

Abstracting Complex Data Structures by Hyperedge Replacement

Thomas Noll Stefan Rieger

MOVES: Software Modeling and Verification
RWTH Aachen University, Germany

09.09.2008



ICGT 2008, Leicester/UK

Verification of Pointer Structures

Problem

- unbounded **heap space**
 - destructive updates via pointers
 - **arbitrary** data structures
- ⇒ possibly **infinite state space**

Verification of Pointer Structures

Problem

- unbounded **heap space**
 - destructive updates via pointers
 - **arbitrary** data structures
- ⇒ possibly **infinite state space**

Approach: Abstraction

- use **HRGs** to model data structures
 - **abstraction** and **concretization** based on HRG rules
- ⇒ **finite state spaces** for e.g. model checking

Pointer Programs

Programming language

- pointer assignment ($x.a := y.b$)
- creation of objects ($\mathbf{new}(x)$)
- conditional ($\mathbf{if...goto}$) and unconditional jumps

Delete on Cyclic DLL

```
delete() {  
  1 if  $x = nil$  goto 10;  
  2 if  $x = x.n$  goto 9;  
  3  $y := x.n$ ;  
  4  $x := x.p$ ;  
  5  $x.n := y$ ;  
  6  $y.p := x$ ;  
  7  $y := nil$ ;  
  8 goto 10;  
  9  $x := nil$ ;  
 10 }
```

Overview

- ① Heaps as Hypergraphs
- ② Hyperedge Replacement
- ③ Abstraction and Concretization
- ④ Abstract Semantics

Hypergraphs

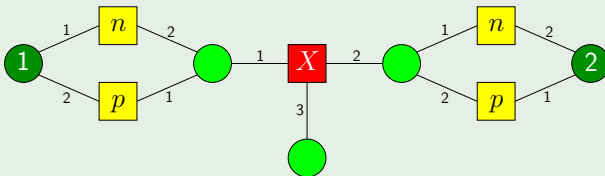
Definition

Given: Alphabet $\Sigma = T_\Sigma \uplus N_\Sigma$, ranking function $rk : \Sigma \rightarrow \mathbb{N}$

Define: $H = (V, E, att, \ell, ext) \in \text{HGraph}_\Sigma$ where

- set of nodes V , set of edges E

Example



Hypergraphs

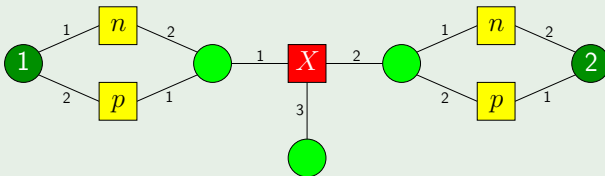
Definition

Given: Alphabet $\Sigma = T_\Sigma \uplus N_\Sigma$, ranking function $rk : \Sigma \rightarrow \mathbb{N}$

Define: $H = (V, E, att, \ell, ext) \in \text{HGraph}_\Sigma$ where

- set of nodes V , set of edges E
- attachment function $att : E \rightarrow V^*$

Example



Hypergraphs

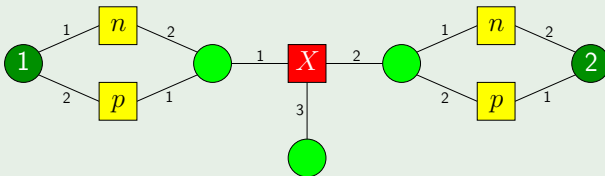
Definition

Given: Alphabet $\Sigma = T_\Sigma \uplus N_\Sigma$, ranking function $rk : \Sigma \rightarrow \mathbb{N}$

Define: $H = (V, E, att, \ell, ext) \in \text{HGraph}_\Sigma$ where

- set of nodes V , set of edges E
- attachment function $att : E \rightarrow V^*$
- labeling function $\ell : E \rightarrow \Sigma$ $rk(e) = rk(\ell(e))$

Example



Hypergraphs

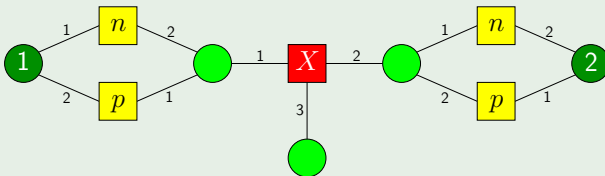
Definition

Given: Alphabet $\Sigma = T_\Sigma \uplus N_\Sigma$, ranking function $rk : \Sigma \rightarrow \mathbb{N}$

Define: $H = (V, E, att, \ell, ext) \in \text{HGraph}_\Sigma$ where

- set of nodes V , set of edges E
- attachment function $att : E \rightarrow V^*$
- labeling function $\ell : E \rightarrow \Sigma$ $rk(e) = rk(\ell(e))$
- sequence of (pairw. dist.) external nodes $ext \in V^*$

Example



Representing Heap States

Heapgraph \rightarrow Hypergraph

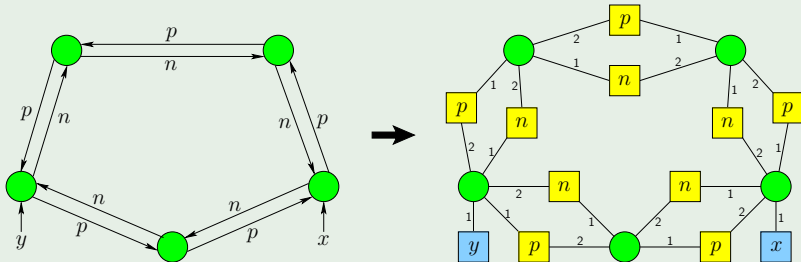
	Rank of Edges	Type of Label
pointers	2	terminal
program variables	1	variable (terminal)
abstract subgraphs	arbitrary	nonterminal

Representing Heap States

Heapgraph \rightarrow Hypergraph

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

Example: Cyclic Doubly Linked List



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

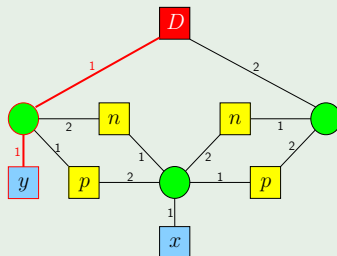
A heap configuration (=hypergraph) is **abstract**, if it contains at least one nonterminal edge.

Admissibility

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

Useful for abstract semantics (“**concrete** assignment”).

Example



Overview

- ① Heaps as Hypergraphs
- ② Hyperedge Replacement
- ③ Abstraction and Concretization
- ④ Abstract Semantics

Hyperedge Replacement

When can we execute a hyperedge replacement?

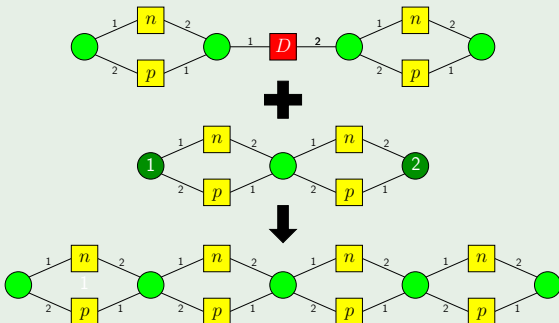
- 1 Hypergraph H with hyperedge $e \in E_H$ s.t. $\ell(e) \in N_\Sigma$
- 2 Hypergraph K with $|ext_K| = rk(e)$

Hyperedge Replacement

When can we execute a hyperedge replacement?

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

Example



Hyperedge Replacement Grammars

Definition

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

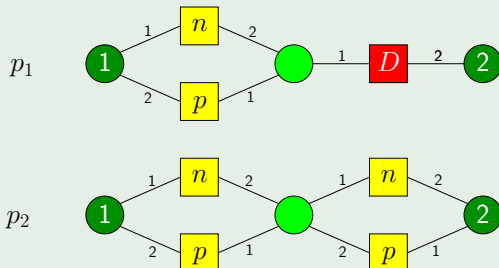
Hyperedge Replacement Grammars

Definition

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

Example: HRG for DLLs (with minimal size 3)

$D \longrightarrow$



Derivations

Applicability

A rule $X \rightarrow K$ is **applicable** to hypergraph H if it exists an $e \in E_H$ with $\ell(e) = X$.

Derivations

Applicability

A rule $X \rightarrow K$ is **applicable** to hypergraph H if it exists an $e \in E_H$ with $\ell(e) = X$.

Derivation

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

Derivations

Applicability

A rule $X \rightarrow K$ is **applicable** to hypergraph H if it exists an $e \in E_H$ with $\ell(e) = X$.

Derivation

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

Language of HRG G

$L(G, H) = \{K \in \text{HGraph}_{T_\Sigma} \mid H \Longrightarrow_G^* K\}$
(= all **terminal** graphs which are derivable from H)

Derivations

Applicability

A rule $X \rightarrow K$ is **applicable** to hypergraph H if it exists an $e \in E_H$ with $\ell(e) = X$.

Derivation

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

Language of HRG G

$L(G, H) = \{K \in \text{HGraph}_{T_\Sigma} \mid H \Longrightarrow_G^* K\}$
(= all **terminal** graphs which are derivable from H)

Note

Language of HRG = abstractable DS \subseteq allowed/verifiable DS

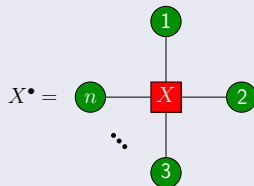
Overview

- ① Heaps as Hypergraphs
- ② Hyperedge Replacement
- ③ Abstraction and Concretization
- ④ Abstract Semantics

Necessary Preconditions

Productivity

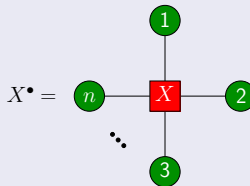
HRG G is productive if for all
 $X \rightarrow H \in G$ we have
 $L(G, X^\bullet) \neq \emptyset$.



Necessary Preconditions

Productivity

HRG G is productive if for all $X \rightarrow H \in G$ we have $L(G, X^\bullet) \neq \emptyset$.



Increasingness: LHS < RHS

For all $X \rightarrow H \in G$:

- H is a **concrete** graph, **or**
- H contains a **nonterminal**, **and**
 - H contains more edges than X^\bullet , **or**
 - H contains more vertices than X^\bullet .

Abstracting the Heap I

Abstraction

For HRG G and hypergraph H the set of **abstractions** of H is $\{K \in \text{HGraph}_\Sigma \mid K \Longrightarrow_G^+ H\}$.

Abstracting the Heap I

Abstraction

For HRG G and hypergraph H the set of **abstractions** of H is $\{K \in \text{HGraph}_\Sigma \mid K \Longrightarrow_G^+ H\}$.

Theorem

For **increasing** G the set abstractions of hypergraph H is **finite**.
(This implies the **termination** of the abstraction technique.)

Abstracting the Heap I

Abstraction

For HRG G and hypergraph H the set of **abstractions** of H is $\{K \in \text{HGraph}_\Sigma \mid K \Longrightarrow_G^+ H\}$.

Theorem

For **increasing** G the set abstractions of hypergraph H is **finite**.
(This implies the **termination** of the abstraction technique.)

Proof idea

- 1 Hypergraph H is abstract (contains a nonterminal):
For any H' with $H' \Longrightarrow_G H$ we have $|H'| < |H|$. Since H' is also abstract the same holds for H'' with $H'' \Longrightarrow_G H'$. This can only be done finitely often since $|H|$ is finite.
- 2 Hypergraph H is concrete (terminal):
Then all H' with $H' \Longrightarrow_G H$ are abstract and case 1 applies.

Abstracting the Heap II

Idea

- Represent **structured subgraphs** in the heap by nonterminals
- Compute abstractions by **reverse application** of HRG rules
- Computation according to abstraction **strategy**

Abstracting the Heap II

Idea

- Represent **structured subgraphs** in the heap by nonterminals
- Compute abstractions by **reverse application** of HRG rules
- Computation according to abstraction **strategy**

Parametrization

Abstraction function \mathfrak{A}_G yields an **admissible** abstraction.

- 1 **Smallest irreducible** abstraction (“brute-force approach”)
- 2 Abstraction by given **order of rules** until **irreducible** graph obtained
 - first applicable
 - largest redex
 - ...
- 3 ...

Abstracting the Heap III

Correctness

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

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

Abstracting the Heap III

Correctness

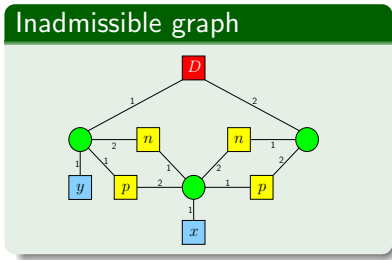
By definition every concrete heap configuration can be regenerated from its abstractions.

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

Abstraction alone insufficient

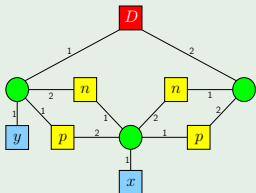
- Assignment itself easy since admissibility guarantees concrete edges near variables.
- Partial **concretization** necessary since **assignments** may **lead to inadmissible configurations**.

Partial Concretization I



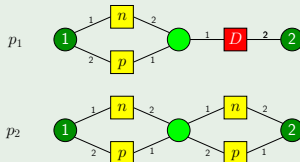
Partial Concretization I

Inadmissible graph



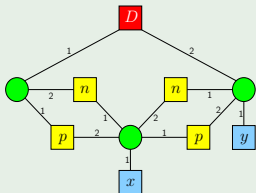
DLL-Grammar

$D \rightarrow$



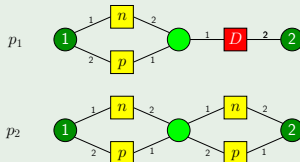
Partial Concretization II

Modified inadmissible graph



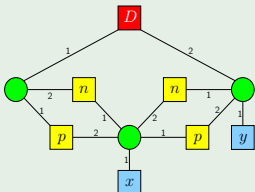
DLL-Grammar

$D \rightarrow$



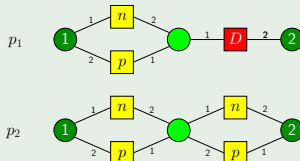
Partial Concretization II

Modified inadmissible graph

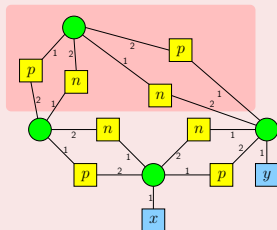
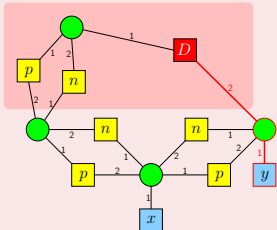


DLL-Grammar

$D \rightarrow$



Inadmissible result



Partial Concretization III

Solving the Problem

- Enforcing an extended **Greibach Normal Form** (for all $X \rightarrow H$, the nodes ext_H are only adjacent to terminals)

Partial Concretization III

Solving the Problem

- Enforcing an extended **Greibach Normal Form** (for all $X \rightarrow H$, the nodes ext_H are only adjacent to terminals)
⇒ **impractical** [Engelfriet, 1992]

Partial Concretization III

Solving the Problem

- Enforcing an extended **Greibach Normal Form** (for all $X \rightarrow H$, the nodes ext_H are only adjacent to terminals)
- ⇒ **impractical** [Engelfriet, 1992]
- ⇒ Introducing **additional redundant** grammar-rules that do not modify the language

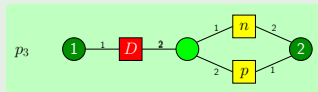
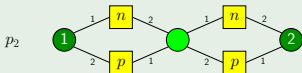
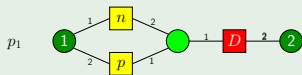
Partial Concretization III

Solving the Problem

- Enforcing an extended **Greibach Normal Form** (for all $X \rightarrow H$, the nodes ext_H are only adjacent to terminals)
- ⇒ **impractical** [Engelfriet, 1992]
- ⇒ Introducing **additional redundant** grammar-rules that do not modify the language

Example

$D \longrightarrow$

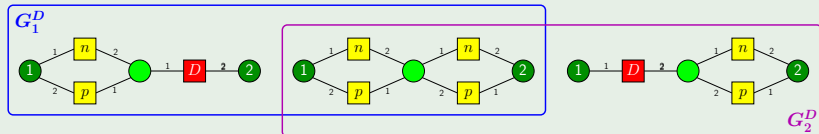


Partial Concretization IV

Formally

$\forall X \in N_\Sigma$ with $rk(X) = k$ there exist $G_1^X, \dots, G_k^X \subseteq G$ such that

Example



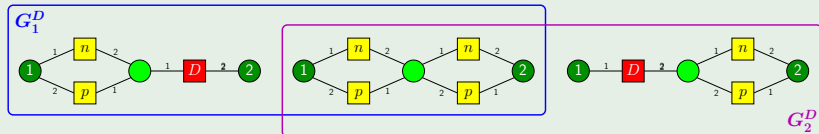
Partial Concretization IV

Formally

$\forall X \in N_\Sigma$ with $rk(X) = k$ there exist $G_1^X, \dots, G_k^X \subseteq G$ such that

- $\bigcup_{i=1}^k G_i^X = G$,

Example



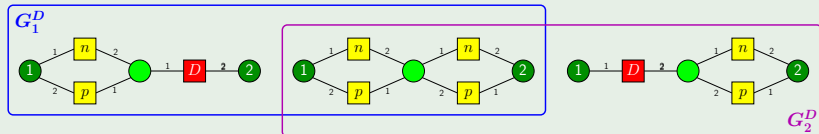
Partial Concretization IV

Formally

$\forall X \in N_\Sigma$ with $rk(X) = k$ there exist $G_1^X, \dots, G_k^X \subseteq G$ such that

- $\bigcup_{i=1}^k G_i^X = G$,
- $L(G_i^X, X^\bullet) = L(G, X^\bullet)$ for all $1 \leq i \leq k$, and

Example



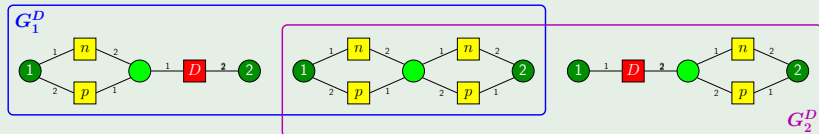
Partial Concretization IV

Formally

$\forall X \in N_\Sigma$ with $rk(X) = k$ there exist $G_1^X, \dots, G_k^X \subseteq G$ such that

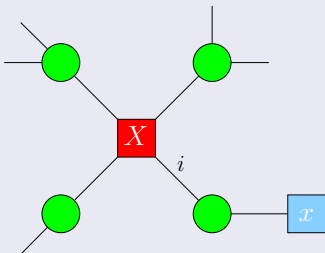
- $\bigcup_{i=1}^k G_i^X = G$,
- $L(G_i^X, X^\bullet) = L(G, X^\bullet)$ for all $1 \leq i \leq k$, and
- $\ell(E(ext(i))) \subseteq T_\Sigma$ for all $X \rightarrow (V, E, att, \ell, ext) \in G_i^X$.

Example



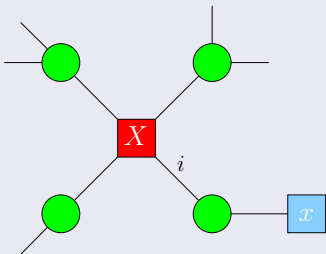
Partial Concretization V

General Case



Partial Concretization V

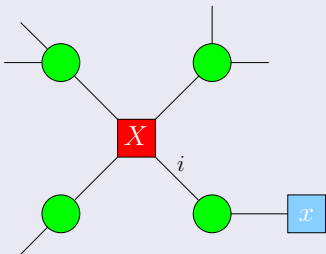
General Case



Apply all rules from G_i^X for concretization

Partial Concretization V

General Case

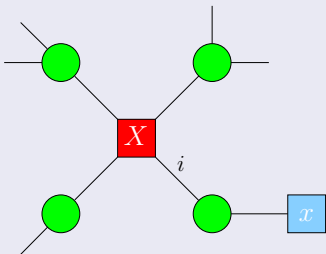


Apply all rules from G_i^X for concretization

Since $L(G_i^X, X^\bullet) = L(G, X^\bullet)$
(by def.) this is complete.

Partial Concretization V

General Case



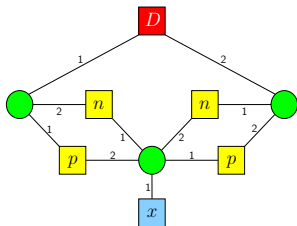
Apply all rules from G_i^X for concretization

Since $L(G_i^X, X^\bullet) = L(G, X^\bullet)$ (by def.) this is complete.

Since $\ell(E(\text{ext}(i))) \subseteq T_\Sigma$ for all $X \rightarrow (V, E, \text{att}, \ell, \text{ext}) \in G_i^X$ the result is admissible.

Overview

- ① Heaps as Hypergraphs
- ② Hyperedge Replacement
- ③ Abstraction and Concretization
- ④ Abstract Semantics

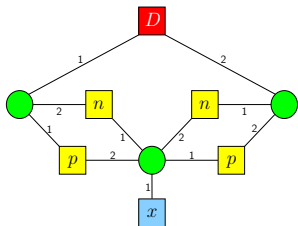


Steps

- 1 assignment
- 2 garbage collection
- 3 concretization
- 4 re-abstraction

Program

```
delete() {  
  1 if  $x = nil$  goto 10;  
  2 if  $x = x.n$  goto 9;  
  3  $y := x.n$ ;  
  4  $x := x.p$ ;  
  5  $x.n := y$ ;  
  6  $y.p := x$ ;  
  7  $y := nil$ ;  
  8 goto 10;  
  9  $x := nil$ ;  
}
```



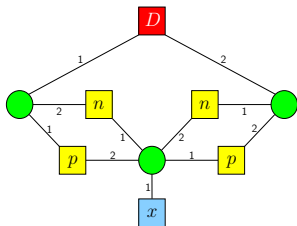
Steps

- ① assignment
- ② garbage collection
- ③ concretization
- ④ re-abstraction

Program

```

delete() {
  1 if x = nil goto 10;
  2 if x = x.n goto 9;
  3 y := x.n;
  4 x := x.p;
  5 x.n := y;
  6 y.p := x;
  7 y := nil;
  8 goto 10;
  9 x := nil;}
  
```



Steps

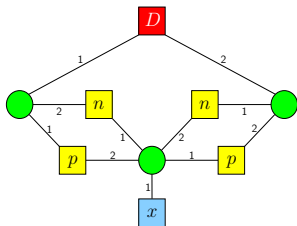
- 1 assignment
- 2 garbage collection
- 3 concretization
- 4 re-abstraction

Program

```

delete() {
  1 if  $x = nil$  goto 10;
  2 if  $x = x.n$  goto 9;
  3  $y := x.n$ ;
  4  $x := x.p$ ;
  5  $x.n := y$ ;
  6  $y.p := x$ ;
  7  $y := nil$ ;
  8 goto 10;
  9  $x := nil$ ;
}

```

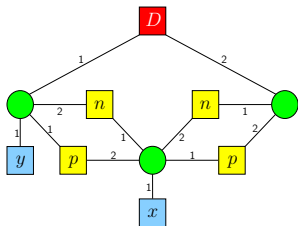


Steps

- 1 assignment
- 2 garbage collection
- 3 concretization
- 4 re-abstraction

Program

```
delete() {  
  1 if  $x = nil$  goto 10;  
  2 if  $x = x.n$  goto 9;  
  3  $y := x.n$ ;  
  4  $x := x.p$ ;  
  5  $x.n := y$ ;  
  6  $y.p := x$ ;  
  7  $y := nil$ ;  
  8 goto 10;  
  9  $x := nil$ ;  
}
```



Steps

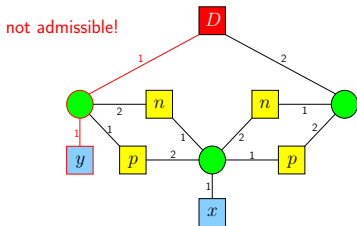
- ① assignment
- ② garbage collection
- ③ concretization
- ④ re-abstraction

Program

```

delete() {
  1 if x = nil goto 10;
  2 if x = x.n goto 9;
  3 y := x.n;
  4 x := x.p;
  5 x.n := y;
  6 y.p := x;
  7 y := nil;
  8 goto 10;
  9 x := nil;}

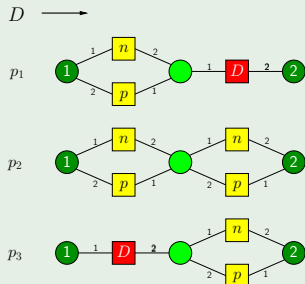
```

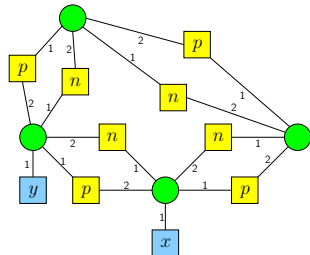
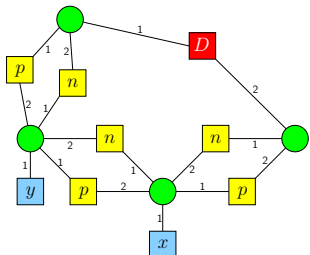


Steps

- 1 assignment
- 2 garbage collection
- 3 concretization
- 4 re-abstraction

HRG

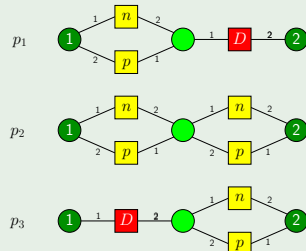


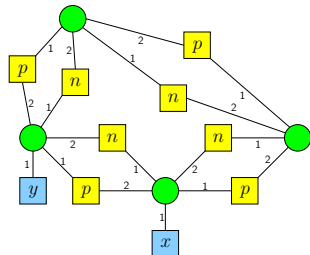
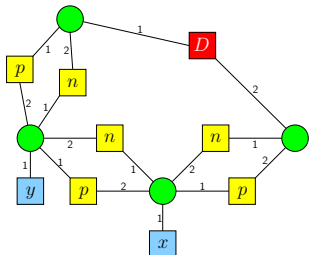


Steps

- ① assignment
- ② garbage collection
- ③ concretization
- ④ re-abstraction

HRG

 $D \longrightarrow$




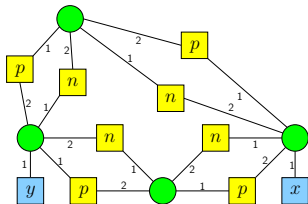
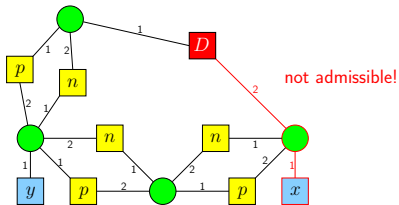
Steps

- 1 assignment
- 2 garbage collection
- 3 concretization
- 4 re-abstraction

Program

```

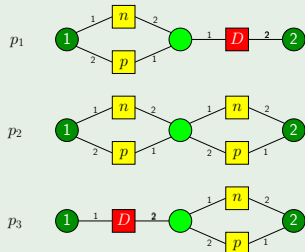
delete() {
  1 if x = nil goto 10;
  2 if x = x.n goto 9;
  3 y := x.n;
  4 x := x.p;
  5 x.n := y;
  6 y.p := x;
  7 y := nil;
  8 goto 10;
  9 x := nil;}
  
```

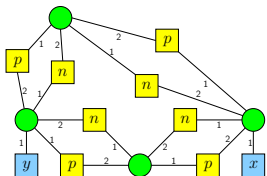
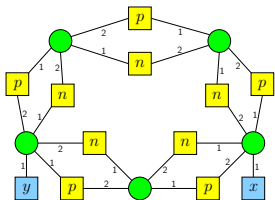
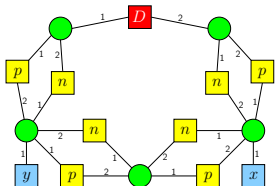



Steps

- 1 assignment
- 2 garbage collection
- 3 concretization
- 4 re-abstraction

HRG

 $D \longrightarrow$


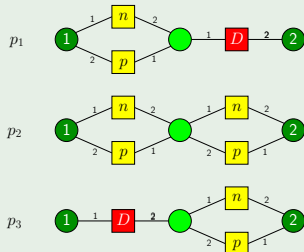


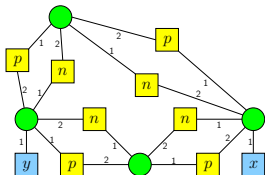
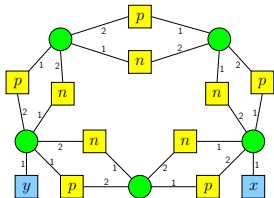
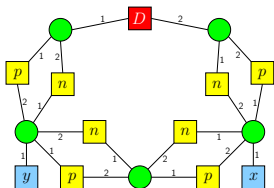
Steps

- 1 assignment
- 2 garbage collection
- 3 concretization
- 4 re-abstraction

HRG

$D \longrightarrow$





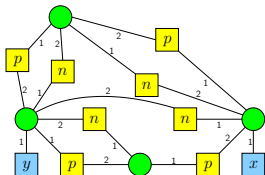
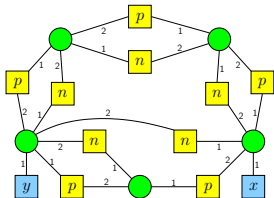
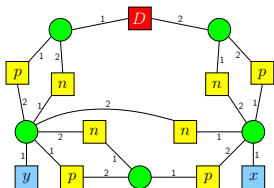
Steps

- ① assignment
- ② garbage collection
- ③ concretization
- ④ re-abstraction

Program

```

delete() {
  1 if x = nil goto 10;
  2 if x = x.n goto 9;
  3 y := x.n;
  4 x := x.p;
  5 x.n := y;
  6 y.p := x;
  7 y := nil;
  8 goto 10;
  9 x := nil;}
  
```



Steps

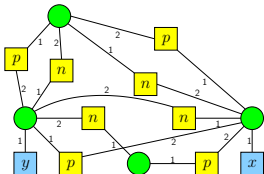
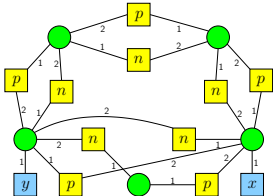
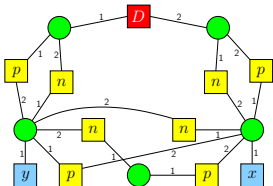
- ① assignment
- ② garbage collection
- ③ concretization
- ④ re-abstraction

Program

```

delete() {
  1 if x = nil goto 10;
  2 if x = x.n goto 9;
  3 y := x.n;
  4 x := x.p;
  5 x.n := y;
  6 y.p := x;
  7 y := nil;
  8 goto 10;
  9 x := nil;}

```



Steps

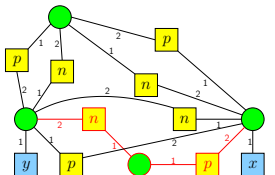
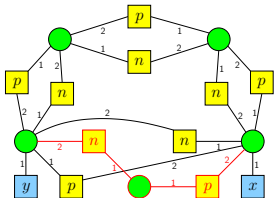
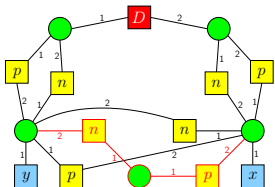
- 1 assignment
- 2 garbage collection
- 3 concretization
- 4 re-abstraction

Program

```

delete() {
  1 if x = nil goto 10;
  2 if x = x.n goto 9;
  3 y := x.n;
  4 x := x.p;
  5 x.n := y;
  6 y.p := x;
  7 y := nil;
  8 goto 10;
  9 x := nil;}

```



Steps

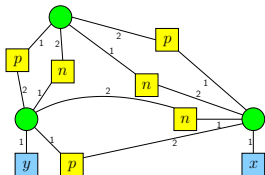
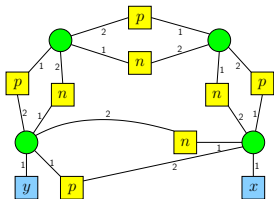
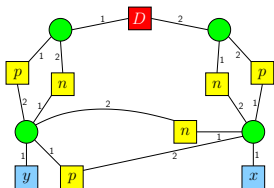
- ① assignment
- ② garbage collection
- ③ concretization
- ④ re-abstraction

Program

```

delete() {
  1 if x = nil goto 10;
  2 if x = x.n goto 9;
  3 y := x.n;
  4 x := x.p;
  5 x.n := y;
  6 y.p := x;
  7 y := nil;
  8 goto 10;
  9 x := nil;}

```



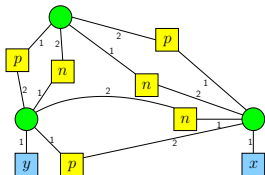
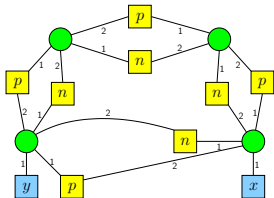
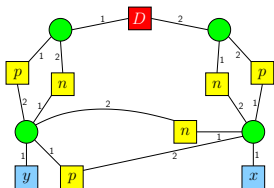
Steps

- ① assignment
- ② garbage collection
- ③ concretization
- ④ re-abstraction

Program

```

delete() {
  1 if x = nil goto 10;
  2 if x = x.n goto 9;
  3 y := x.n;
  4 x := x.p;
  5 x.n := y;
  6 y.p := x;
  7 y := nil;
  8 goto 10;
  9 x := nil;}
  
```



Steps

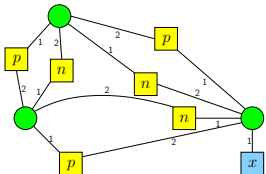
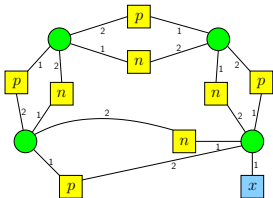
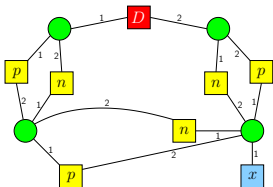
- 1 assignment
- 2 garbage collection
- 3 concretization
- 4 re-abstraction

Program

```

delete() {
  1 if x = nil goto 10;
  2 if x = x.n goto 9;
  3 y := x.n;
  4 x := x.p;
  5 x.n := y;
  6 y.p := x;
  7 y := nil;
  8 goto 10;
  9 x := nil;}

```



Steps

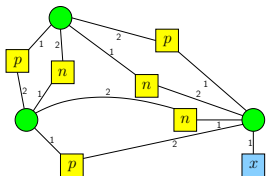
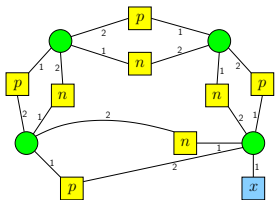
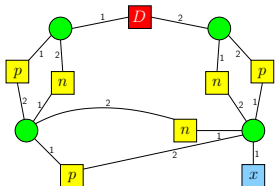
- 1 assignment
- 2 garbage collection
- 3 concretization
- 4 re-abstraction

Program

```

delete() {
  1 if x = nil goto 10;
  2 if x = x.n goto 9;
  3 y := x.n;
  4 x := x.p;
  5 x.n := y;
  6 y.p := x;
  7 y := nil;
  8 goto 10;
  9 x := nil;}

```

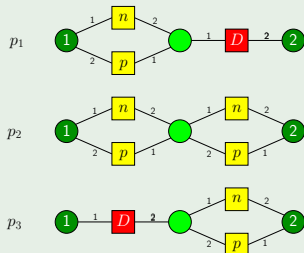


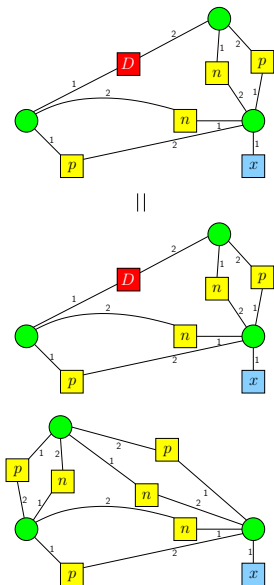
Steps

- 1 assignment
- 2 garbage collection
- 3 concretization
- 4 re-abstraction

HRG

$D \longrightarrow$

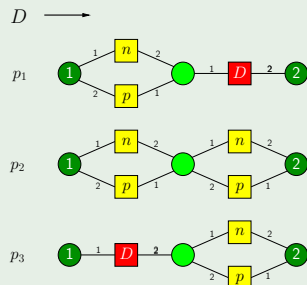




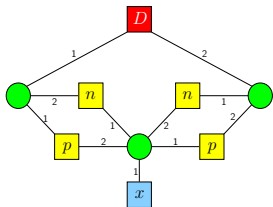
Steps

- 1 assignment
- 2 garbage collection
- 3 concretization
- 4 re-abstraction

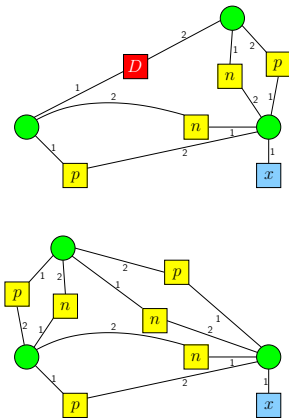
HRG



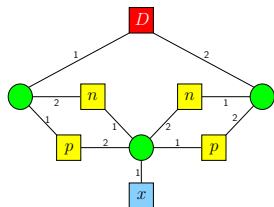
Before



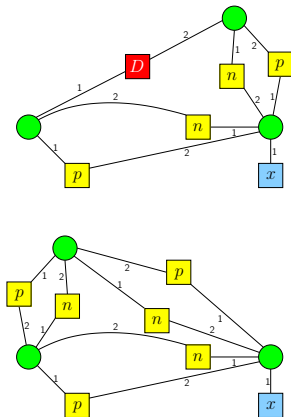
After



Before



After



The result of `delete()` retains the structure of the cyclic list. The result contains 3 nodes if D represented a list with one node.

Conclusion and Future Work

Conclusion

- analysis and verification of **arbitrary** data structures
- **highly parametrized** framework
- more **intuitive**
- easy extension with **concurrency** [Noll and Rieger, 2008]

Conclusion and Future Work

Conclusion

- analysis and verification of **arbitrary** data structures
- **highly parametrized** framework
- more **intuitive**
- easy extension with **concurrency** [Noll and Rieger, 2008]





Outlook

- **implementation**, including model checker
 - almost complete
 - promising results verifying the **DSW tree traversal algorithm** ($19 \cdot 10^6$ states in < 40 min)
- expressive heap **logic**
- **automated inference of HRGs** from data structure definitions

Thank you for your attention!

Related Work

- [Bakewell et al., 2004, Dodds and Plump, 2006]: Graph Reduction Grammars, shape safety, specification of abstract transformation for each operation
- [Rensink, 2004, Rensink and Distefano, 2006]: Model pointer assignments directly by graph transformations (no low-level programming language)
- [Lee et al., 2005]: Supports only trees

-  Bakewell, A., Plump, D., and Runciman, C. (2004).
Specifying pointer structures by graph reduction.
In Applications of Graph Transformations with Industrial Relevance '03, volume 3062 of *LNCS*, pages 30–44. 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 ICALP '92, volume 623 of *LNCS*, pages 138–149. Springer.
-  Lee, O., Yang, H., and Yi, K. (2005).
Automatic verification of pointer programs using
grammar-based shape analysis.
In Proc. of 14th European Symposium on Programming (ESOP '05), volume 3444 of *LNCS*, pages 124–140. Springer.



Noll, T. and Rieger, S. (2008).

Verifying dynamic pointer-manipulating threads.

In *15th International Symposium on Formal Methods (FM '08)*. Springer.



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.*