Verification of Pointer Programs

Stefan Rieger

MOVES: Software Modeling and Verification
RWTH Aachen University, Germany

23.09.2009

Pointers

Pointers are indispensable and omnipresent

- object-oriented programming
- dynamic memory management and data structures
- data bases and index structures
- ...

Pointers

Pointers are indispensable and omnipresent

- object-oriented programming
- dynamic memory management and data structures
- data bases and index structures
- ...

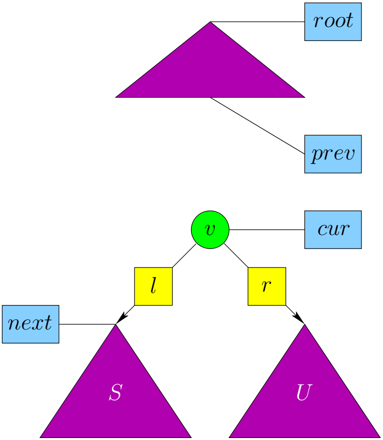
Difficulties

- aliasing creates dependencies
- destructive updates
- dereferencing invalid/null pointers

$\Rightarrow$ automatic verification desirable

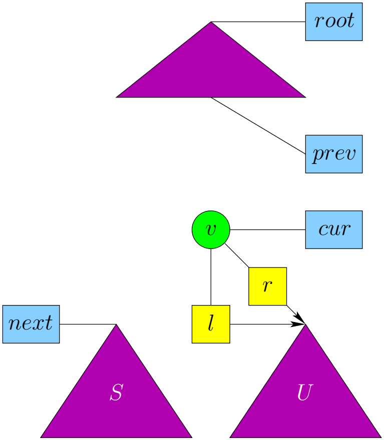
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;     ←
7  cur.r := prev;
8  prev := cur;
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



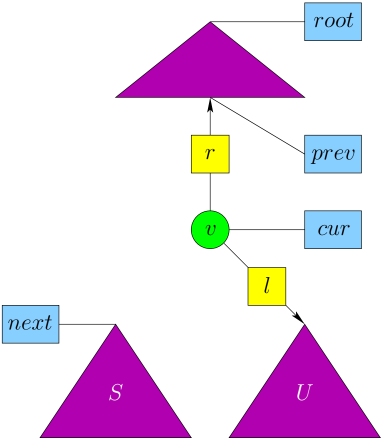
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;
7  cur.r := prev;      ←
8  prev := cur;
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



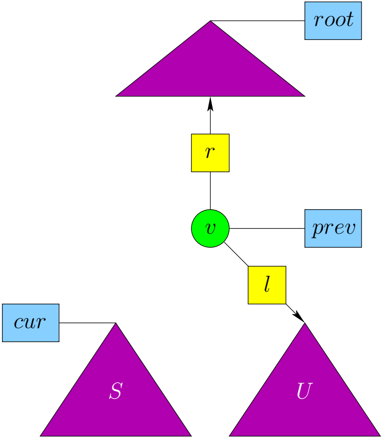
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;
7  cur.r := prev;
8  prev := cur;        ←
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



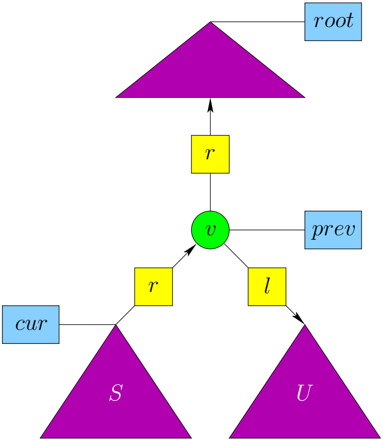
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;      ←
7  cur.r := prev;
8  prev := cur;
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



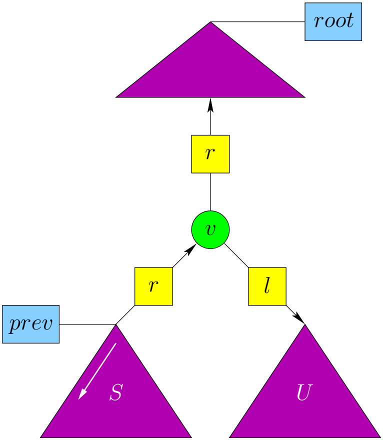
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;
7  cur.r := prev;
8  prev := cur;        ←
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



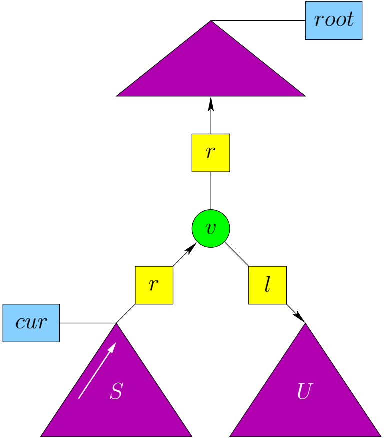
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;     ←
7  cur.r := prev;
8  prev := cur;
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



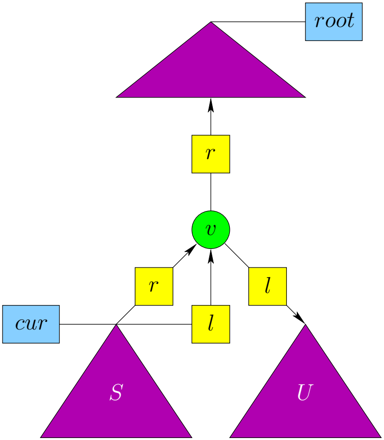
Example: The Deutsch-Schorr-Waite Algorithm

1 **if** root = null **goto** 15;
2 **new**(sen);
3 prev := sen;
4 cur := root;
5 next := cur.l;
6 cur.l := cur.r; ←
7 cur.r := prev;
8 prev := cur;
9 cur := next;
10 **if** (cur = sen) **goto** 15;
11 **if** (cur ≠ null) **goto** 5;
12 cur := prev;
13 prev := null;
14 **goto** 5;



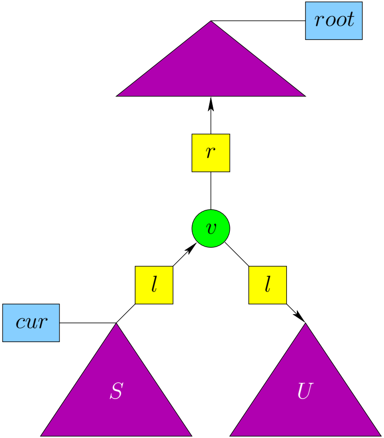
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;
7  cur.r := prev;        ←
8  prev := cur;
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



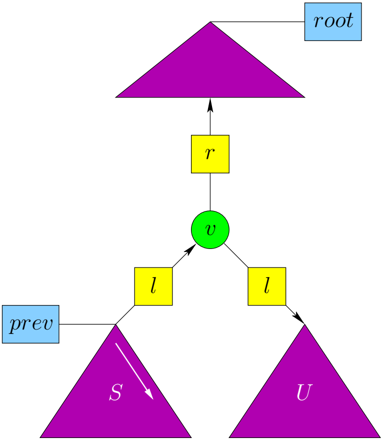
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;
7  cur.r := prev;
8  prev := cur;        ←
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



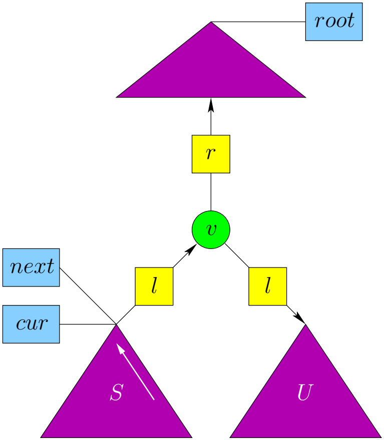
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;      ←
7  cur.r := prev;
8  prev := cur;
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



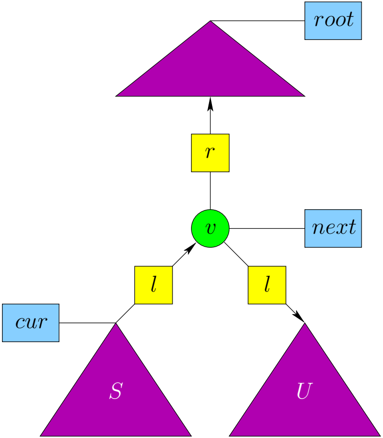
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;        ←
6  cur.l := cur.r;
7  cur.r := prev;
8  prev := cur;
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



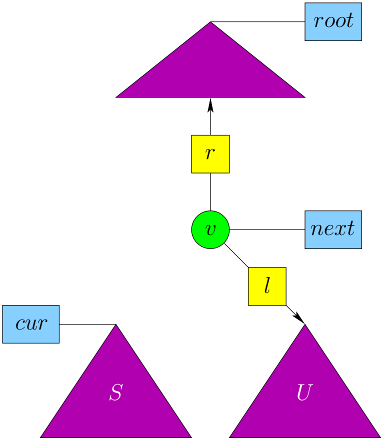
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;      ←
7  cur.r := prev;
8  prev := cur;
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



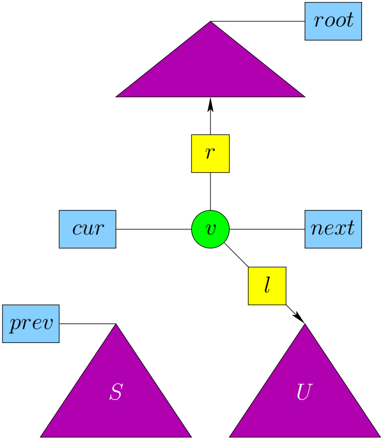
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;
7  cur.r := prev;
8  prev := cur;        ←
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



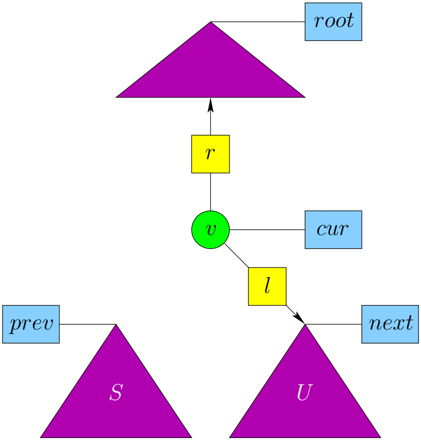
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;        ←
6  cur.l := cur.r;
7  cur.r := prev;
8  prev := cur;
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



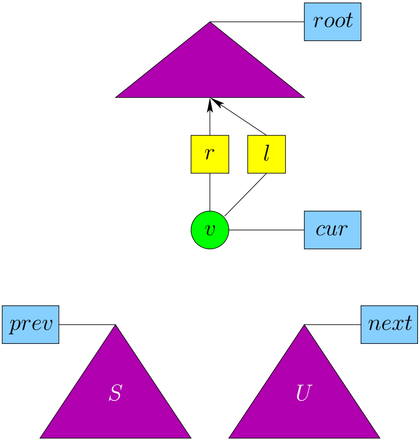
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;      ←
7  cur.r := prev;
8  prev := cur;
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



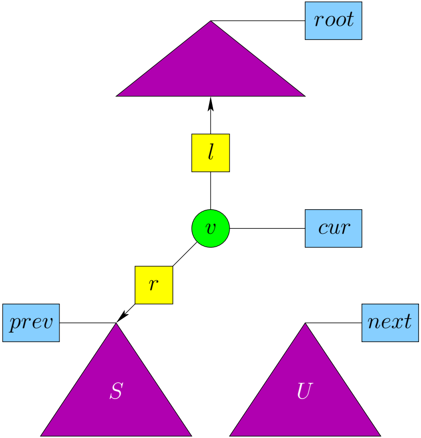
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;
7  cur.r := prev;     ←
8  prev := cur;
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



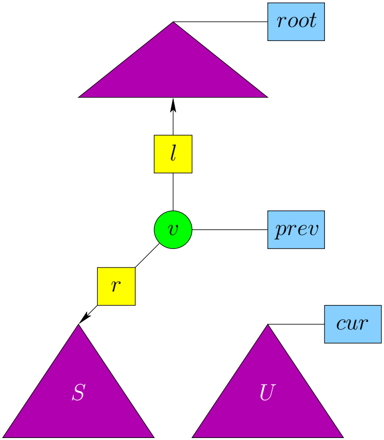
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;
7  cur.r := prev;
8  prev := cur;        ←
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



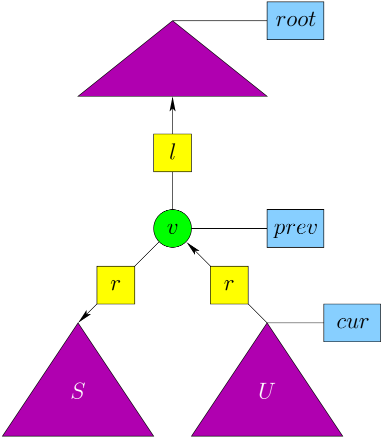
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;      ←
7  cur.r := prev;
8  prev := cur;
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



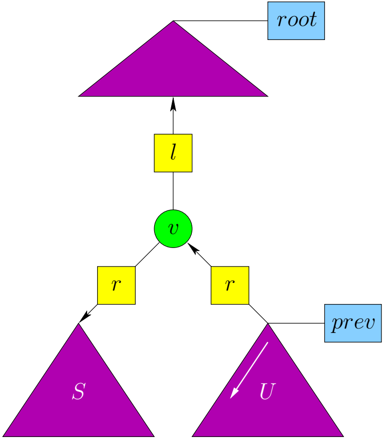
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;
7  cur.r := prev;
8  prev := cur;        ←
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



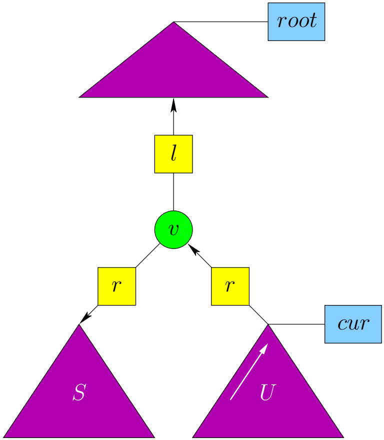
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;   ←
7  cur.r := prev;
8  prev := cur;
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



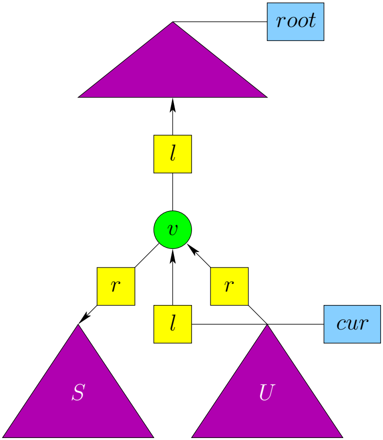
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;     ←
7  cur.r := prev;
8  prev := cur;
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



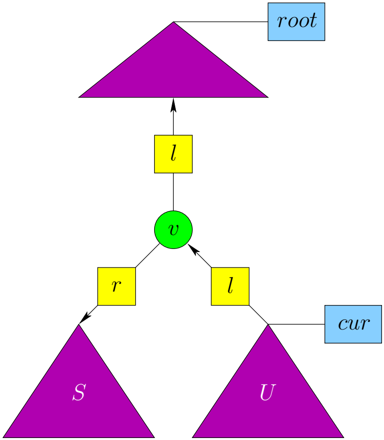
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;
7  cur.r := prev;      ←
8  prev := cur;
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



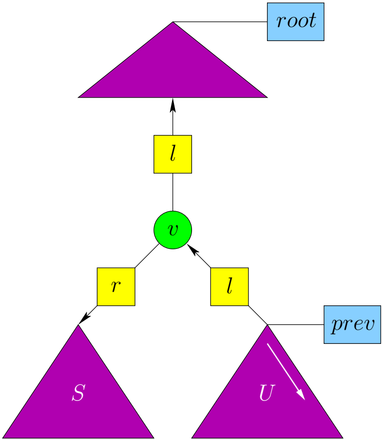
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;
7  cur.r := prev;
8  prev := cur;        ←
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



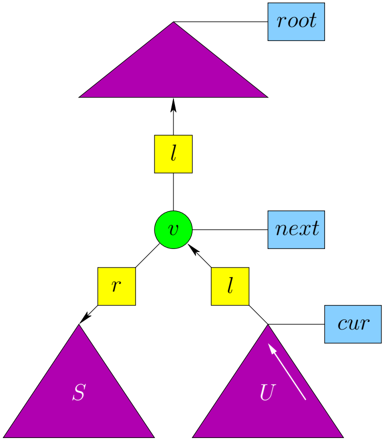
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;      ←
7  cur.r := prev;
8  prev := cur;
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



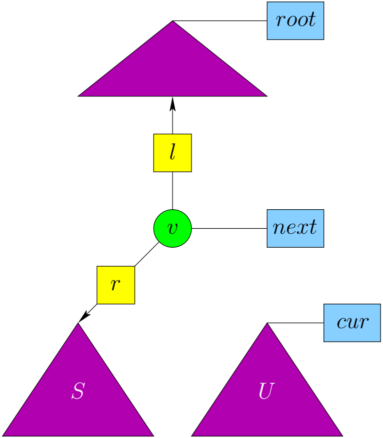
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;     ←
7  cur.r := prev;
8  prev := cur;
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



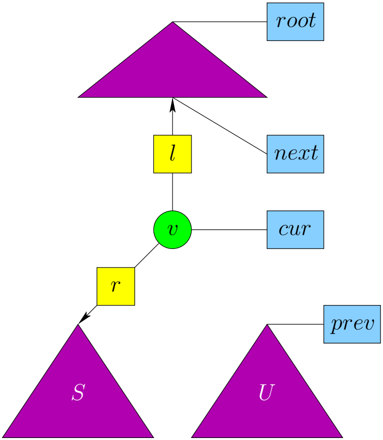
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;
7  cur.r := prev;
8  prev := cur;        ←
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



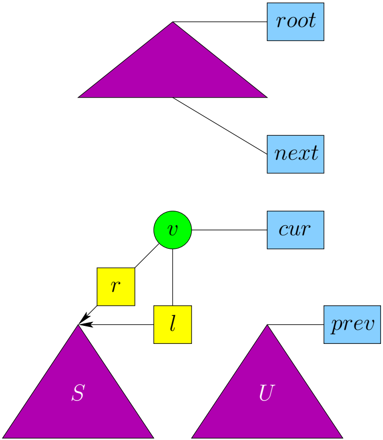
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;      ←
7  cur.r := prev;
8  prev := cur;
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



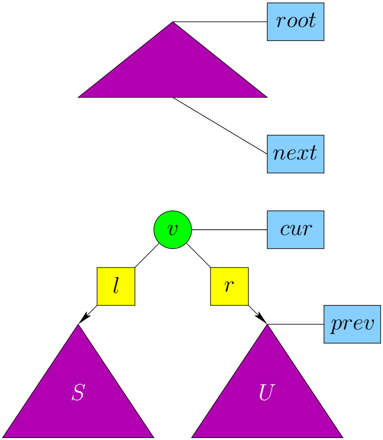
Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;
7  cur.r := prev;      ←
8  prev := cur;
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



Example: The Deutsch-Schorr-Waite Algorithm

```
1  if root = null goto 15;
2  new(sen);
3  prev := sen;
4  cur := root;
5  next := cur.l;
6  cur.l := cur.r;
7  cur.r := prev;
8  prev := cur;        ←
9  cur := next;
10 if (cur = sen) goto 15;
11 if (cur ≠ null) goto 5;
12 cur := prev;
13 prev := null;
14 goto 5;
```



Verification of Pointer Structures

Problems

- handling inputs of arbitrary size
- dynamic memory allocation at runtime

$\Rightarrow$ possibly infinite state space

Verification of Pointer Structures

Problems

- handling inputs of arbitrary size
- dynamic memory allocation at runtime

⇒ possibly infinite state space

Approach: Over-Approximation by Abstraction

- use HRGs to model data structures
- abstraction and concretization based on HRG rules

⇒ finite state spaces for e.g. model checking

Verification of Pointer Structures

Problems

- handling inputs of arbitrary size
- dynamic memory allocation at runtime

⇒ possibly infinite state space

Approach: Over-Approximation by Abstraction

- use HRGs to model data structures
- abstraction and concretization based on HRG rules

⇒ finite state spaces for e.g. model checking

Simple Pointer Programming Language (only pointers as data)

- pointer assignment ($x.a := y.b$)
- creation of objects ($\mathbf{new}(x)$)
- limited dereferencing depth (no real restriction)

Related Work

Shape Analysis represents unbounded heap graphs by three-valued logical structures [Sagiv et al., 2002, Beyer et al., 2006]

Separation Logic is an extension of Hoare logic [Reynolds, 2002, O'Hearn et al., 2004]

Graph Transformation is used in different approaches:

- abstraction and verification of graph transformation systems [Baldan and König, 2002, Baldan et al., 2004, Kastenberg and Rensink, 2006]
- model pointer assignments directly by graph transformations [Rensink, 2004, Rensink and Distefano, 2006]
- graph reduction grammars [Bakewell et al., 2004a, Bakewell et al., 2004b, Dodds and Plump, 2006]

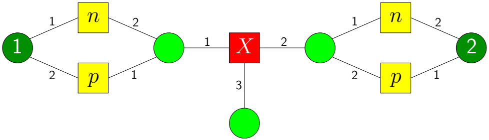
Alphabets and Hypergraphs

Ranked Alphabet Σ

- ranking function $rk : \Sigma \to \mathbb{N}$
- Σ consists of terminals and nonterminals: $\Sigma = T_\Sigma \uplus N_\Sigma$

Hypergraphs

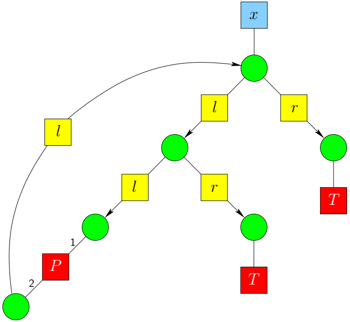
- hyperedges connect an arbitrary number of vertices
- hyperedges are labeled with symbols from Σ
- rank of label determines the arity of the edge
- external vertices are used for hyperedge replacements

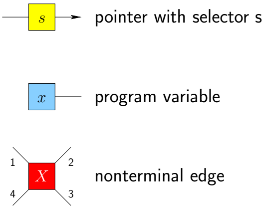


Representing Heap States

Heapgraph → Hypergraph

	Rank of Edges	Type of Label
pointers	2	terminal
program variables	1	variable (terminal)
abstract subgraphs	arbitrary	nonterminal





omit tentacle numbers when order clear

Concrete and Abstract Heaps

Abstract Heap

A heap configuration (=hypergraph) is abstract, if it contains at least one nonterminal edge.

Concrete and Abstract Heaps

Abstract Heap

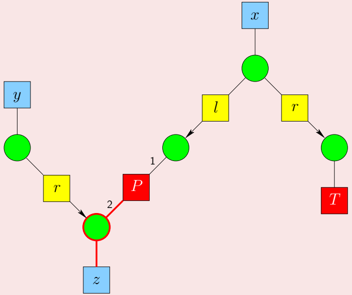
A heap configuration (=hypergraph) is abstract, if it contains at least one nonterminal edge.

Admissibility

A heap configuration is admissible if nodes referred by variables are not adjacent to nonterminal edges.

Useful for abstract semantics ("concrete assignment").

Inadmissible



Overview

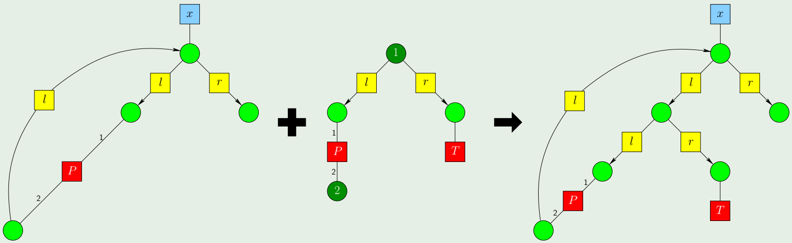
1. Hyperedge Replacement
2. Abstraction and Concretization
3. Pointer Logic
4. Verification and Model Checking

Hyperedge Replacement

Executing a hyperedge replacement

1. Hypergraph H with hyperedge $e \in E_H$ s.t. $\ell(e) \in N_\Sigma$
2. Hypergraph R with $|ext_R| = rk(e)$

Example

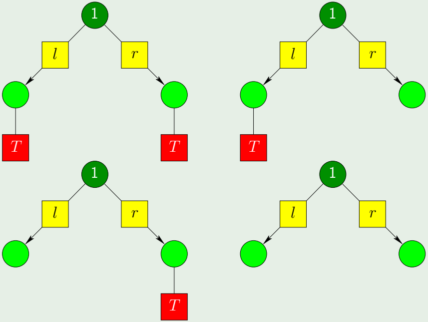


Hyperedge Replacement Grammars

Definition

A HRG G is a set of productions of the form $X \rightarrow R$ with $X \in N_{\Sigma}$ and hypergraph R where $|ext_R| = rk(X)$.

Example: HRG for (fully branched) Binary Trees



Properties

Context-freeness

HRGs are context-free and confluent.

Applicability

A rule is applicable to a hypergraph if it contains a nonterminal that matches the rule's LHS.

Derivation

A derivation is a sequence $H_0 \overset{G}{\Longrightarrow} H_1 \overset{G}{\Longrightarrow} H_2 \overset{G}{\Longrightarrow} \ldots$ where each $H_i \overset{G}{\Longrightarrow} H_{i+1}$ is a application of a rule from G.

Properties

Context-freeness

HRGs are context-free and confluent.

Applicability

A rule is applicable to a hypergraph if it contains a nonterminal that matches the rule's LHS.

Derivation

A derivation is a sequence $H_0 \overset{G}{\Longrightarrow} H_1 \overset{G}{\Longrightarrow} H_2 \overset{G}{\Longrightarrow} \ldots$ where each $H_i \overset{G}{\Longrightarrow} H_{i+1}$ is a application of a rule from G.

Graph Language of HRG G

$\mathfrak{L}(G, H) = \{K \in \mathrm{HGraph}_{T_\Sigma} \mid H \overset{G}{\Longrightarrow}^{\star} K\}$
(= all terminal graphs which are derivable from H)

Overview

1. Hyperedge Replacement
2. Abstraction and Concretization
3. Pointer Logic
4. Verification and Model Checking

Abstracting the Heap

Abstraction

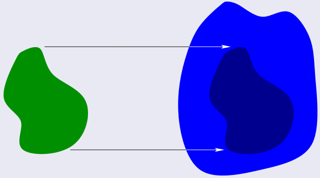
For HRG G and hypergraph H the set of abstractions of H is

$$\text{Abstractions}(H) = \{K \in \text{HGraph}_\Sigma \mid K \overset{G}{\Longrightarrow}^{+} H\}$$

If LHS $<$ RHS for all rules in G, Abstractions(H) is finite.

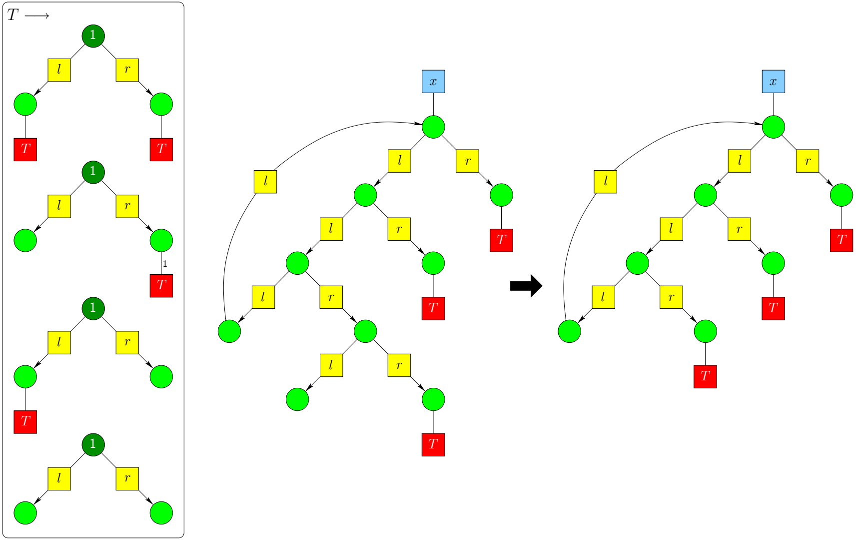
Idea

- Compute abstractions by reverse application of HRG rules
- Reverse application requires finding a subgraph isomorphism

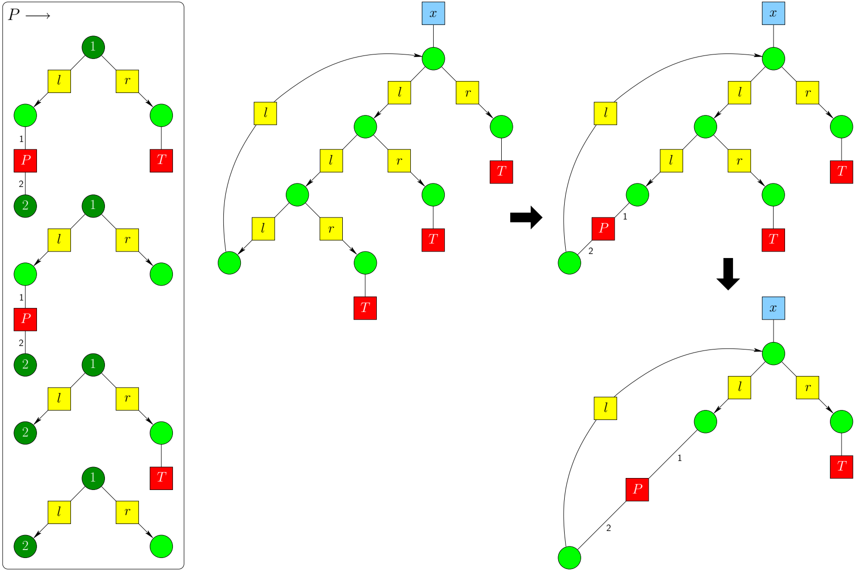


- Reverse rule application is not confluent

Abstraction Example - Binary Trees



Path Abstraction



Abstracting the Heap II

Correctness (but Over-Approximation)

By definition every concrete heap configuration can be regenerated from its abstractions.

$$\text{Abstractions}(H) = \{K \in \mathrm{HGraph}_\Sigma \mid K \stackrel{G}{\Longrightarrow}^{+} H\}$$

Abstracting the Heap II

Correctness (but Over-Approximation)

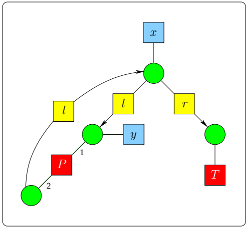
By definition every concrete heap configuration can be regenerated from its abstractions.

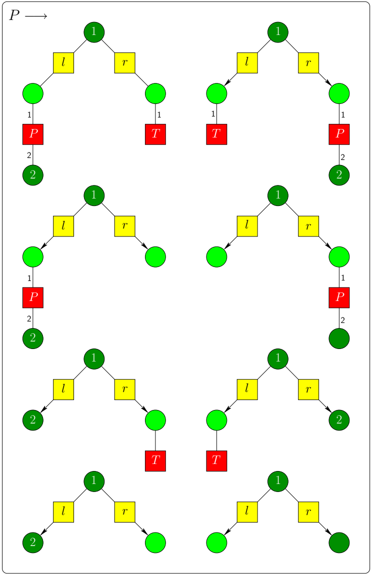
$$\text{Abstractions}(H) = \{K \in \mathrm{HGraph}_\Sigma \mid K \stackrel{G}{\Longrightarrow}^{+} H\}$$

Abstraction alone insufficient

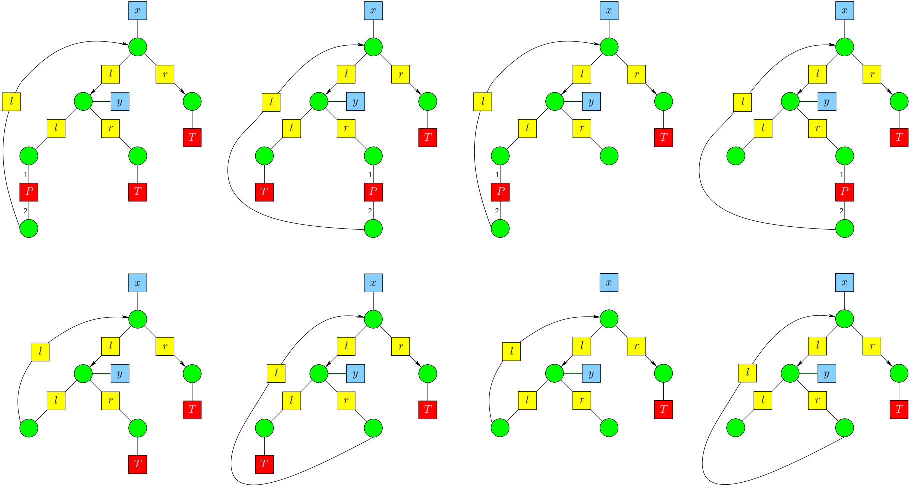
- Assignment easy since admissibility guarantees concrete edges near variables.
- But: assignments may yield inadmissible configurations
- Idea: materialize concrete objects from nonterminals (partial concretization)

Partial Concretization by Forward Rule Application

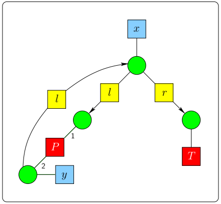


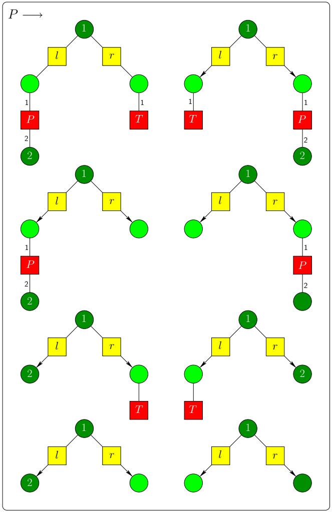


Resulting Hypergraphs

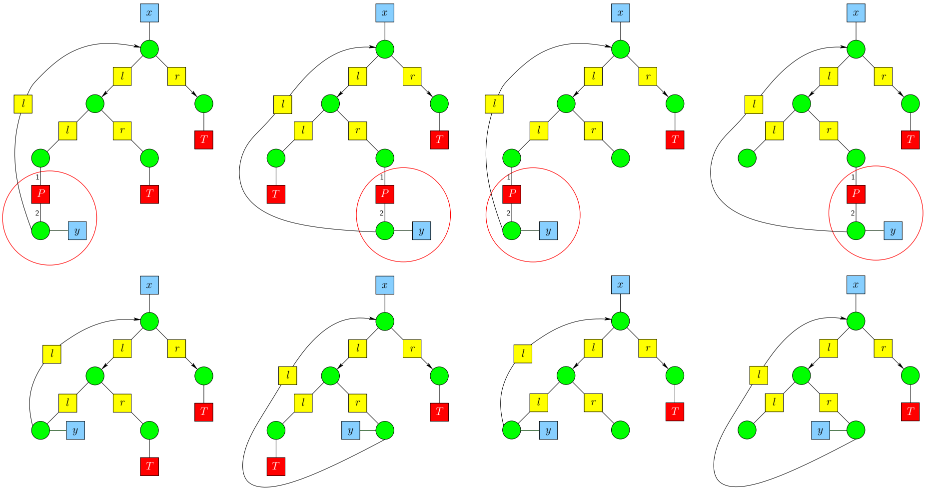


Different Situation





Inadmissible Results

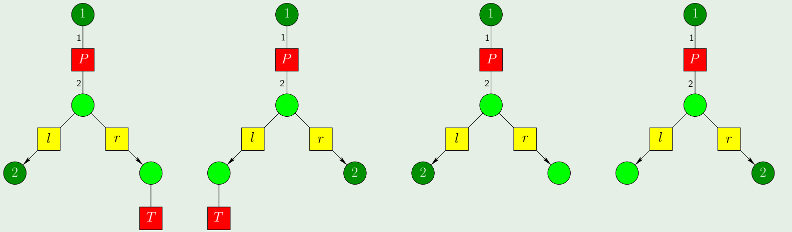


The Solution

Heap Abstraction Grammars

- introduce redundant rules allowing concretization "from below"
- additional rules must guarantee completeness

Additional Rules



Overview

1. Hyperedge Replacement
2. Abstraction and Concretization
3. Pointer Logic
4. Verification and Model Checking

Temporal Pointer Logic

- Combination of LTL operators and pointer comparisons
- Arbitrarily deep dereferencing

Formal Definition

Let $\mathfrak{F}$ be a set of flags ($\texttt{err}, \texttt{term}, \ldots$).

$$\begin{aligned}\mathrm{TPL}(\Sigma, \mathfrak{F}) ::= \ & \textsc{true} \mid \mathfrak{F} \mid \textsc{Deref}_\Sigma = \textsc{Deref}_\Sigma \\ & \mid \neg \mathrm{TPL}(\Sigma, \mathfrak{F}) \mid \mathrm{TPL}(\Sigma, \mathfrak{F}) \wedge \mathrm{TPL}(\Sigma, \mathfrak{F}) \\ & \mid \mathbf{X}\ \mathrm{TPL}(\Sigma, \mathfrak{F}) \mid \mathrm{TPL}(\Sigma, \mathfrak{F}) \mathbf{U}\ \mathrm{TPL}(\Sigma, \mathfrak{F})\end{aligned}$$

$$\textsc{Deref}_\Sigma ::= \mathrm{null} \mid \mathit{Var}_\Sigma \mid \textsc{Deref}_\Sigma\ .\ \mathit{Sel}_\Sigma$$

$$\mathbf{F}\varphi = \textsc{true}\ \mathbf{U}\ \varphi \qquad \mathbf{G}\ \varphi = \neg \mathbf{F}\ \neg\varphi$$

Semantics of TPL

Interpretation

- Interpret TPL formulae on infinite and finite sequences of heap configurations.
- Every trace of heap configurations has an associated trace of (sets of) flags of equal length.

Finite Traces

Let $t \in \mathrm{aHHC}_\Sigma{}^\star$ and $u \in \mathfrak{F}^\star$ be a finite traces of length n. Implicit extension as follows:

$$\begin{array}{cccccccc} t(1) & t(2) & \ldots & t(n) & t(n) & t(n) & \ldots \\ u(1) & u(2) & \ldots & u(n) & u(n) \cup \{\mathtt{term}\} & u(n) \cup \{\mathtt{term}\} & \ldots \end{array}$$

Formal Semantics of Pointer Comparisons

Concrete Semantics

$$\textsc{CSat}[\![\xi = \zeta, H]\!] = \begin{cases} 1 & \mathcal{D}[\![\xi, H]\!] = \mathcal{D}[\![\zeta, H]\!] \neq \bot \\ 0 & \text{otherwise} \end{cases}$$

$\mathcal{D}[\![\zeta, H]\!]$: "intuitive" concrete expression semantics

Formal Semantics of Pointer Comparisons

Concrete Semantics

$$\text{CSAT}[\![\xi = \zeta, H]\!] = \begin{cases} 1 & \mathcal{D}[\![\xi, H]\!] = \mathcal{D}[\![\zeta, H]\!] \neq \perp \\ 0 & \text{otherwise} \end{cases}$$

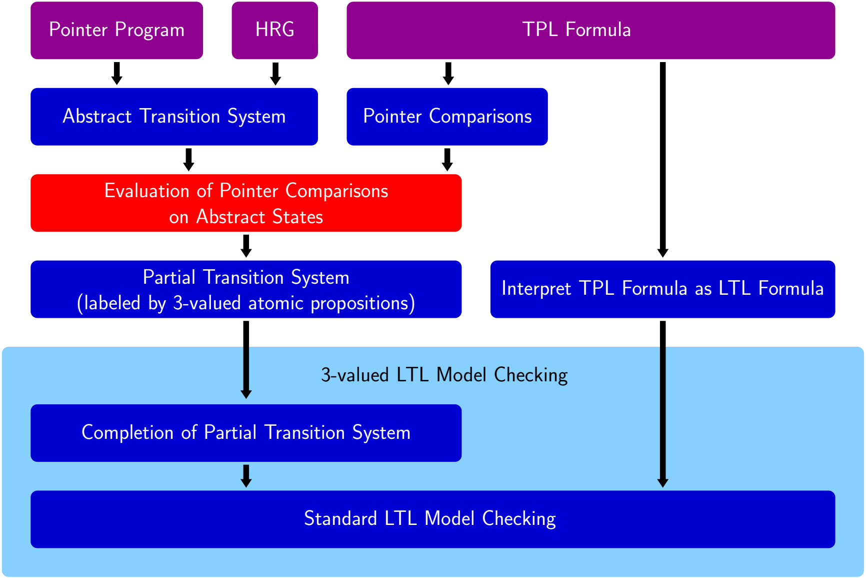
$\mathcal{D}[\![\zeta, H]\!]$: "intuitive" concrete expression semantics

Abstract Semantics – 3 cases

$$\text{ASAT}[\![\gamma, H]\!] = \begin{cases} 1 & \text{if } \forall H' \in \mathfrak{L}(G, H) : \text{CSAT}[\![\gamma, H']\!] = 1 \\ 0 & \text{if } \forall H' \in \mathfrak{L}(G, H) : \text{CSAT}[\![\gamma, H']\!] = 0 \\ \frac{1}{2} & \text{otherwise} \end{cases}$$

Overview

1. Hyperedge Replacement
2. Abstraction and Concretization
3. Pointer Logic
4. Verification and Model Checking

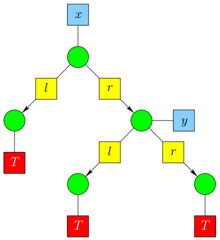

Pointer Program
HRG
TPL Formula
Abstract Transition System
Pointer Comparisons
Evaluation of Pointer Comparisons on Abstract States
Partial Transition System (labeled by 3-valued atomic propositions)
Interpret TPL Formula as LTL Formula
3-valued LTL Model Checking
Completion of Partial Transition System
Standard LTL Model Checking

Evaluating Pointer Comparisons

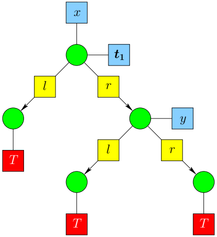
$x.a_1.a_2...a_m = y.b_1.b_2...b_n$

- Use two auxiliary variables t_1 and t_2 to walk along "paths"
- Assignments followed (not preceeded) by concretization steps
- Check if in all concretizations $t_1 = t_2$

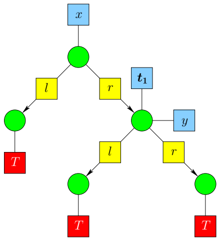
Does $\text{ASAT}[\![x.r.l.r = y.l.r, H]\!] = 1$ hold?



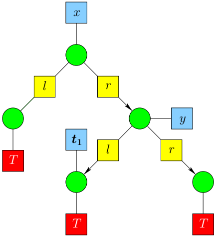
Does $\text{ASAT}[\![x.r.l.r = y.l.r, H\,]\!] = 1$ hold?



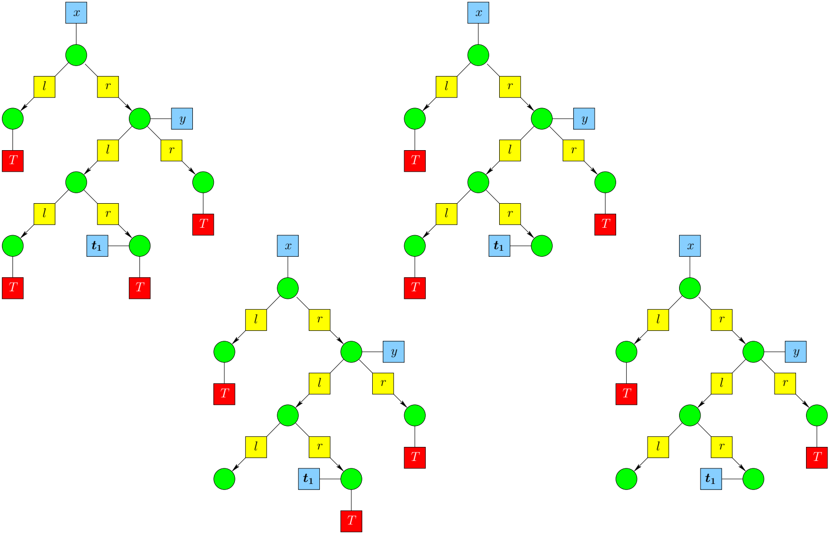
Does $\textsc{ASat}[\![x.r.l.r = y.l.r, H]\!] = 1$ hold?



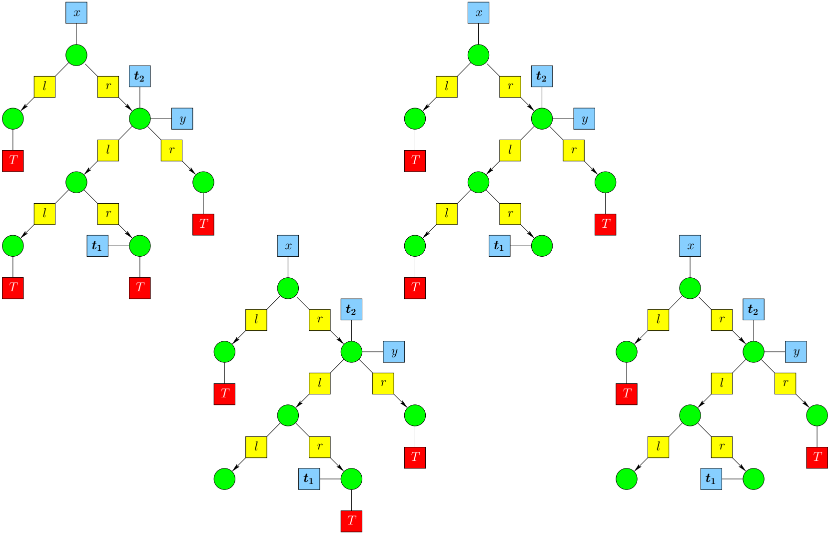
Does $\mathrm{AS{\scriptsize AT}}[\![x.r.l.r = y.l.r, H]\!] = 1$ hold?



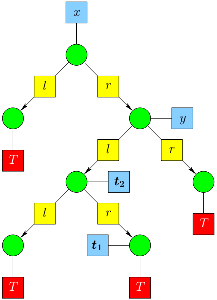
Does $\text{ASAT}[\![x.r.l.r = y.l.r, H]\!] = 1$ hold?

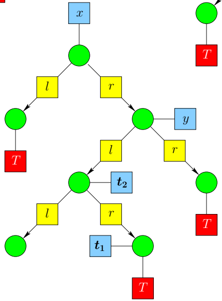


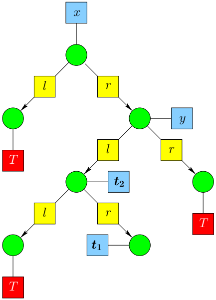
Does $\text{ASAT}[\![x.r.l.r = y.l.r, H]\!] = 1$ hold?

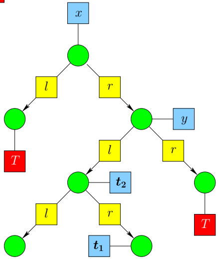


Does $\mathrm{ASAT}[\![x.r.l.r = y.l.r, H]\!] = 1$ hold?

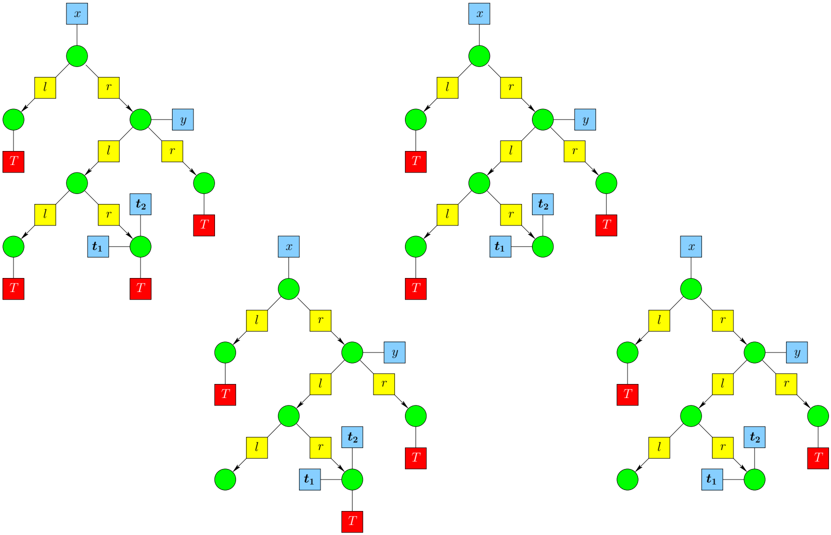








$\forall K : \text{CSAT}[\![t_1 = t_2, H]\!] = 1 \;\Rightarrow\; \text{ASAT}[\![\ldots]\!] = 1$

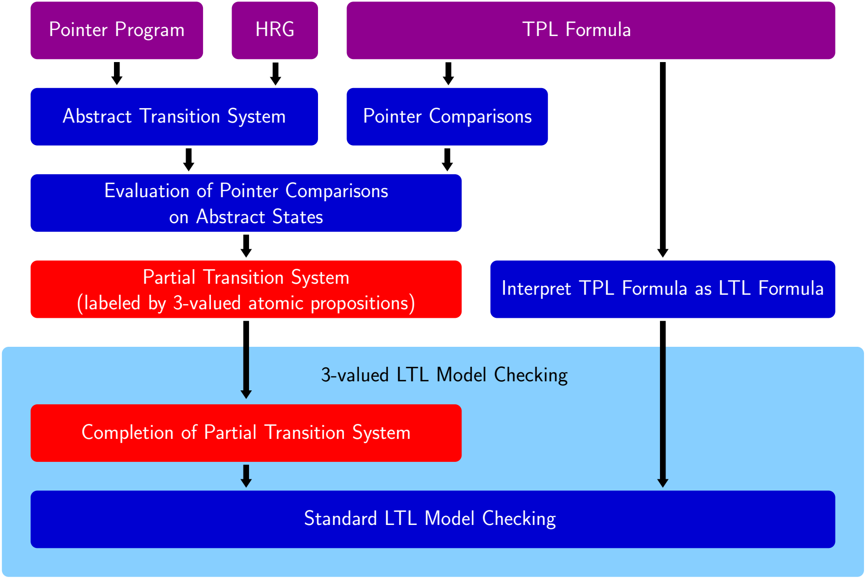


A Special Case

Limiting Dereferencing Depth

When dereferencing depth in pointer comparisons is limited to one, we always get clearly determined results (0 or 1).

Reason: admissibility of heap configurations


Pointer Program
HRG
TPL Formula
Abstract Transition System
Pointer Comparisons
Evaluation of Pointer Comparisons on Abstract States
Partial Transition System (labeled by 3-valued atomic propositions)
Interpret TPL Formula as LTL Formula
3-valued LTL Model Checking
Completion of Partial Transition System
Standard LTL Model Checking

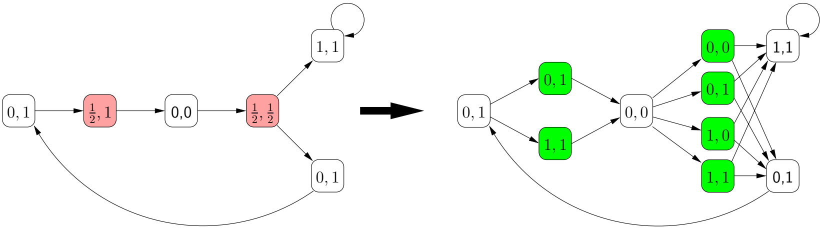
Three-valued LTL Model Checking

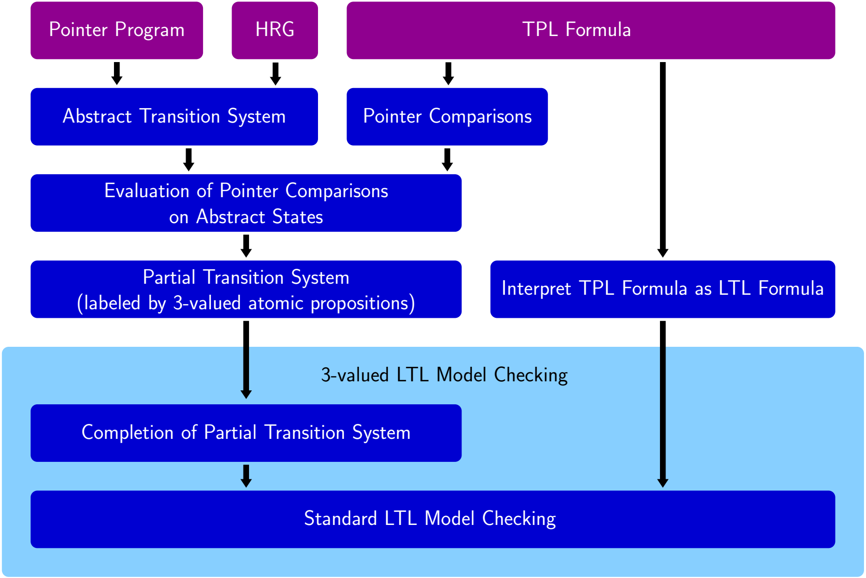
Setting

- Evaluation of pointer comparisons can result in either $0, 1$ or $\frac{1}{2}$
- Transition system has 3-valued labeling

Transformation

Transform transition system to represent all possibilities for $\frac{1}{2}$-valued predicates.




Pointer Program
HRG
TPL Formula
Abstract Transition System
Pointer Comparisons
Evaluation of Pointer Comparisons on Abstract States
Partial Transition System (labeled by 3-valued atomic propositions)
Interpret TPL Formula as LTL Formula
3-valued LTL Model Checking
Completion of Partial Transition System
Standard LTL Model Checking

Quantifiers

Quantified TPL

$$Q_1 X_1 Q_2 X_2 \dots Q_k X_k : \varphi(X_1, X_2, \dots, X_k)$$

- Quantification over heap objects present in the initial states
- Preservation of object identities between states by nondeterministic marking with variables
- For every quantor an additional marking is necessary (exponential blow-up of state space)

Example: The Deutsch-Schorr-Waite Algorithm

Pointer Safety: No pointer errors / null dereferences

Shape Safety: Input structure is retained

Completeness: all vertices are visited at least once

$$\forall X : \neg(cur \neq X \ \mathbf{U} \ \texttt{term})$$

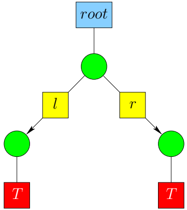
Termination: finally X never points to cur anymore

$$\forall X : \mathbf{FG}(cur \neq X)$$

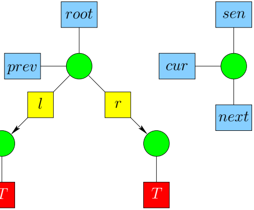
Correctness: for all vertices the left- and right successors are the same after program termination

$$\forall X \ \forall X_l \ \forall X_r : \ X.l = X_l \ \wedge \ X.r = X_r \rightarrow ((X = root \rightarrow \mathbf{G}(X = root)) \ \wedge \ \mathbf{G}(\texttt{term} \rightarrow (X.l = X_l \ \wedge \ X.r = X_r)))$$

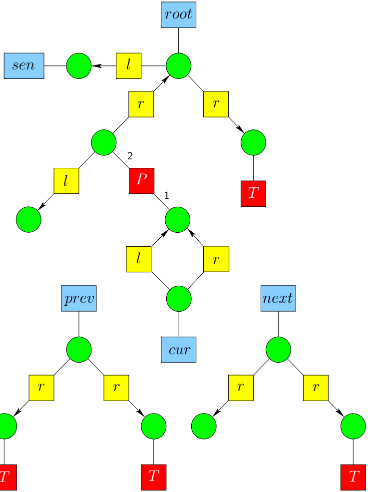
Experimental Results: Verifying the DSW Algorithm



an initial heap



a final heap



an intermediate heap state

Experimental Results: Verifying the DSW Algorithm

	no marking	1 marking	3 markings	TVLA
Initial States	5	185	962	
Number of States	20,678	6,220,798	35,983,627	$>$ 80,000
Number of Transitions	23,359	7,078,257	40,909,648	
State Space Gen. (h:min:sec)	$<$0:01	10:14	1:18:03	
Memory Consumption	41 MB	788 MB	3,900 MB	150 MB
Pointer Safety	on-the-fly	-	-	
Shape Safety	on-the-fly	-	-	
Completeness (min:sec)	-	0:16	-	
Termination (min:sec)	-	0:39	-	
Correctness (min:sec)	-	-	4:05	
Total Time (State Space Gen. + all Properties)			1:28:35	$<$9:00:00

Conclusion

- analysis and verification of complex data structures
- highly parametrized framework
- handling of inconsistencies wrt. the data structure
- more intuitive than other approaches
- promising experimental results

Conclusion

- analysis and verification of complex data structures
- highly parametrized framework
- handling of inconsistencies wrt. the data structure
- more intuitive than other approaches
- promising experimental results

Additional Features

- abstraction-only grammars
- optimized concretization possible
- unbounded thread creation [Noll and Rieger, 2008]

Conclusion

- analysis and verification of complex data structures
- highly parametrized framework
- handling of inconsistencies wrt. the data structure
- more intuitive than other approaches
- promising experimental results

Additional Features

- abstraction-only grammars
- optimized concretization possible
- unbounded thread creation [Noll and Rieger, 2008]

Outlook

- learning of HRGs
- typed/attributed HRGs

(FM 2008) Thomas Noll and Stefan Rieger. Verifying Dynamic Pointer-Manipulating Threads

(ICGT 2008) Stefan Rieger and Thomas Noll. Abstracting Complex Data Structures by Hyperedge Replacement

(TTSS 2009) Jonathan Heinen, Thomas Noll, and Stefan Rieger. Juggrnaut: Graph Grammar Abstraction for Unbounded Heap Structures (to be published)

(ICTAC 2007) Thomas Noll and Stefan Rieger. Composing Transformations to Optimize Linear Code

Thank you for your attention!

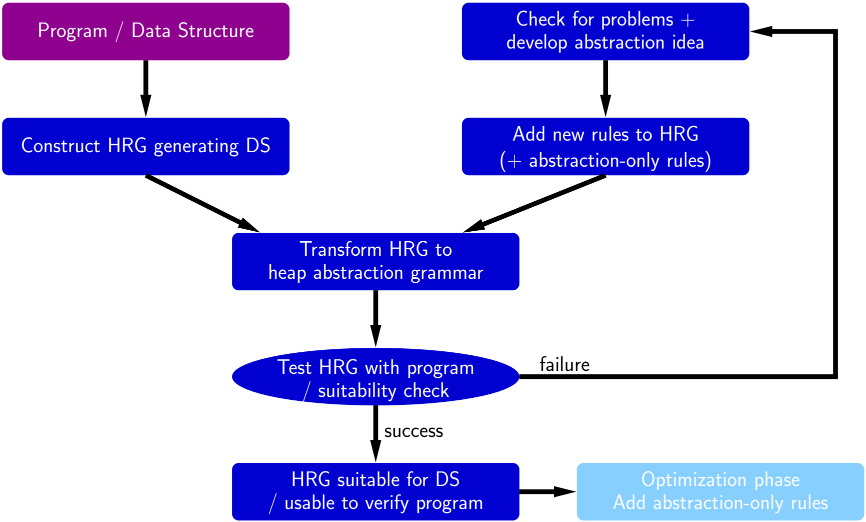
(FM 2008) Thomas Noll and Stefan Rieger. Verifying Dynamic Pointer-Manipulating Threads

(ICGT 2008) Stefan Rieger and Thomas Noll. Abstracting Complex Data Structures by Hyperedge Replacement

(TTSS 2009) Jonathan Heinen, Thomas Noll, and Stefan Rieger. Juggrnaut: Graph Grammar Abstraction for Unbounded Heap Structures (to be published)

(ICTAC 2007) Thomas Noll and Stefan Rieger. Composing Transformations to Optimize Linear Code

Development of HRGs



Partial Concretization II

Solving the Problem

- Enforcing HRGs to be in apex form (for all $X \to H$, the nodes ext_H are only adjacent to terminals)

Partial Concretization II

Solving the Problem

- Enforcing HRGs to be in apex form (for all $X \rightarrow H$, the nodes ext_H are only adjacent to terminals)

$\Rightarrow$ impractical [Engelfriet, 1992]

Partial Concretization II

Solving the Problem

- Enforcing HRGs to be in apex form (for all $X \to H$, the nodes ext_H are only adjacent to terminals)

⇒ impractical [Engelfriet, 1992]

⇒ Introducing additional redundant grammar-rules that do not modify the language

Partial Concretization II

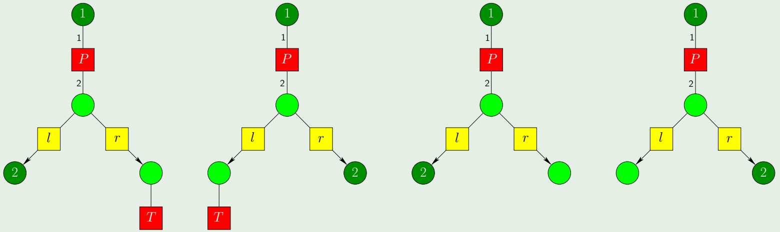
Solving the Problem

- Enforcing HRGs to be in apex form (for all $X \to H$, the nodes ext_H are only adjacent to terminals)

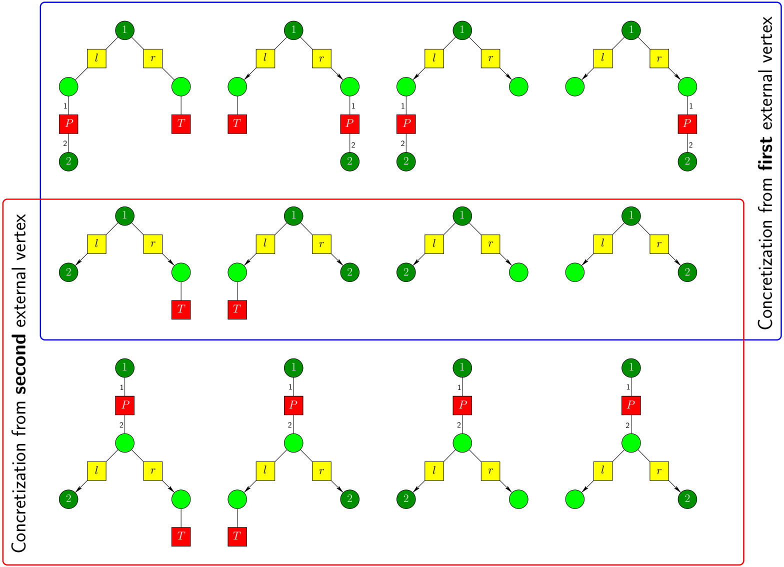
$\Rightarrow$ impractical [Engelfriet, 1992]

$\Rightarrow$ Introducing additional redundant grammar-rules that do not modify the language

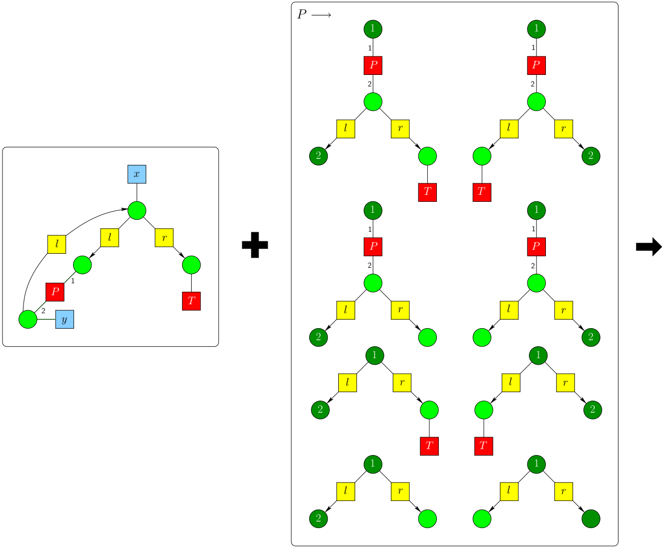
Additional Rules



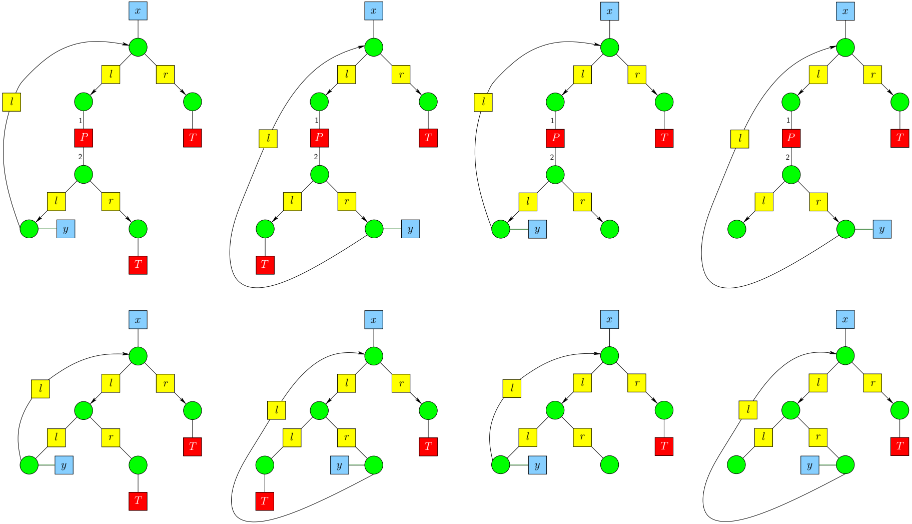
All P-Rules


Concretization from **first** external vertex
Concretization from **second** external vertex
1
2
l
r
P
T

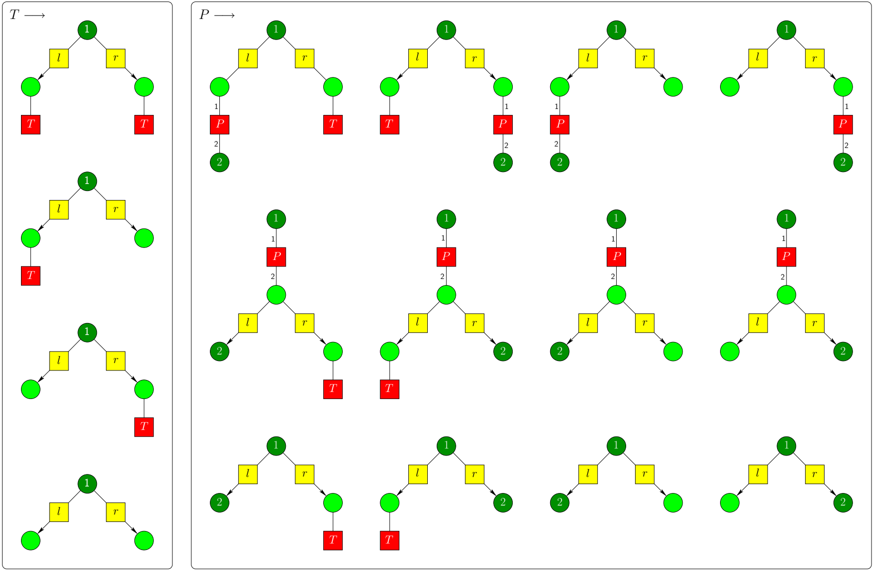
Again the Example



Admissible Results



Entire Grammar



Bakewell, A., Plump, D., and Runciman, C. (2004a).
Checking the shape safety of pointer manipulations.
In *Relational and Kleene-Algebraic Methods in Computer Science '03*, volume 3051 of *Lecture Notes in Computer Science*, pages 48–61. Springer.

Bakewell, A., Plump, D., and Runciman, C. (2004b).
Specifying pointer structures by graph reduction.
In *Applications of Graph Transformations with Industrial Relevance '03*, volume 3062 of *Lecture Notes in Computer Science*, pages 30–44. Springer.

Baldan, P., Corradini, A., and König, B. (2004).
Verifying Finite-State Graph Grammars: An Unfolding-Based Approach.
In *CONCUR '04*, volume 3170 of *Lecture Notes in Computer Science*, pages 83–98. Springer.

Baldan, P. and König, B. (2002).

Approximating the behaviour of graph transformation systems.
In *1st International Conference on Graph Transformations, ICGT 2002*, volume 2505 of *Lecture Notes in Computer Science*, pages 14–29. Springer.

Beyer, D., Henzinger, T. A., and Théoduloz, G. (2006).
Lazy shape analysis.
In *Computer Aided Verification, 18th International Conference, CAV '06*, volume 4144 of *Lecture Notes in Computer Science*, pages 532–546. Springer.

Dodds, M. and Plump, D. (2006).
Extending C for checking shape safety.
In *Graph Transformation for Verification and Concurrency '05*, volume 154(2) of *ENTCS*, pages 95–112. Elsevier.

Engelfriet, J. (1992).
A Greibach Normal Form for Context-Free Graph Grammars.

In *19th International Colloquium on Automata, Languages and Programming, ICALP 1992*, volume 623 of *Lecture Notes in Computer Science*, pages 138–149. Springer.

Kastenberg, H. and Rensink, A. (2006).
Model checking dynamic states in GROOVE.
In *Model Checking Software (SPIN '06)*, volume 3925 of *Lecture Notes in Computer Science*, pages 299–305. Springer.

Noll, T. and Rieger, S. (2008).
Verifying dynamic pointer-manipulating threads.
In *15th International Symposium on Formal Methods (FM '08)*, volume 5014 of *Lecture Notes in Computer Science*, pages 84–99. Springer.

O'Hearn, P. W., Yang, H., and Reynolds, J. C. (2004).
Separation and information hiding.

In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004*, pages 268–280. ACM Press.

Rensink, A. (2004).
Canonical graph shapes.
In *Proc. of 13th European Symposium on Programming (ESOP '04)*, volume 2986 of *Lecture Notes in Computer Science*, pages 401–415. Springer.

Rensink, A. and Distefano, D. (2006).
Abstract graph transformation.
In *Proc. of Int. Workshop on Software Verification and Validation (SVV '05)*, volume 157(1) of *Electr. Notes Theor. Comput. Sci.*

Reynolds, J. C. (2002).
Separation logic: A logic for shared mutable data structures.

In *IEEE Symposium on Logic in Computer Science, LICS 2002*, pages 55–74. IEEE Computer Society.

Sagiv, M., Reps, T., and Wilhelm, R. (2002).
Parametric shape analysis via 3–valued logic.
ACM Transactions on Programming Languages and Systems, 24(3):217–298.