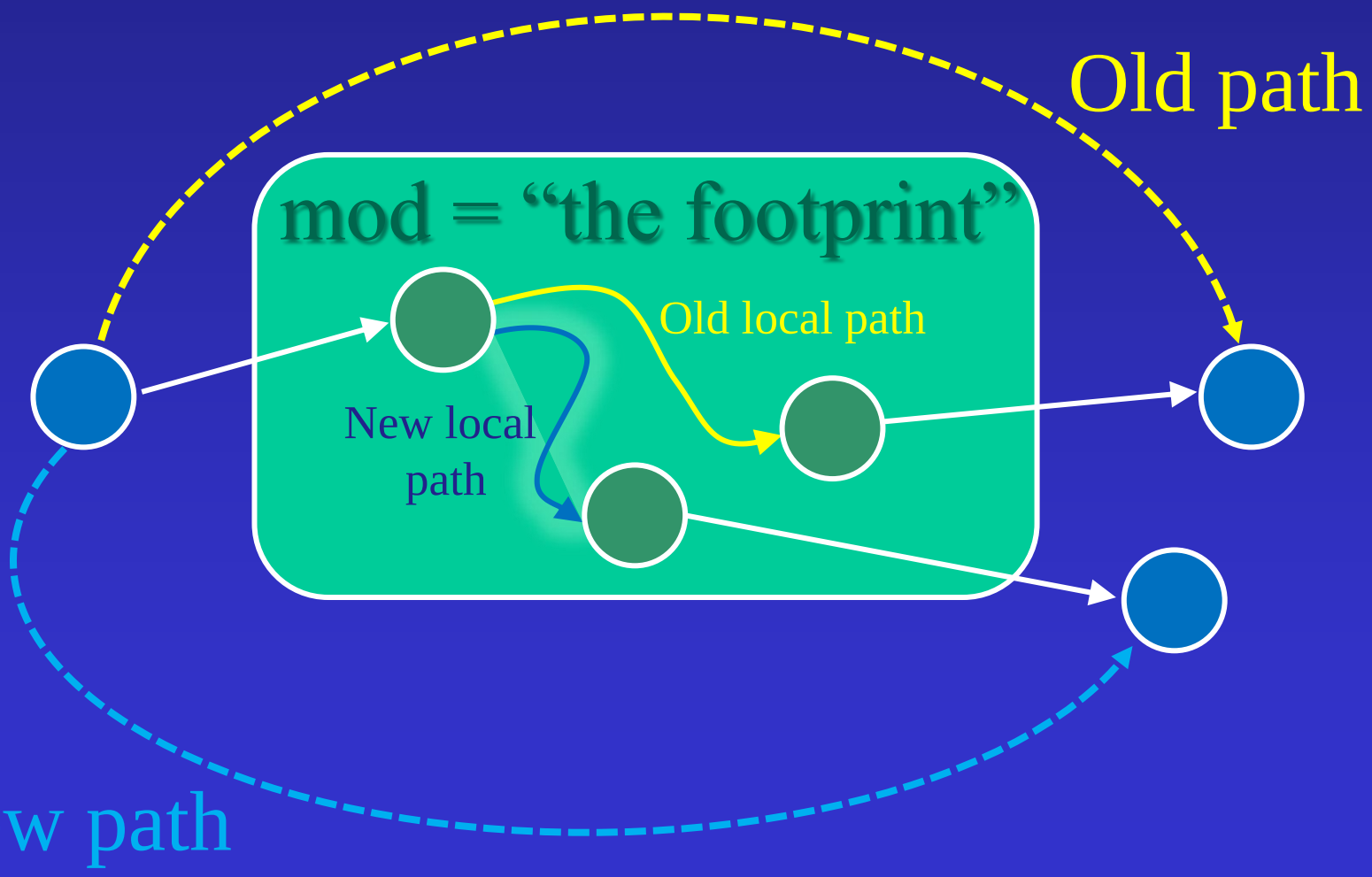
Verification Process



Modular Specification

- Every procedure mutates a limited part of the heap
  - footprint
- Can we specify the effect of the procedure on the footprint only?
- Provide a general adaptation rule for the context
  - Possible for second order logics
    - Transitive closure
    - Separation Logic
- Can this be done in a weak logic?

An Adaptation Rule



List Reversal (isolated)

{ac [h]$\wedge\forall\alpha$: 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
  }
```

{?}

reverse(i);

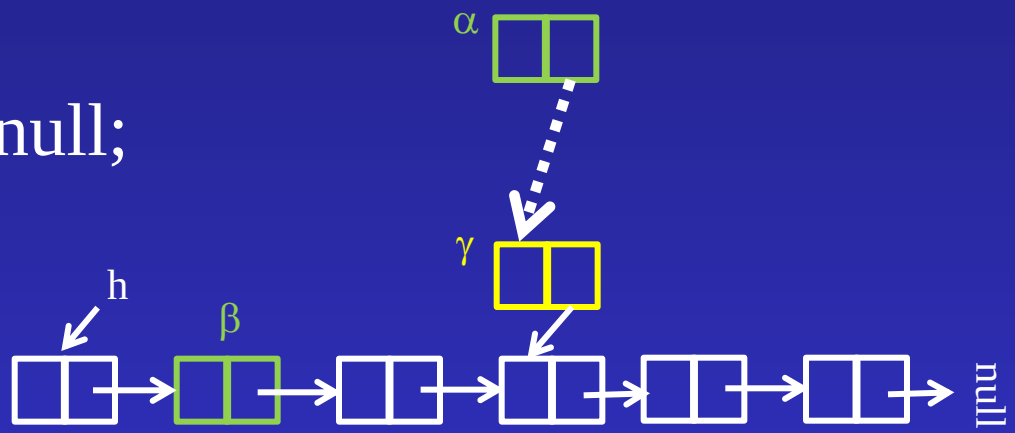
{ac[i]] $\wedge$i<n*>j}

{ac[d] $\wedge$ $\forall\alpha, \beta$: α<n^*>β $\leftrightarrow$ β<$\underline{n}^*$>α $\wedge$ $\forall\alpha$: d <n^*>α}

When ownership breaks

{ac [h]}

```
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
}
```



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

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

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

false

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

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

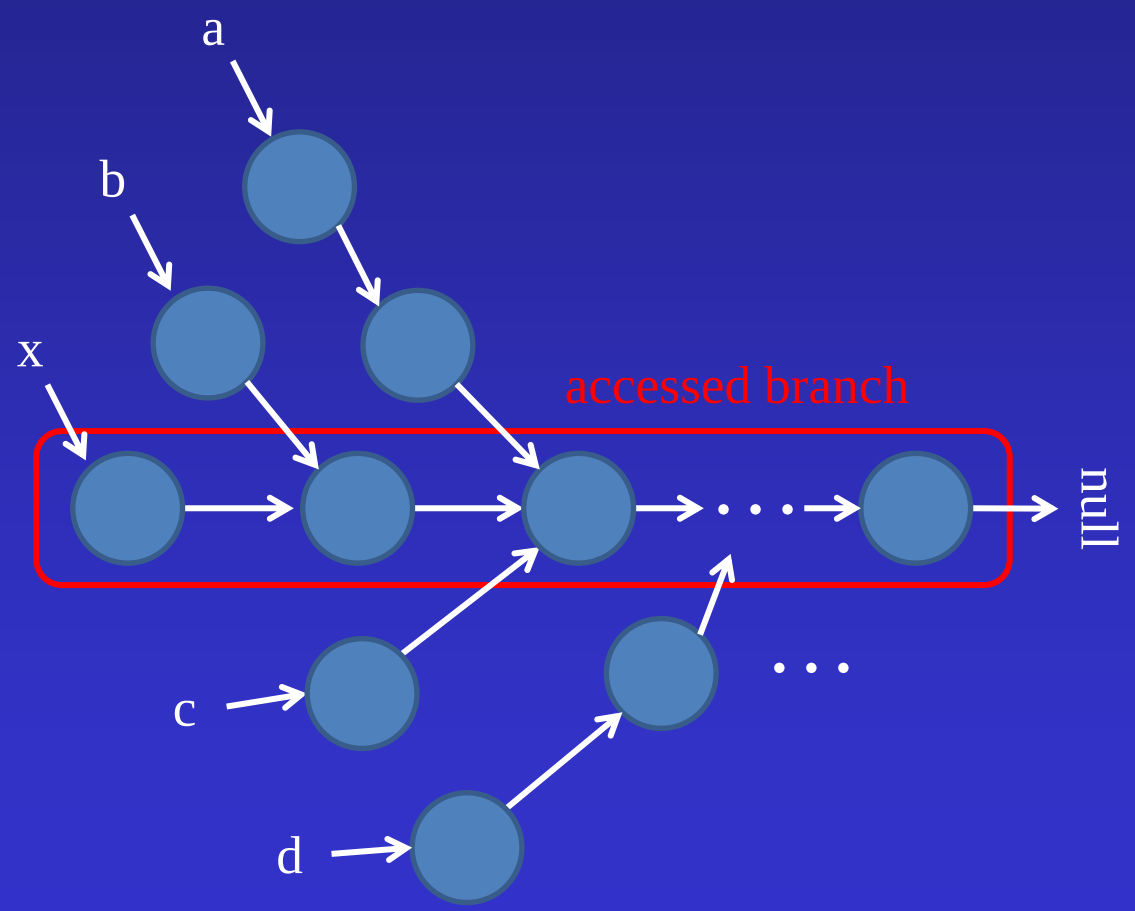
$\neg h<\underline{n}^*>\alpha \wedge \neg\, h<\underline{n}^*>\beta$

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

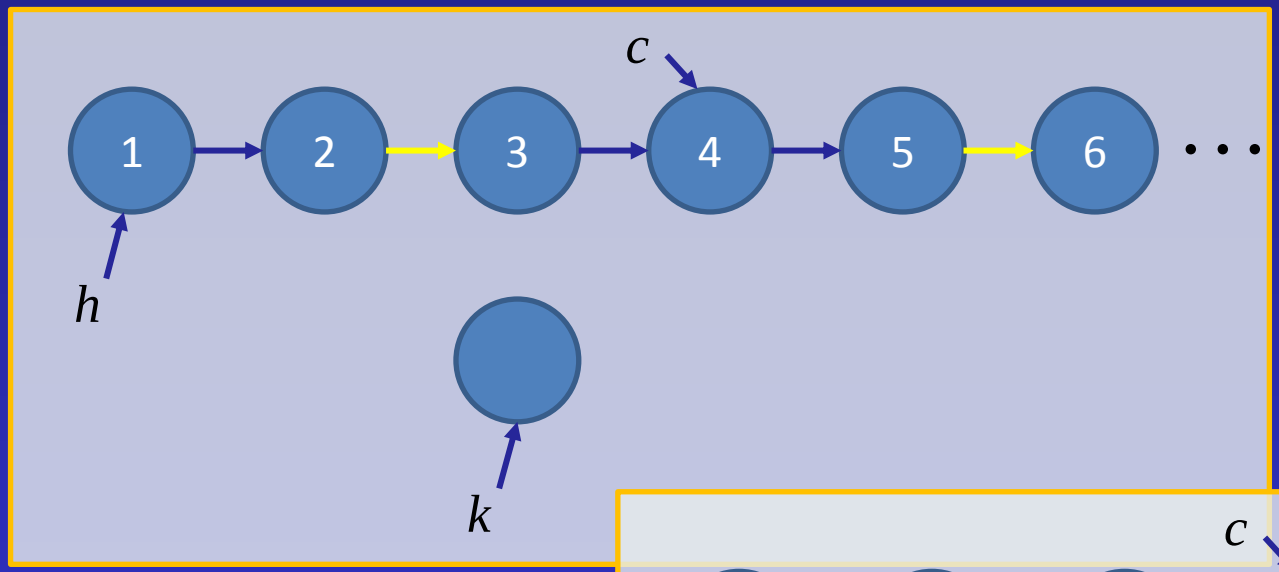
$\neg h<\underline{n}^*>\alpha \wedge h<\underline{n}^*>\beta$

Unbounded Cutpoints

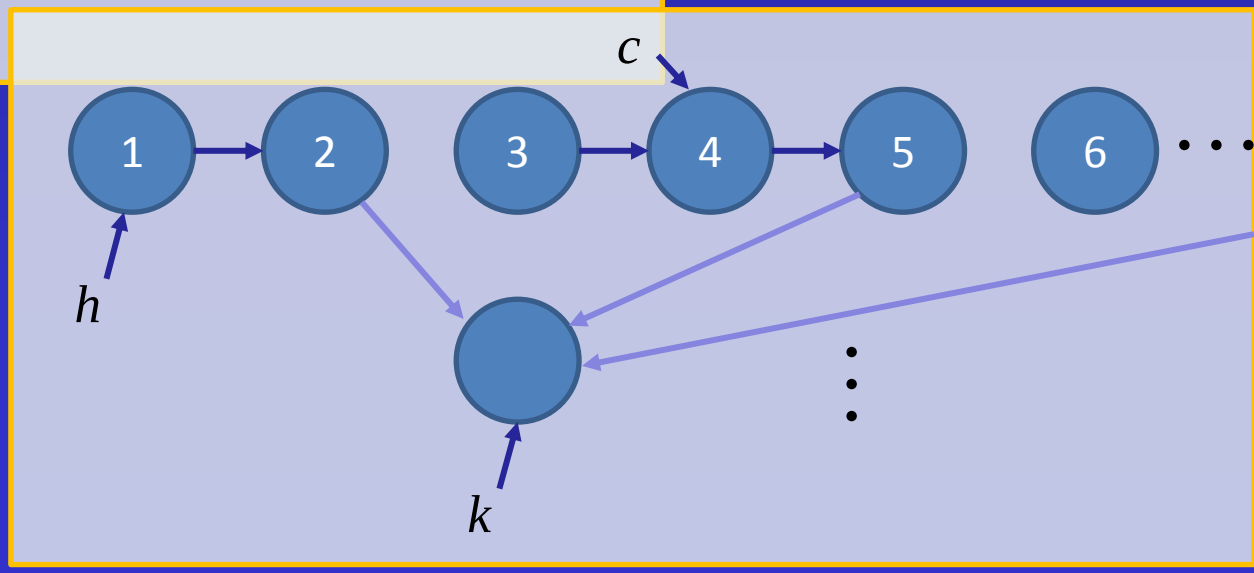
```
Node find(Node x) {
  Node i = x.p;
  if (i != null) {
    i = find(i);
    x.p = i;
  }
  else i = x;
  return i;
}
```



Complicated Mutations



$h\langle n^* \rangle c \wedge \neg c\langle n^* \rangle k$

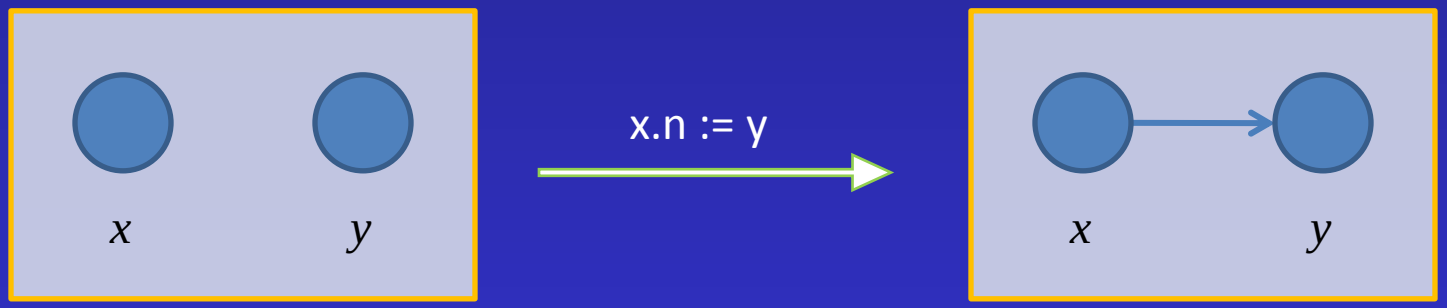


$\neg h\langle n^* \rangle c \wedge c\langle n^* \rangle k$

Limited Programming Model

- Type correct programs
- Recursion instead of loops
- Limited amount of new sharing per call
- Deterministic transitive closure
- Specified changes
- Uniform changes
  - Fixed number of contiguous intervals of changed parts of the heap

Small Footprint Destructive Updates



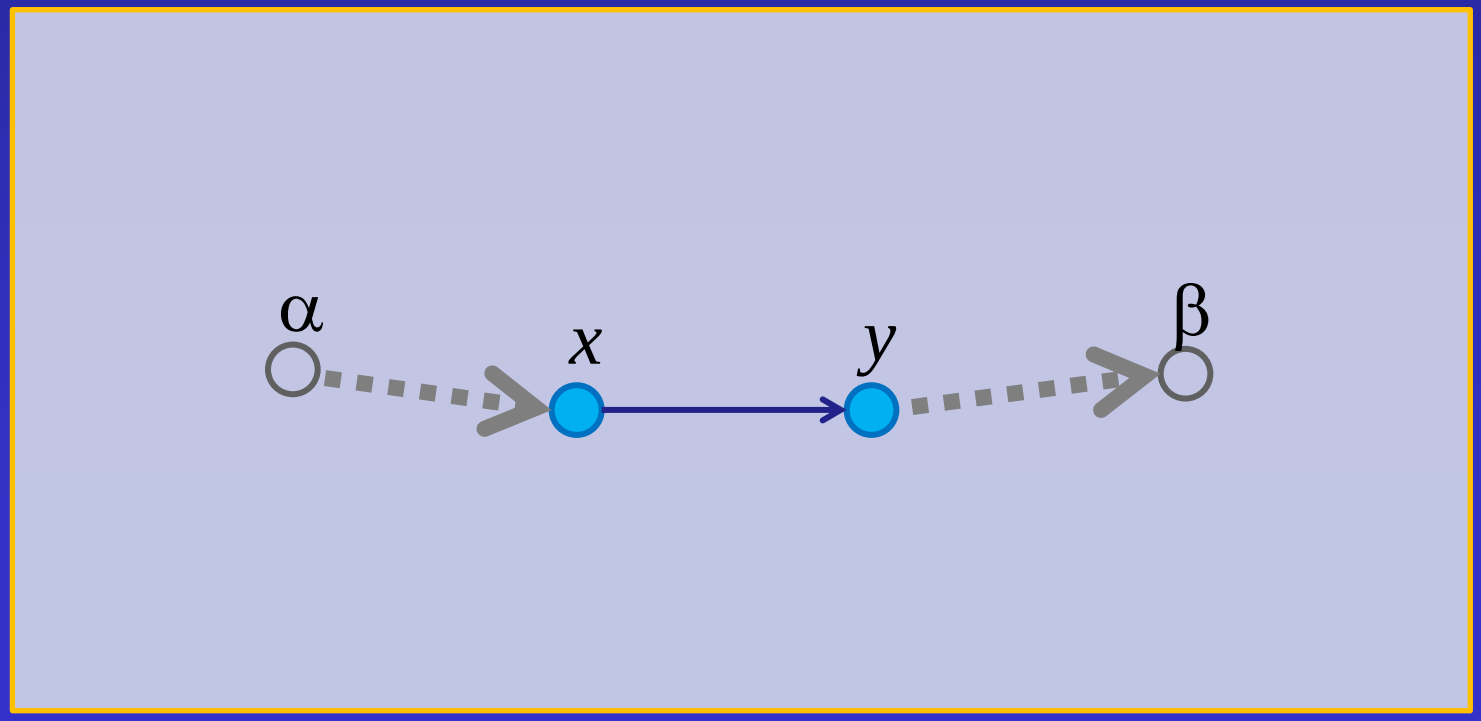
$x \neq \text{null} \wedge x\langle n \rangle \text{null}$

$x\langle n^* \rangle y \wedge \neg y\langle n^* \rangle x$

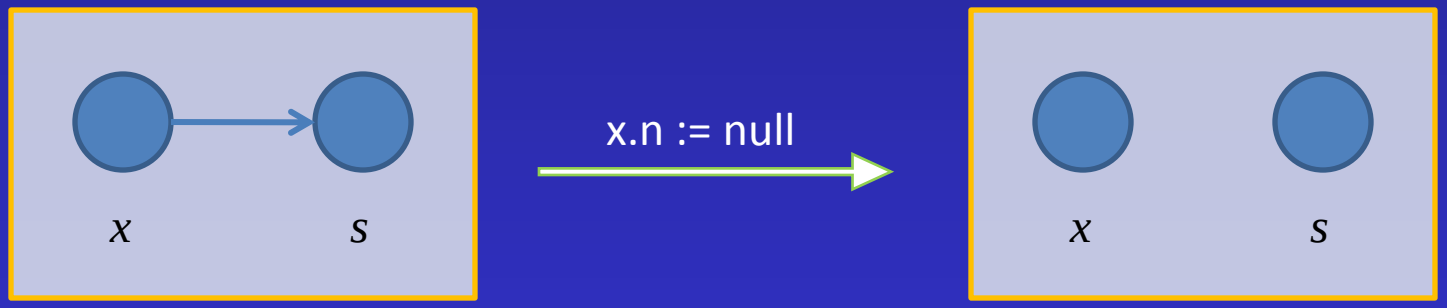
Large Footprint Destructive Updates

x.n := y

$$\alpha\langle n^* \rangle \beta \leftrightarrow \alpha\langle \underline{n}^* \rangle \beta \vee (\neg \alpha\langle \underline{n}^* \rangle x \wedge y\langle \underline{n}^* \rangle \beta))$$



Small Footprint Destructive Updates



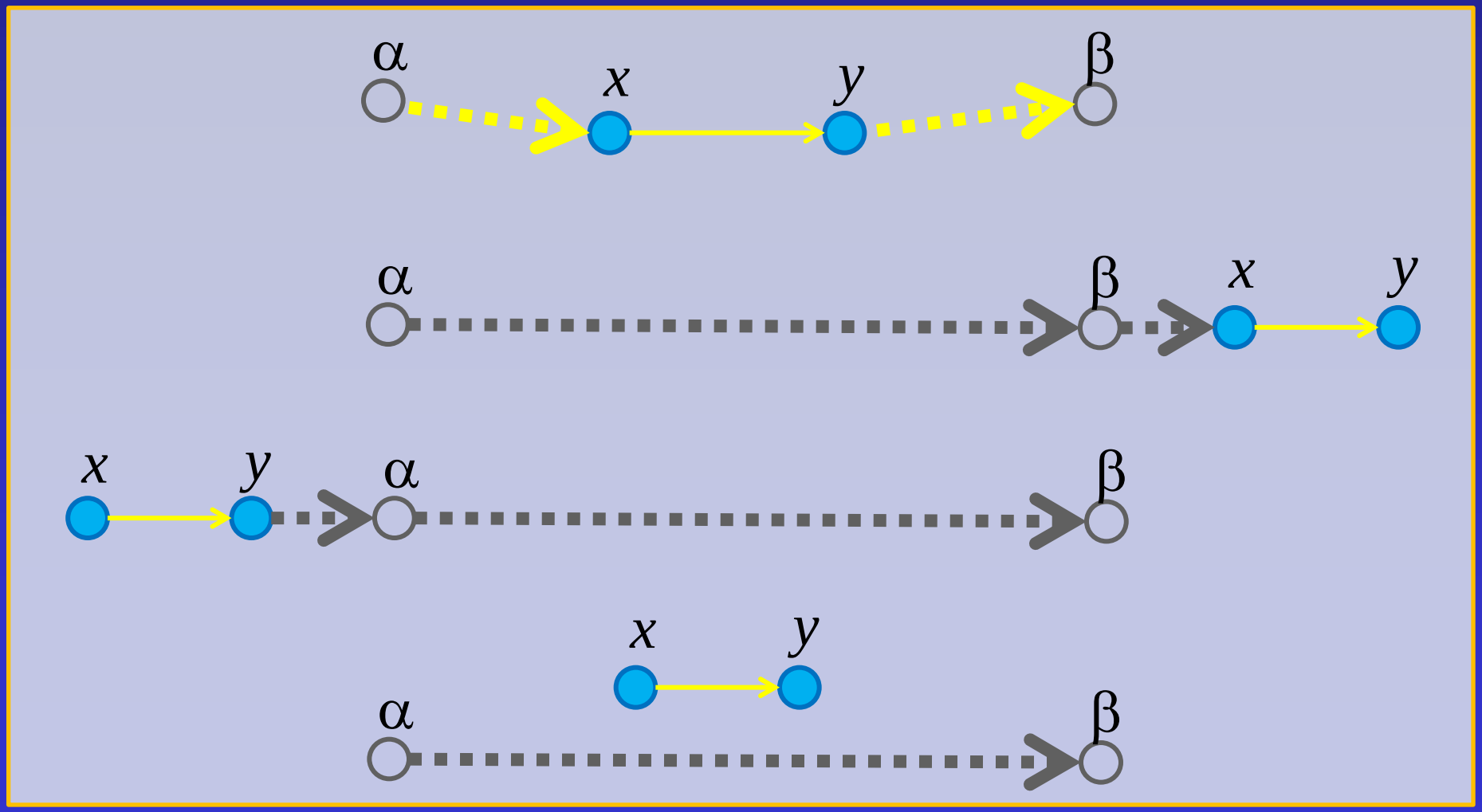
$x \neq \text{null} \wedge x\langle n \rangle s$

$\neg x\langle n^* \rangle s \wedge \neg s\langle n^* \rangle x$

Large Footprint

x.n := null

$$\alpha\langle n^* \rangle \beta \leftrightarrow \alpha\langle \underline{n}^* \rangle \beta \wedge (\neg \alpha\langle \underline{n}^* \rangle x \vee \beta\langle \underline{n}^* \rangle x))$$



Small Footprint Reverse

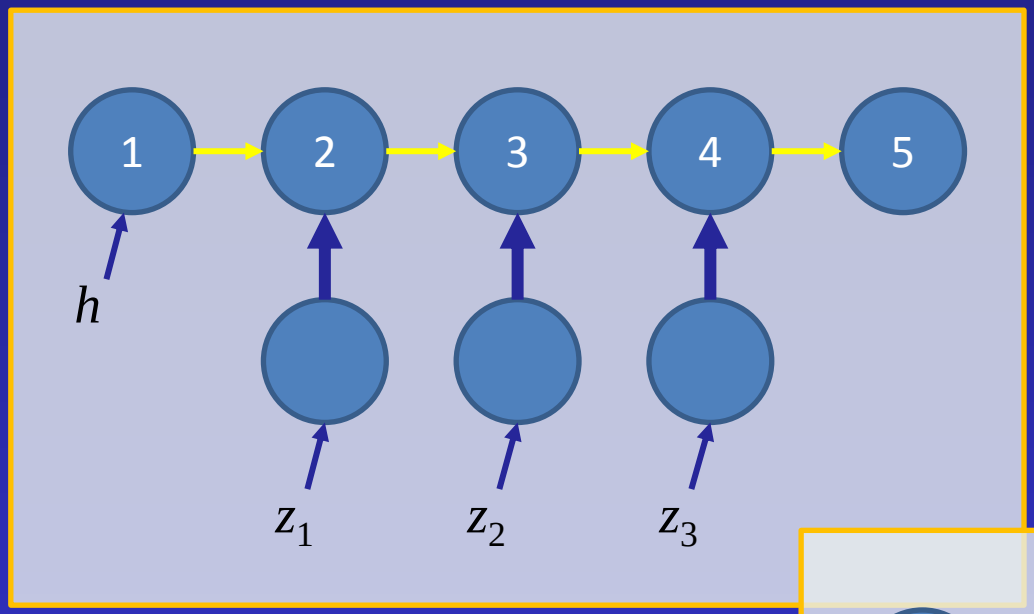
modifies mod = [h, null) $\cup$ [d, d]

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

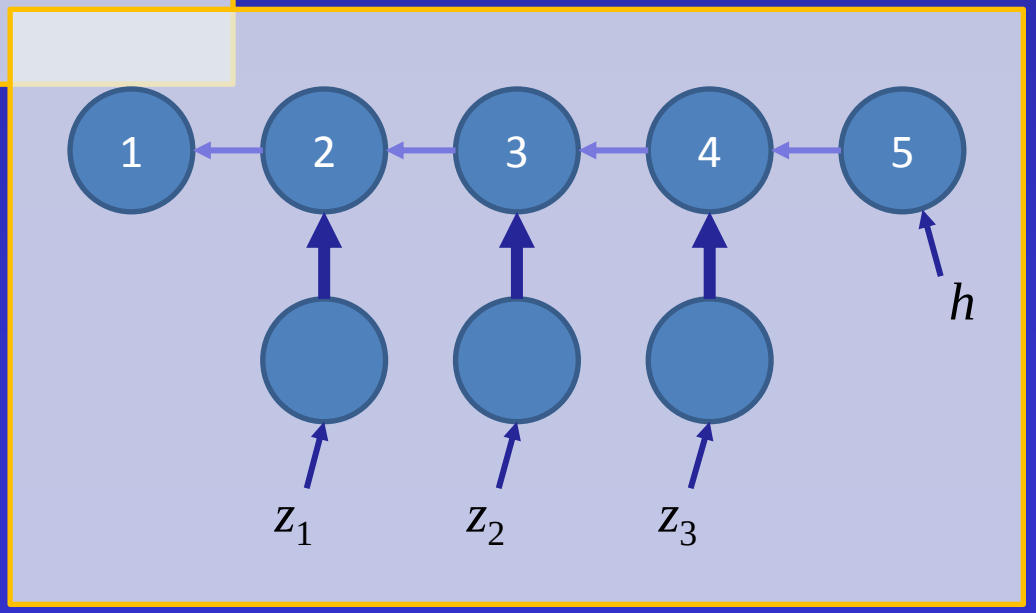
```
Node reverse(Node h, Node d) {
  if (h == null) return d;
  else {
    Node t = h.next;
    h.next = d;
    return reverse(t, h);
  }
}
```

$\{\forall \alpha, \beta \in mod : \alpha<n^*>\beta \leftrightarrow (\beta<\underline{n}^*>\alpha \vee \beta=d)\}$

Large Footprint Reverse

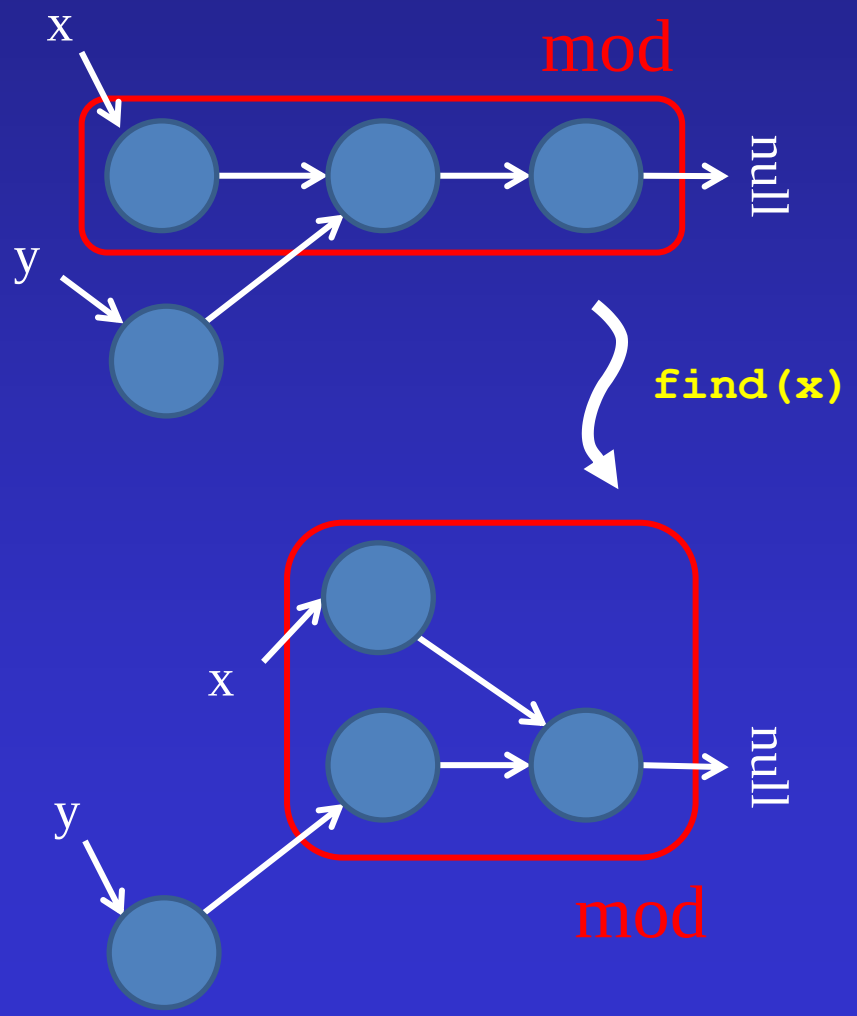


reverse(*h*, null)

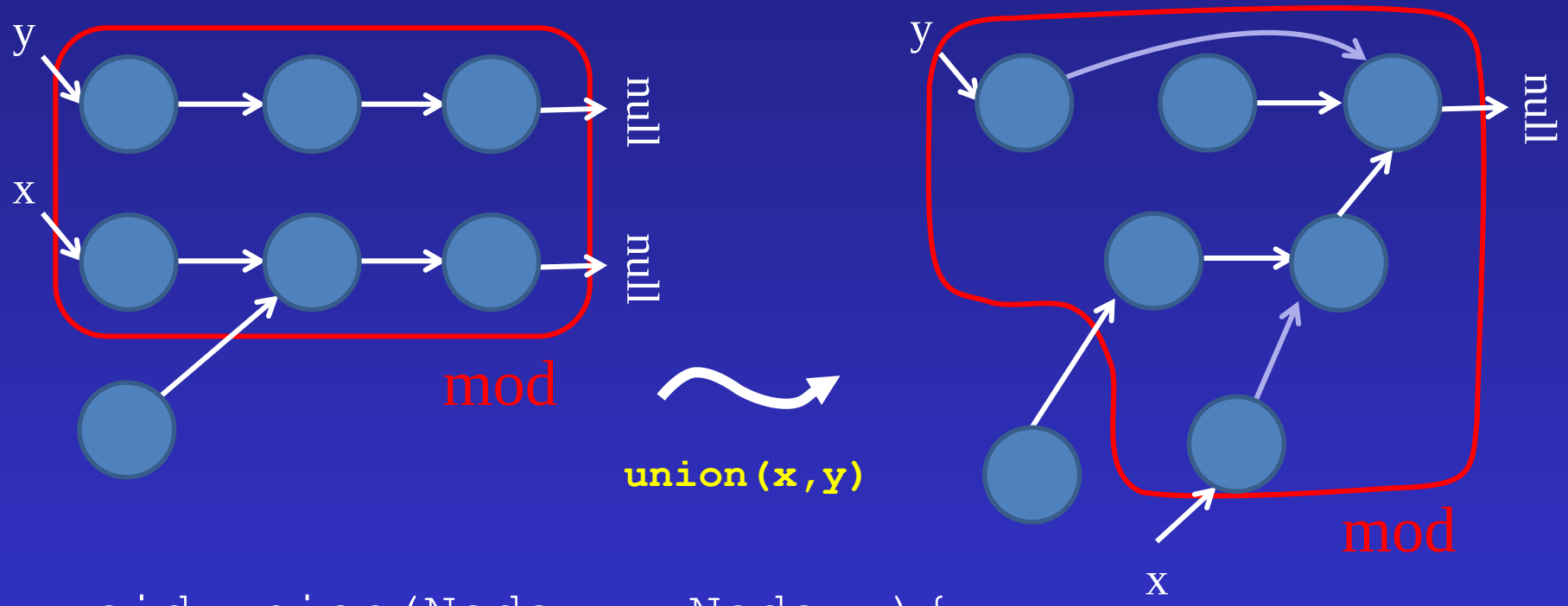


Modular Reasoning

```
Node find(Node x) {
   Node i = x.p;
   if (i != null) {
      i = find(i);
      x.p = i;
   }
   else i = x;
   return i;
}
```

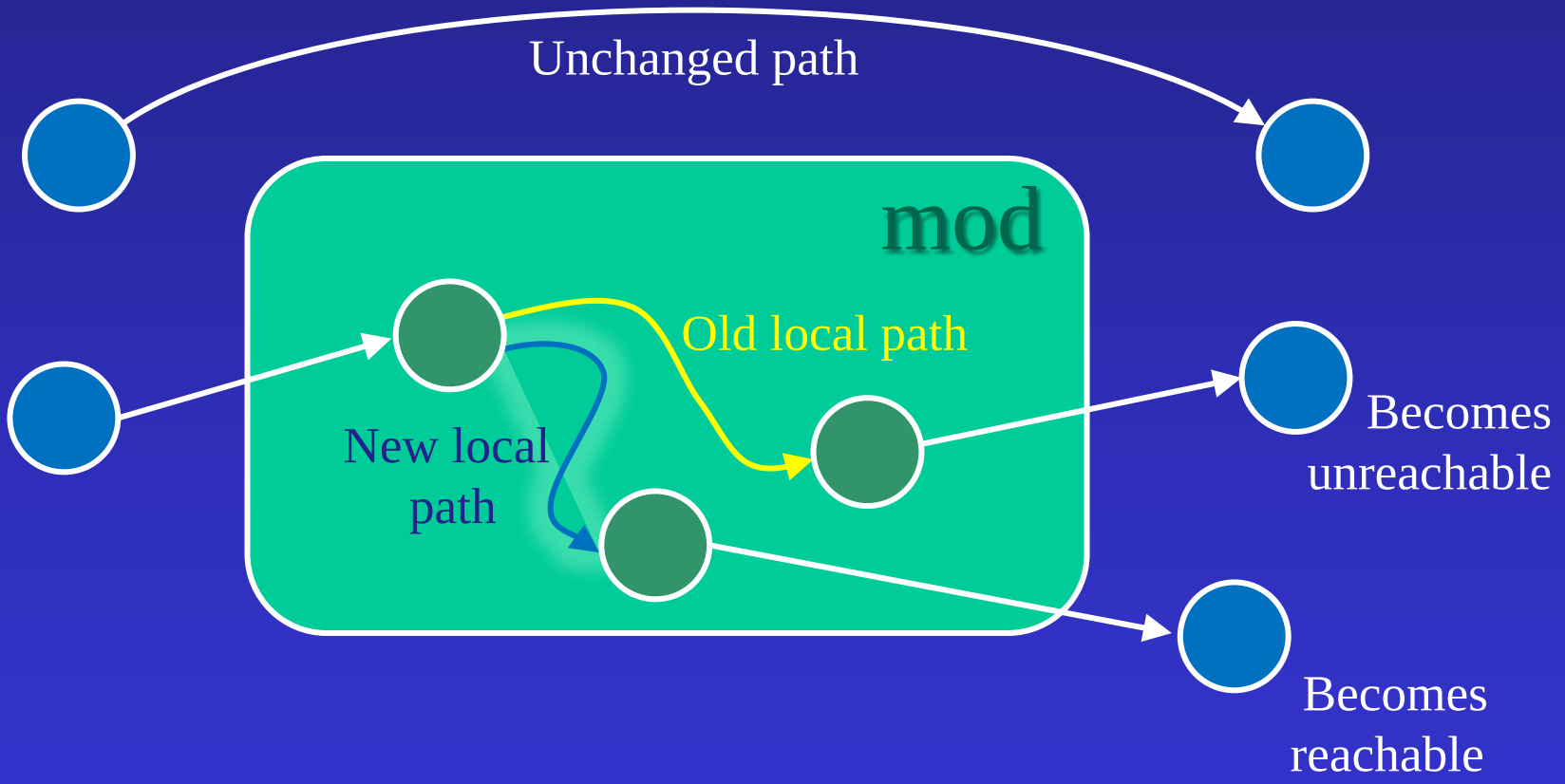


Modular Reasoning




```
void union(Node x, Node y){
  Node t = find(x);
  Node s = find(y);
  if (t != s) t.p = s;
}
```

An Adaptation Rule



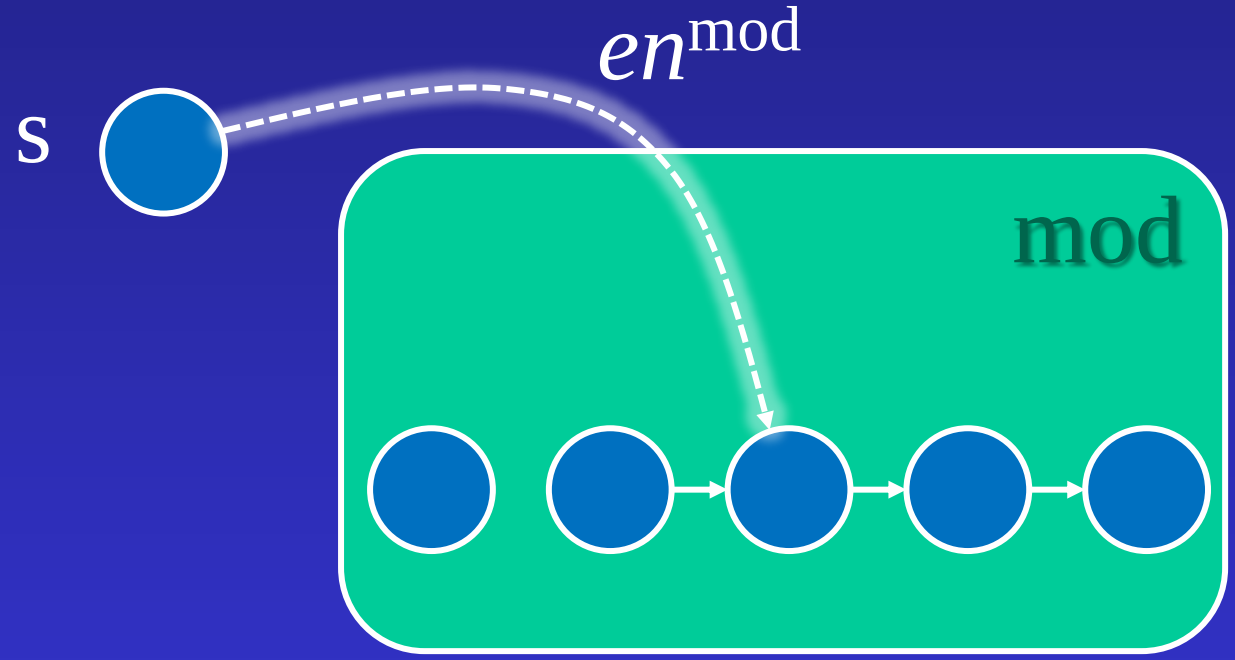
An FOTC Adaptation Rule

- Unmodified edges q
  - $\forall s,t: s\langle q\rangle t \leftrightarrow s\langle \underline{f}\rangle t \wedge s \notin mod \wedge t \notin mod$
- Paths in post state
  - $\forall s, t: s\langle f^*\rangle t \leftrightarrow s\langle q^*\rangle t \vee$
    $\exists \alpha,\beta \in mod : s\langle q^*\rangle \alpha \wedge \alpha\langle f^*\rangle \beta \wedge \beta\langle q^*\rangle t$

An Adaptable Heap Logic

- Extend AF^R with an idempotent function en^{mod}
- Maps nodes into footprint entrance points
- Reducible to effectively propositional
  - Even with nested recursive calls

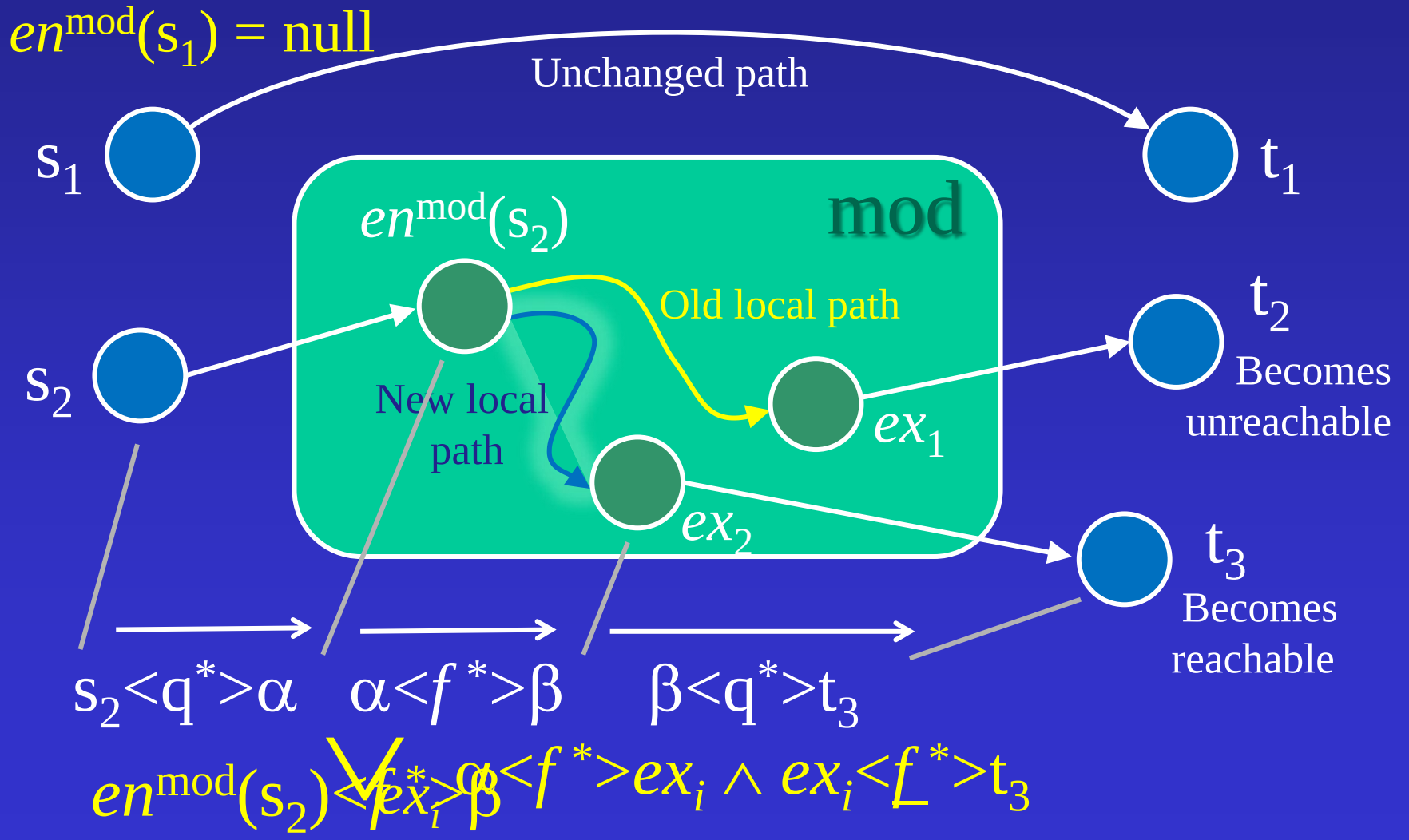
Large Footprint Adaptation



$$en^{\text{mod}}(\text{s}) = \min\left([\text{s},\text{null}) \cap \text{mod}\right)$$

$$\forall \text{s} \notin \text{mod},\ \text{t} \in \text{mod}: \text{s}\langle f^* \rangle \text{t} \leftrightarrow \text{en}^{\text{mod}}(\text{s})\langle f^* \rangle \text{t}$$

An Adaptation Rule (Simplified)



An AE^R Adaptation Rule

$$\forall s,t: s\langle f^* \rangle t \leftrightarrow$$

$$\left(\begin{array}{l} en^{mod}(s) = null \wedge s\langle f^* \rangle t \\ \qquad \vee \\ t \in mod \wedge en^{mod}(s)\langle f^* \rangle t \\ \qquad \vee \\ \bigvee_{ex_i} en^{mod}(s)\langle f^* \rangle ex_i \wedge ex_i\langle f^* \rangle t \end{array} \right)$$

$\forall^* \exists^*$

Small Footprint Specification in AF^R

```
Node find(Node x)
```

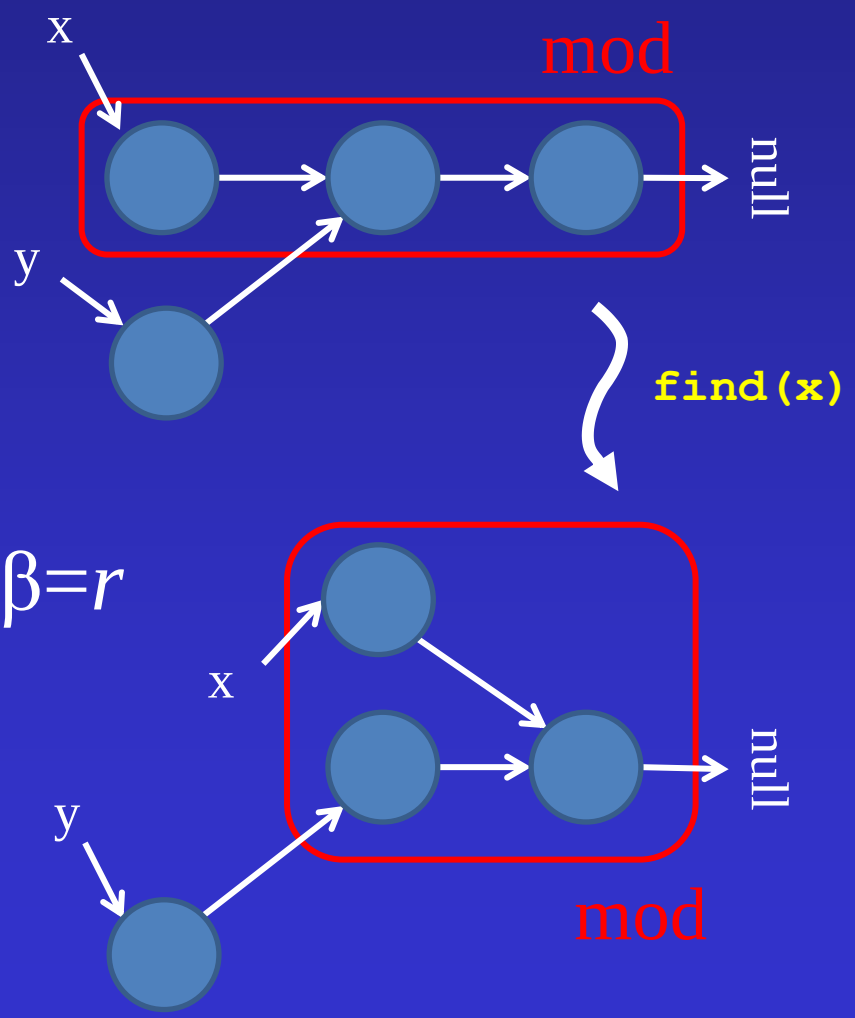
requires: x != null

ensures:

$\forall \alpha,\beta: \alpha<p^*>\beta \leftrightarrow \alpha=\beta \vee \beta=r$

where

$r = \max_p[x, \text{null})$



Verification Time (Z3)

Benchmark	*Formula Size*					*Solving time* (Z3)
	P,Q		mod	VC		
	#	∀	#	#	∀	
SLL: filter	7	2	1	217	6	0.48s
SLL: quicksort	25	2	1	745	9	1.06s
SLL: insert-sort	21	2	1	284	11	0.37s
UF:find	13	2	1	203	6	0.40s
UF:union	20	2	2	188	6	1.39s

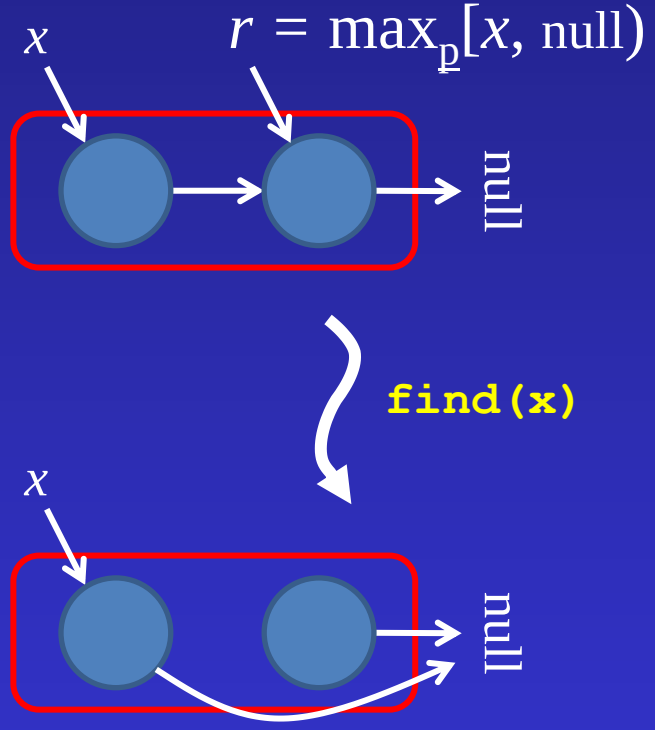
Disproving with SAT

Benchmark	*Nature of defect*	*Formula Size*					*Solving time* (Z3)	*C.e. Size* (vertices)
		P,Q		mod	VC			
		#	∀		#	∀		
UF: find	Incorrect handling of corner case	27	3	2	201	6	1.6s	2
UF: union	Incorrect specification	19	2	2	186	6	0.7s	8
SLL: filter	Uncontrolled sharing	36	4	1	317	6	0.49s	14
SLL: insert-sort	Violating call precondition	21	2	1	283	9	0.88s	8

Example Bug

```
Node find(Node x) {
  Node i = x.p;
  if (i != null) {
    i = find(i);
    x.p = i;
  }
  // else i = x;
  return i;
}
```

Bug: missing `else` branch



violates spec

$\forall \alpha: x\langle \underline{p}^* \rangle \alpha \rightarrow \alpha \langle p^* \rangle r$

(for α=x)

Data Structures outside AF^R

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

Mutations outside our adaptable logic

- Creation of unbounded sharing
- Changing multiple fields
  - Nested linked lists

Property Guided Shape Analysis
N. Bjorner, T. Reps, A.Thakur, T. Weiss

- Predictable Shape Analysis
  - Simple fixed Predicate Abstraction
  - Infer propositional invariants guided by the verification problem
  - When the analysis fails:
    - Concrete counterexamples
    - A trace showing overly coarse abstraction
  - Programmer can define new predicates using AF^R
- Employ IC3/PDR

Related Work

- First Order Rechability Axioms
  - [Nelson POPL'83] Useful axioms
  - [Lev-Ami'09] Useful axioms + completeness study
- Incremental Methods [Hesse'03, Reps'03, Lahiri&Qadeer POPL'06]
- Decidable Logics [Mona, STRAND, LRP, Berdine'2004,Lahiri&Qadeer POPL'08, Wies, Muñiz,Kuncak'2011,12 …]

Summary

- Reduction to SAT
- Support modularity
- Works for many programs
- Principles
  - Restricted invariants
  - Inversion n*
  - Uniform mutations
  - Two logics