Static Program Analysis

Lecture 20: Pointer & Shape Analysis II

Summer Semester 2018

Thomas Noll
Software Modeling and Verification Group
RWTH Aachen University

https://moves.rwth-aachen.de/teaching/ss-18/spa/

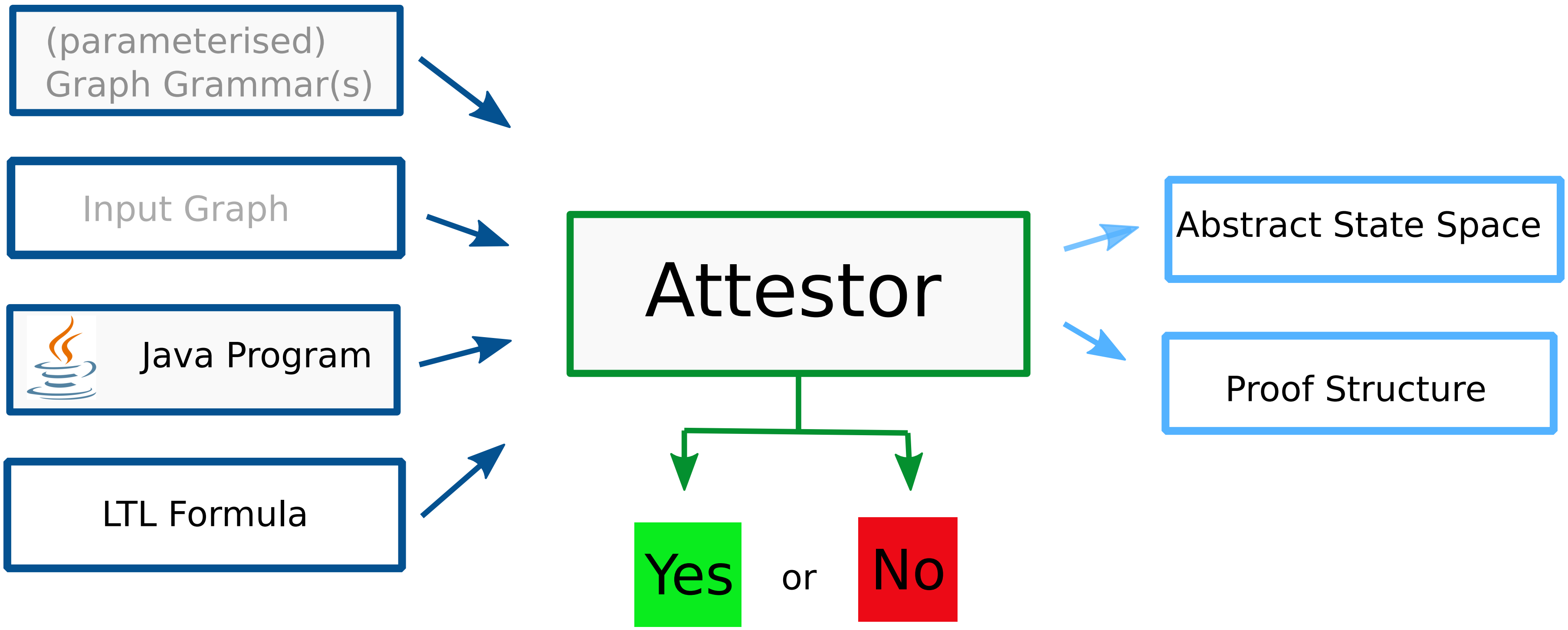
Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

The Shape Analysis Approach

- **Goal:** determine the possible shapes of a dynamically allocated data structure at given program point
- **Interesting information:**
  - data types (to avoid type errors, such as dereferencing `null`)
  - aliasing (different pointer variables having same value)
  - sharing (different heap pointers referencing same location)
  - reachability of nodes (garbage collection)
  - disjointness of heap regions (parallelisability)
  - shapes (lists, trees, absence of cycles, ...)
- **Concrete questions:**
  - Does `x.next` point to a shared element?
  - Does a variable `p` point to an allocated element every time `p` is dereferenced?
  - Does a variable point to the head of an acyclic list?
  - Does a variable point to the root of a tree?
  - Can a loop or procedure cause a memory leak?

Recap: Shape Analysis using Hyperedge Replacement Grammars

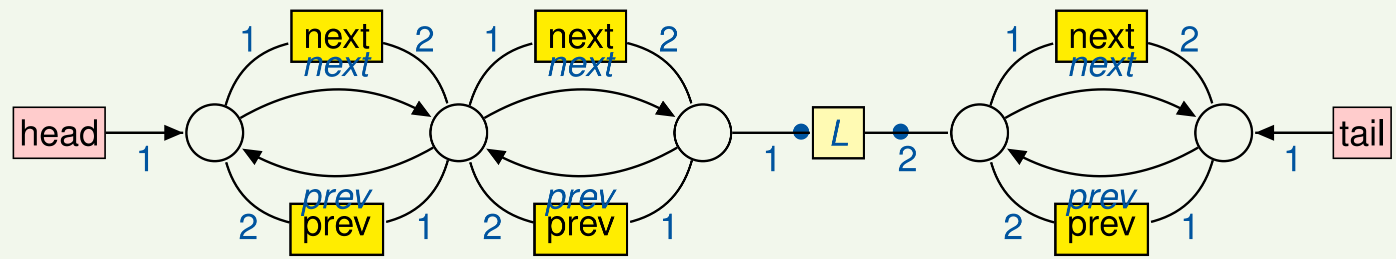
The Attestor[1] Approach



[1] https://github.com/moves-rwth/attestor

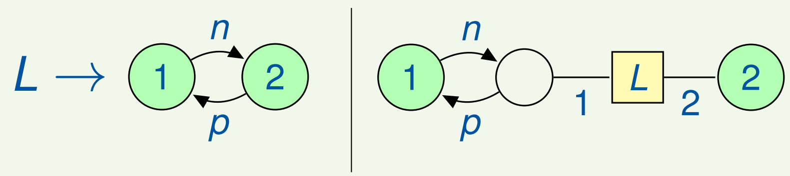
Data Abstraction

Heap representation: hypergraph

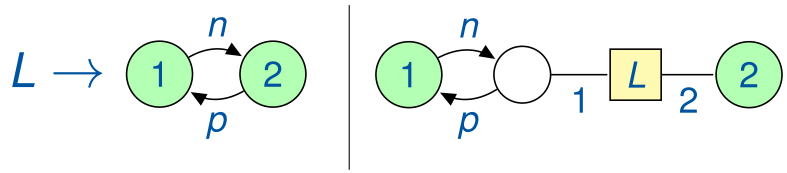


- Placeholders: nonterminal (labelled) hyperedges of rank n
- Pointers: terminal (labelled) hyperedges of rank 2
- Variables: hyperedges of rank 1

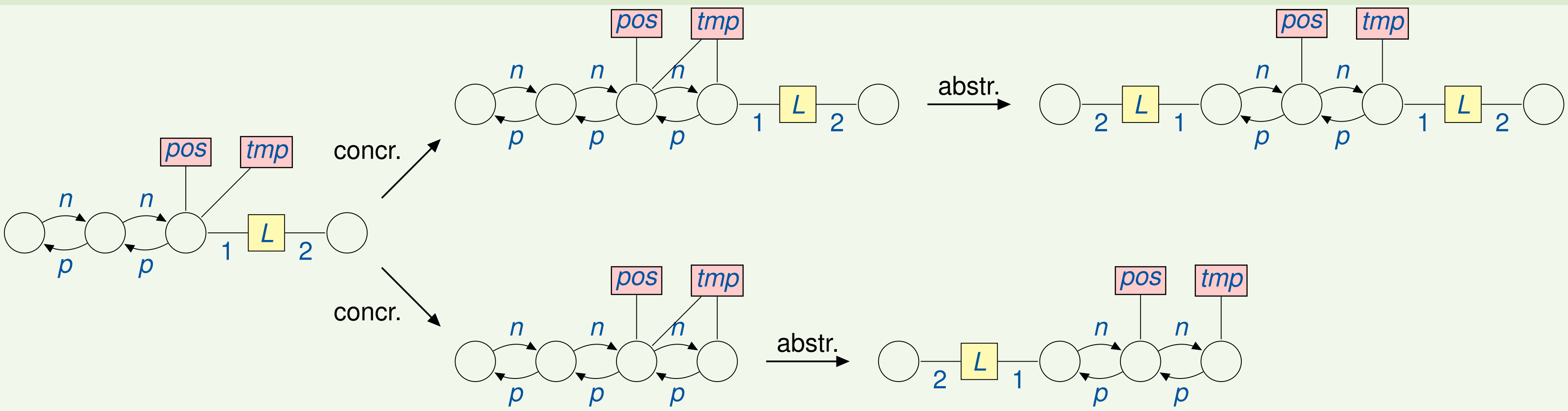
Specification of placeholder(s): Hyperedge Replacement Grammar (HRG)



Abstract Execution




```
tmp := pos.next;
```



Principle

Concretise whenever necessary; abstract whenever possible.

Galois Connections

- Concrete domain L, ordered by $\sqsubseteq_L$ (concrete heaps)
- Abstract domain M, ordered by $\sqsubseteq_M$ (heaps with placeholders)
- Concretisation function $\gamma : M \to L$ (forward derivation)
- Abstraction function $\alpha : L \to M$ (backward derivation)

Reminder: Galois connection (cf. Definition 10.1)

Let $(M, \sqsubseteq_M)$, $(L, \sqsubseteq_L)$ be complete lattices with monotonic functions $\alpha : L \to M$, $\gamma : M \to L$. $L \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} M$ is a Galois connection iff

$$\forall l \in L. l \sqsubseteq_L \gamma(\alpha(l)) \qquad \text{(overapproximation)}$$

and

$$\forall m \in M. \alpha(\gamma(m)) \sqsubseteq_M m \qquad \text{(preservation of precision).}$$

Galois Connection for Pointer Programs

- $HC/HC^\#$ concrete/abstract heap configurations (without/possibly with nonterminals)
- HRG G with derivation relation $\Rightarrow_G \subseteq 2^{HC^\#} \times 2^{HC^\#}$
- Concrete domain: 2^{HC}
  - partially ordered by $\subseteq$
  - concretisation function $\gamma_G(\{H^\#\}) := L_G(H^\#) = \{H \in HC \mid H^\# \Rightarrow_G^* H\}$
- Abstract domain: $2^{HC^\#}$
  - partially ordered by $\sqsubseteq$ with $m_1 \sqsubseteq m_2$ iff $\gamma_G(m_1) \subseteq \gamma_G(m_2)$
  - abstraction function $\alpha_G(\{H\}) := \{H^\# \mid H^\# \Rightarrow_G^* H, \nexists K^\# : K^\# \Rightarrow_G H^\#\}$ (maximal abstraction)

Additional requirements on G

- Data Structure Normal Form (DSNF): ensures that γ_G/α_G yield valid heap configurations
- Backward confluence: for all H, $|\alpha_G(\{H\})| = 1$ (uniqueness of abstraction)

Theorem 20.1

If G is a backward confluent HRG in DSNF, then $2^{HC} \underset{\gamma_G}{\overset{\alpha_G}{\rightleftarrows}} 2^{HC^\#}$ forms a Galois connection.

Soundness of Abstract Interpretation

- Concrete semantics $f : 2^{HC} \to 2^{HC}$ (pointer operation)
- Abstract semantics $f^{\#} : 2^{HC^{\#}} \to 2^{HC^{\#}}$ (1. concretisation, 2. f, 3. abstraction)

Reminder: Safe approximation of functions (cf. Definition 11.1)

Abstract		**Concrete**
m	$\xrightarrow{\gamma}$	$\gamma(m)$
$\downarrow f^{\#}$		$\downarrow f$
$f^{\#}(m) \sqsupseteq_M \alpha(f(\gamma(m)))$	$\xleftarrow{\alpha}$	$f(\gamma(m))$

Abstract Execution of Pointer Programs

Wanted: most precise safe approximation

For all $f : HC \to HC$ and $H^\# \in HC^\#$,

$$f^\#(H^\#) = \alpha_G(f(\gamma_G(H^\#)))$$

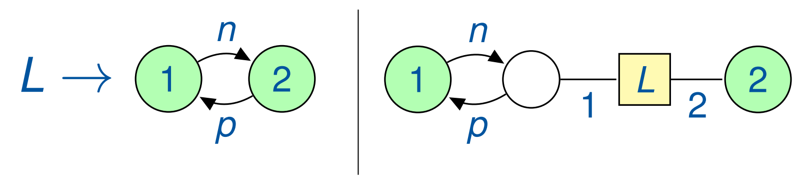
Problem

$\gamma_G(H^\#)$ generally infinite (or too large)

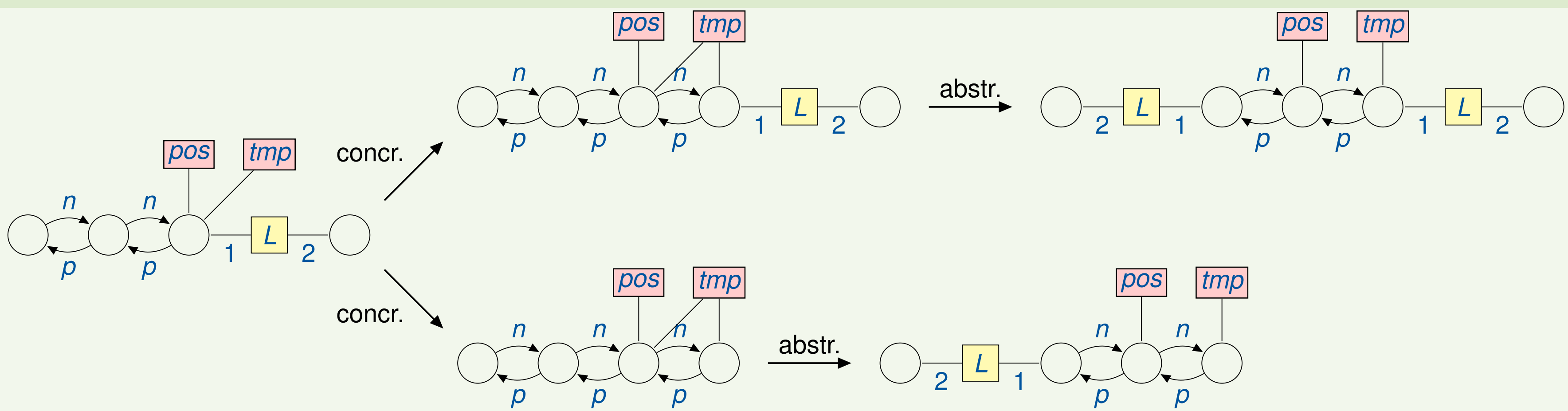
Solution

Stepwise local concretisation (only “as much as necessary”)

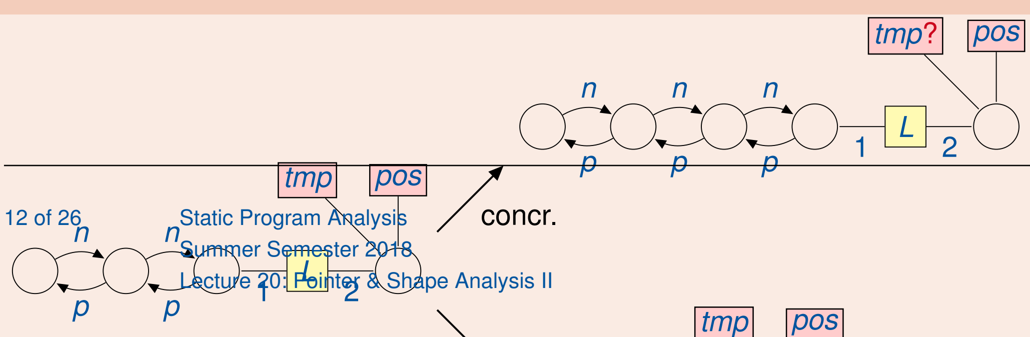
Local Concretisability



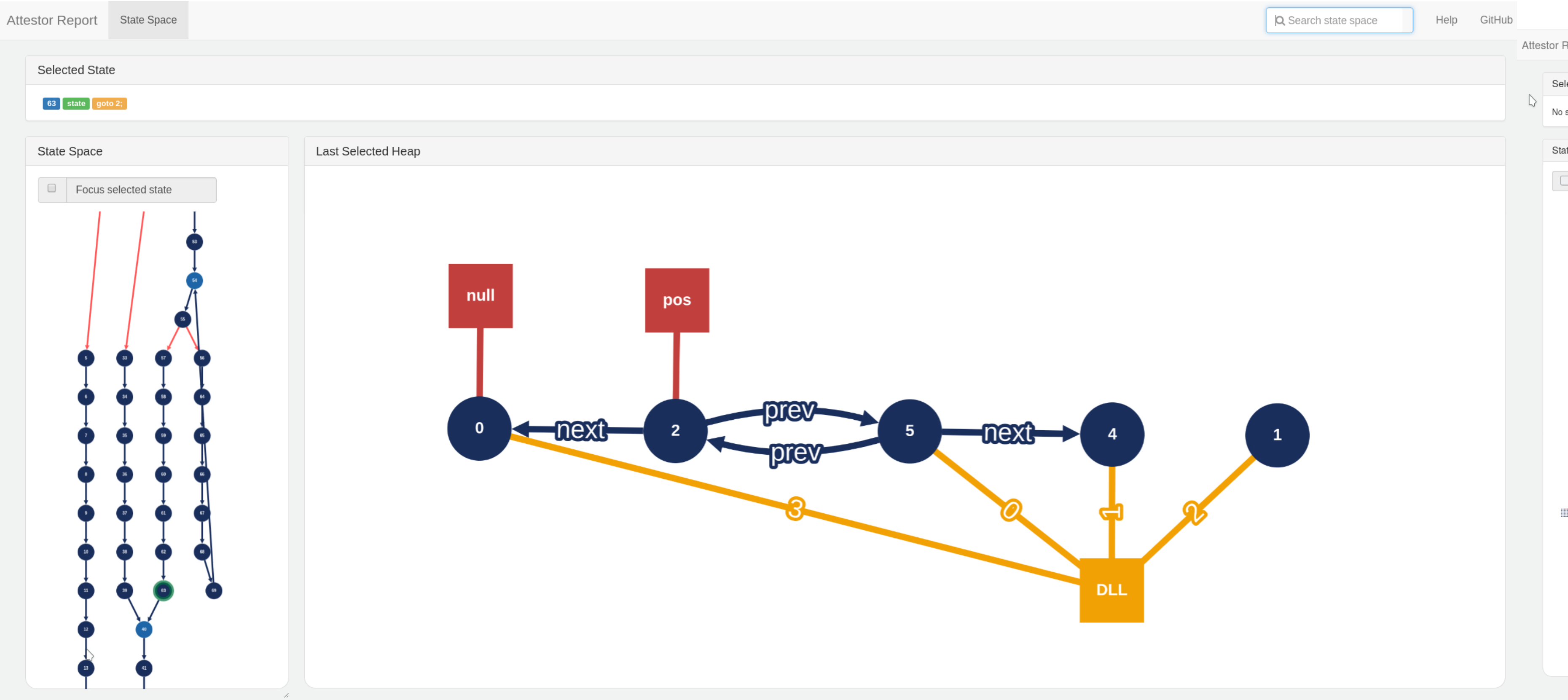
`tmp := pos.next;`



`tmp := pos.prev;`



Visualisation of State Spaces in Attestor[2]



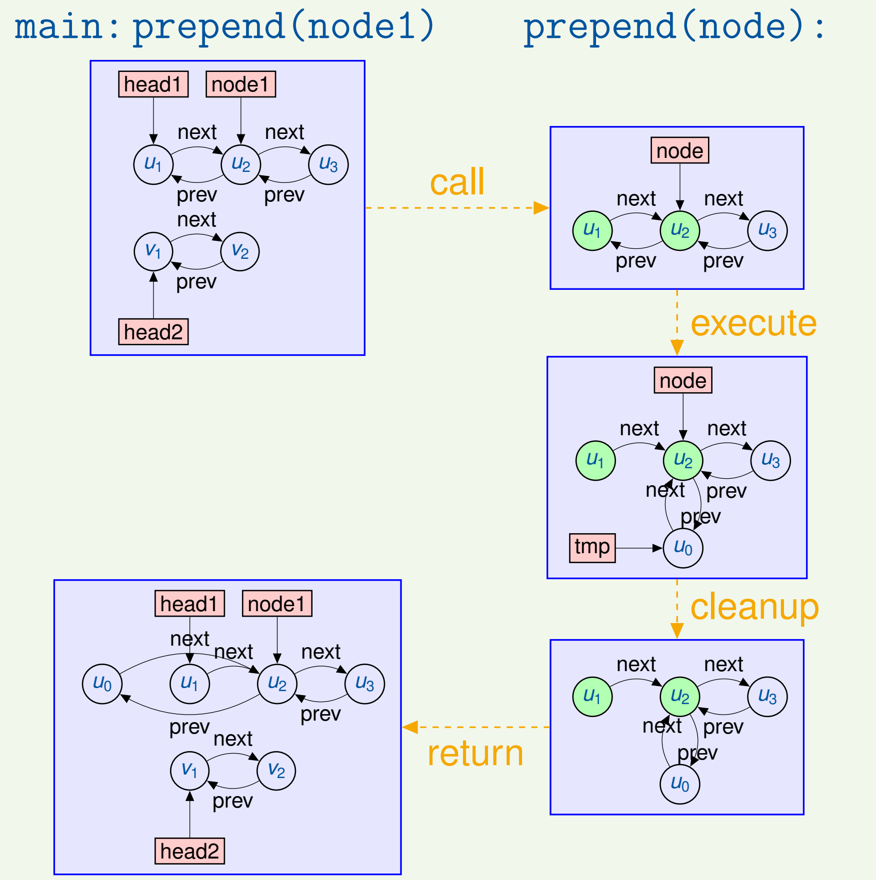
[2] https://github.com/moves-rwth/attestor

Handling of Procedure Calls

Analysing procedure calls

- At call:
  1. truncate to reachable fragment and identify cutpoints (i.e., nodes referenced by local variables of caller)
  2. rename actual $\mapsto$ formal parameters
  3. apply (intraprocedural) semantics of body
- On return:
  1. discard local variables
  2. merge heap at call site with procedure result
- Yields (part of) procedure summary

Example 20.2 (List prepend)



Modularity via Procedure Summaries

Goal

- Determine abstract graph-based procedure summaries (“contracts”)
- Summary = set of (precondition, postcondition)
  - precondition = abstract reachable heap fragment upon call
  - postcondition = set of possible resulting abstract heaps
- Demand-driven computation (only consider preconditions that actually occur in symbolic execution)

Algorithm: interprocedural data-flow analysis

1. Compute program’s control flow graph
2. Set up data-flow equations for each basic block:
   - collect summary information of predecessor blocks
   - apply abstract semantics of present block to update postcondition
3. Solve equation system via fixed-point iteration

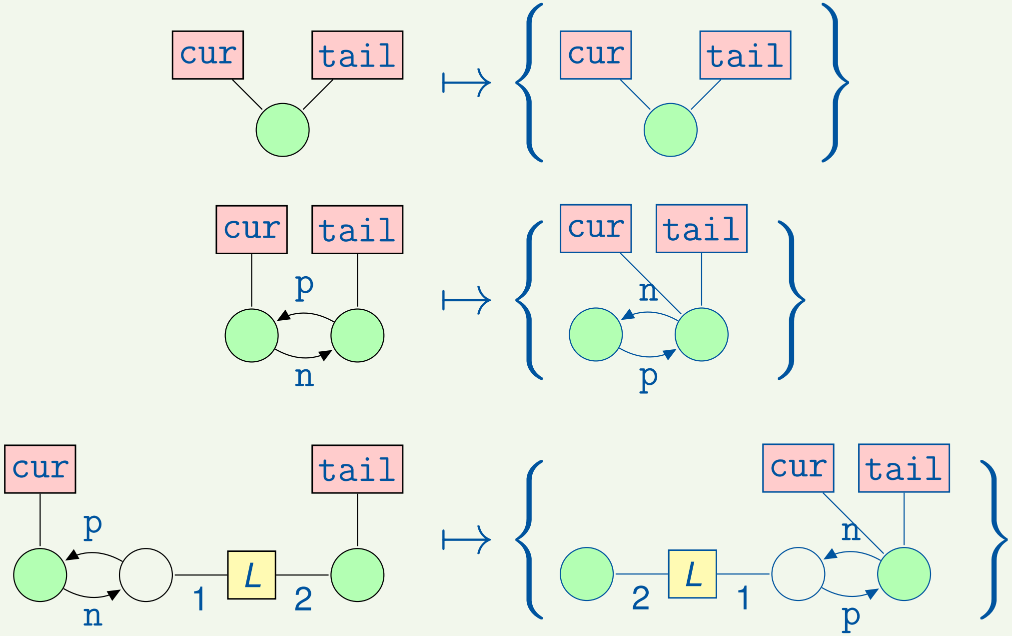
An Example

Example 20.3 (List reversal)

```
main(head, tail: elem){
  var tmp: elem;
  reverse(head, tail);
  tmp := head;
  head := tail;
  tail := tmp;
}

reverse(cur, tail: elem){
  var tmp: elem;
  if (cur != tail){
    tmp := cur.prev;
    cur.prev := cur.next;
    cur.next := tmp;
    reverse(cur.prev, tail);
  }
}
```

`reverse` summary (excerpt):



Adding Threads with fork/join Concurrency

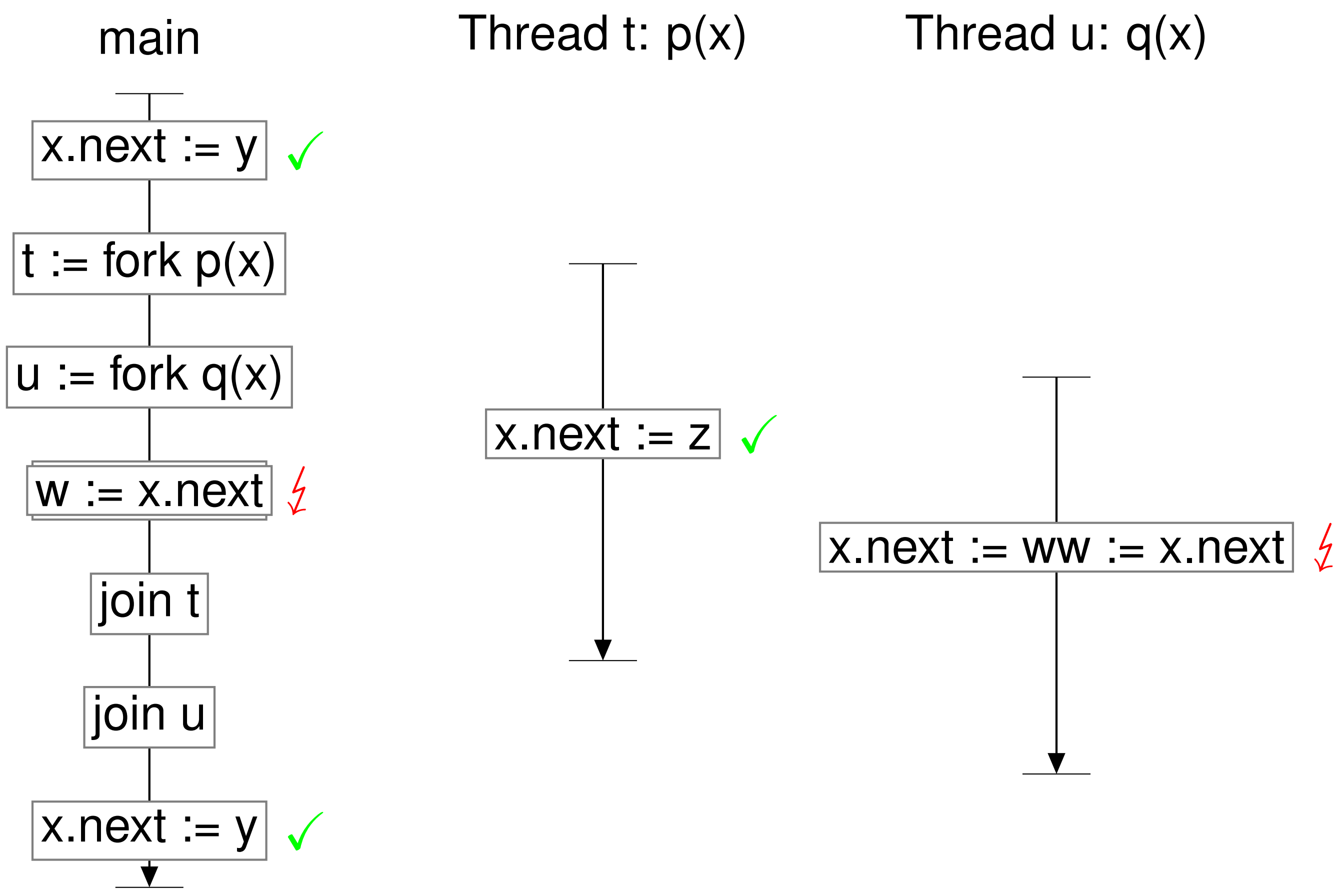
Example 20.4 (Concurrent list copy)

```
main(head: elem, tail: elem){
  thread t1, t2;
  var head1, head2: elem;
  head1 := new(elem);
  head2 := new(elem);
  t1 := fork copy(head, tail, head1);
  t2 := fork copy(head, tail, head2);
  join t1;
  join t2;
}
type elem{
  prev: elem;
  next: elem
}
```

```
copy(cur, tail, cur1: elem){
  var tmp, tmp1: elem;
  tmp := cur.next;
  tmp1 := new(elem);
  cur1.next := tmp1;
  tmp1.prev := cur1;
  if (tmp != tail){
    copy(tmp, tail, tmp1);
  }
}
```

Adding Permissions for Concurrency

Data Races



Adding Permissions for Concurrency

Access Permissions

Idea

- Threads acquire/release read and write permissions
- Read permission for shared read access
- Write permissions for exclusive write access

Observations

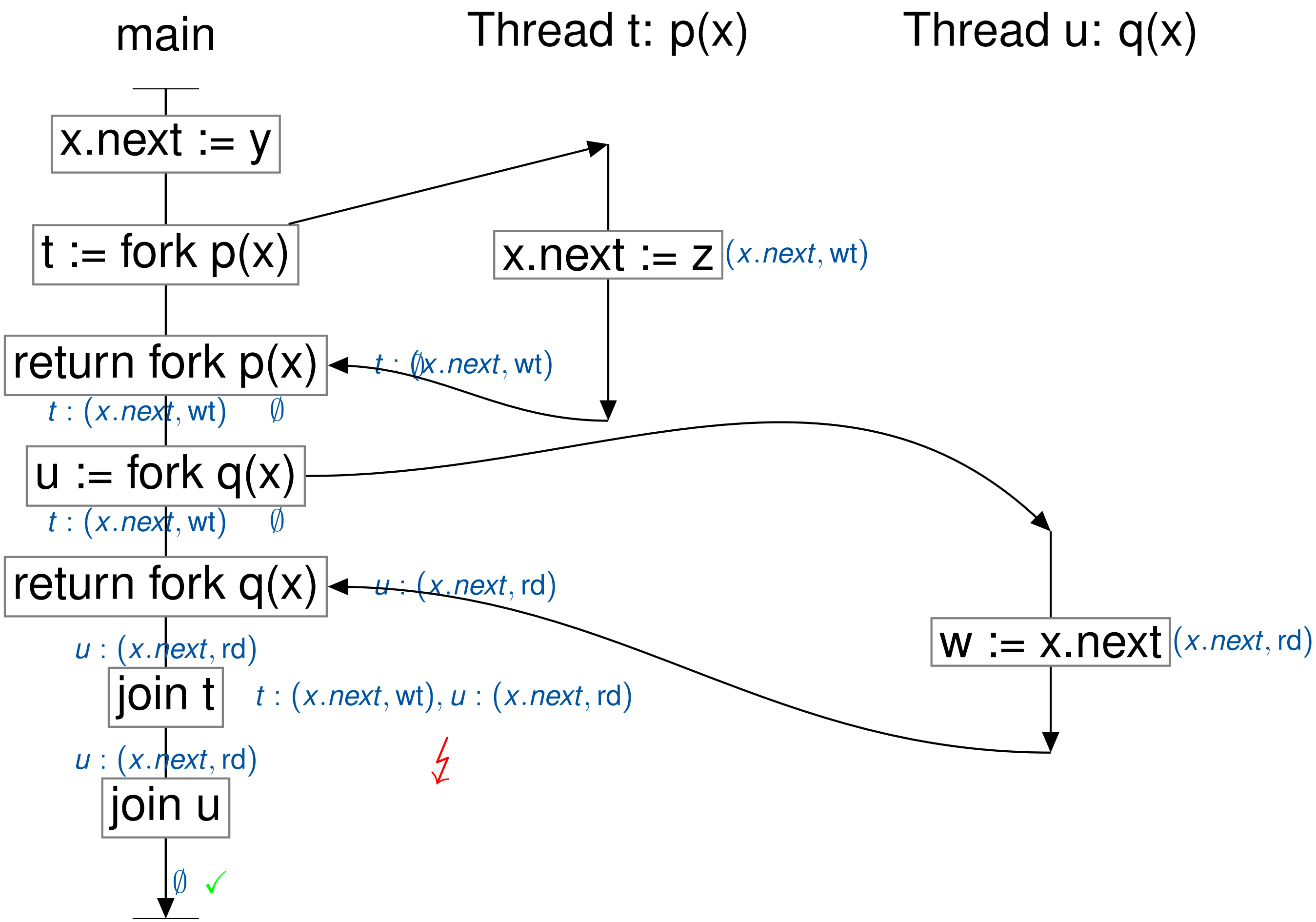
- Permission not available $\Longrightarrow$ potential data race
- Permissions can always be acquired $\Longrightarrow$ data-race freedom

Goal

- Automatically distribute permissions
- Static analysis: no runtime representation!

Adding Permissions for Concurrency

Ensuring Data Race Freedom



Software Modeling and Verification Chair

RWTH AACHEN UNIVERSITY

Adding Permissions for Concurrency

Analysing Concurrent Pointer Programs

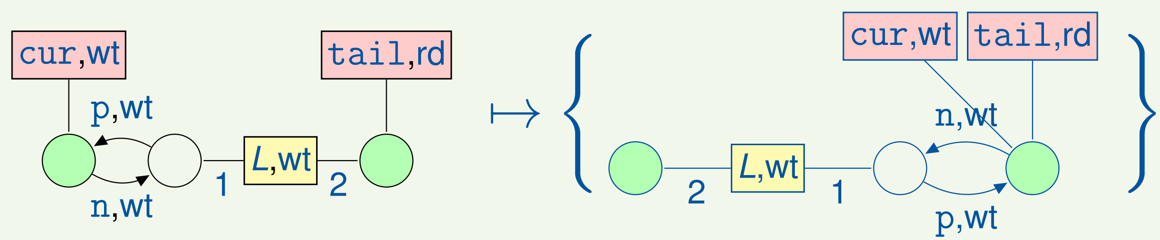
Observation

Data race freedom $\Longrightarrow$ deterministic results $\Longrightarrow$ consider only one interleaving

Algorithm

1. Treat forks just like procedure calls
2. Allocate permissions greedily
3. Keep track of permissions until join
4. Report permission error when conflicts detected
5. Fork without subsequent join $\Longrightarrow$ lost permissions to be remembered

Example 20.5 (Part of thread contract for list reversal)



Experimental Results & Literature

Experimental Results

Program	Property	Rules	States	Time
ReverseList	(1, 2)	3	192	0.23 s
ReverseList	(3)	3	5,615	0.447 s
ReverseList	(4)	3	5,107	0.399 s
TreeFlatten	(1, 2)	14	2,887	0.622 s
TreeFlatten	(3)	14	77,373	1.446 s
TreeFlatten	(4)	14	423,525	5.61 s
Lindstrom	(1, 2)	12	4,520	0.506 s
Lindstrom	(3)	12	160,855	1.537 s
Lindstrom	(4)	12	983,680	6.536 s
AVL rotate	(1, 2, 5)	16	190	0.192 s
AVL search	(1, 2, 5)	16	216	0.172 s
AVL insert	(1, 2, 5)	16	15,202	11.032 s
BiMap search	(1, 2, 6)	4	266	0.160 s
BiMap insert	(1, 2, 6)	4	128	0.144 s
BiMap search	(1, 2, 6)	4	274	0.159 s

Properties

1. Pointer safety
2. Structure preservation
3. "Bag" property (for lists):

$$\forall x : \mathit{head} \rightarrow^* x \implies \Diamond\Box\, \mathit{tail} \rightarrow^* x$$

4. Correctness (for list reversal):

$$\forall x, y : \mathit{head} \rightarrow^* x \wedge x \rightarrow y \implies \Diamond\Box\, y \rightarrow x$$

5. Balancedness (with indices)
6. Equal length (with indices)

Experimental Results & Literature

Literature on Attestor

(available from Attestor web page[3])

Gentle introduction: J. Heinen, C. Jansen, J.-P. Katoen, T. Noll: *Verifying Pointer Programs using Graph Grammars*, Sci. Comp. Progr. 97, 157–162, 2015[4]

General framework: J. Heinen, C. Jansen, J.-P. Katoen, T. Noll: *Juggrnaut: Using Graph Grammars for Abstracting Unbounded Heap Structures*, Formal Methods in System Design 47(2), 159–203, 2015[5]

Procedure summaries: C. Jansen, T. Noll: *Generating Abstract Graph-Based Procedure Summaries for Pointer Programs*, ICGT 2014, LNCS 8571, 49–64[6]

Extension to relational properties (balancedness): H. Arndt, C. Jansen, C. Matheja, T. Noll. *Heap Abstraction Beyond Context-Freeness*[7]. SEFM 2018, LNCS 10886, 271–286

[3] https://github.com/moves-rwth/attestor
[4] https://doi.org/10.1016/j.scico.2013.11.012
[5] https://dx.doi.org/10.1007/s10703-015-0236-1
[6] https://dx.doi.org/10.1007/978-3-319-09108-2_4
[7] http://dx.doi.org/10.1007/978-3-319-92970-5_17