Heap Abstraction by Graph Reduction with Graph Grammars

Jonathan Heinen Thomas Noll Stefan Rieger

MOVES: Software Modeling and Verification
RWTH Aachen University, Germany

27.08.2007

FR 6.2 Informatik, Universität des Saarlandes, Saarbrücken

Overview

1. Introduction
2. Hypergraph Reduction
3. Heap Configurations
4. Modeling Program Semantics
5. Conclusion

Aim: Abstraction of complex data structures for verification

Problem

- **complex data structures (record structures)**
- creation and destruction of objects at runtime
- destructive updates via pointers
- parallelism
- possibly infinite state space

Aim: Abstraction of complex data structures for verification

Problem

- **complex data structures (record structures)**
- creation and destruction of objects at runtime
- destructive updates via pointers
- parallelism
- possibly infinite state space

Approach

- graph grammars to model data structures
- reverse rule application (graph reduction)
- partial concretization
- finiteness ensured by special sink node

Hypergraphs

Definition (Labeled Hypergraph)

Given: Alphabet Σ, ranking function $rk : \Sigma \to \mathbb{N}$
Define: $H = (V, E, att, \ell, ext)$ where

- set of nodes V, set of edges E

Hypergraphs

Definition (Labeled Hypergraph)

Given: Alphabet Σ, ranking function $rk : \Sigma \to \mathbb{N}$
Define: $H = (V, E, att, \ell, ext)$ where

- set of nodes V, set of edges E
- attachment function $att : E \to V^\star$

Hypergraphs

Definition (Labeled Hypergraph)

Given: Alphabet Σ, ranking function $rk : \Sigma \to \mathbb{N}$

Define: $H = (V, E, att, \ell, ext)$ where

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

Hypergraphs

Definition (Labeled Hypergraph)

Given: Alphabet Σ, ranking function $rk : \Sigma \to \mathbb{N}$
Define: $H = (V, E, att, \ell, ext)$ where

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

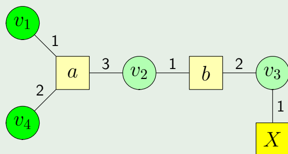
Hypergraphs

Definition (Labeled Hypergraph)

Given: Alphabet Σ, ranking function $rk : \Sigma \to \mathbb{N}$
Define: $H = (V, E, att, \ell, ext)$ where

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

Example



$$
\begin{aligned}
V &= \{v_1, v_2, v_3, v_4\} \\
E &= \{e_1, e_2, e_3\} \\
att &= \{e_1 \mapsto v_1 v_4 v_2, e_2 \mapsto v_2 v_3, e_3 \mapsto v_3\} \\
\ell &= \{e_1 \mapsto a, e_2 \mapsto b, e_3 \mapsto X\} \\
ext &= v_1 v_4
\end{aligned}
$$

Hyperedge Replacement Grammar

Definition

A HRG over Σ is a tuple $G = (N, T, P, S)$ where

- $N \subseteq \Sigma$ is a set of nonterminals,

Hyperedge Replacement Grammar

Definition

A HRG over Σ is a tuple $G = (N, T, P, S)$ where

- $N \subseteq \Sigma$ is a set of nonterminals,
- $T = \Sigma \setminus N$ is a set of terminals,

Hyperedge Replacement Grammar

Definition

A HRG over Σ is a tuple $G = (N, T, P, S)$ where

- $N \subseteq \Sigma$ is a set of nonterminals,
- $T = \Sigma \setminus N$ is a set of terminals,
- P is a set of productions of the form $X \to H$ with $X \in N$ and hypergraph H with $|ext_H| = rk(X)$.

Hyperedge Replacement Grammar

Definition

A HRG over Σ is a tuple $G = (N, T, P, S)$ where

- $N \subseteq \Sigma$ is a set of nonterminals,
- $T = \Sigma \setminus N$ is a set of terminals,
- P is a set of productions of the form $X \rightarrow H$ with $X \in N$ and hypergraph H with $|ext_H| = rk(X)$.
- $S \in N$ is the starting symbol.

Hyperedge Replacement Grammar

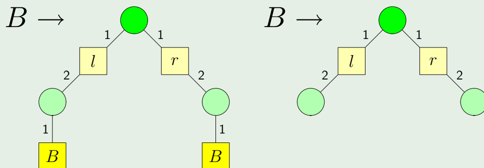
Definition

A HRG over Σ is a tuple $G = (N, T, P, S)$ where

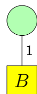
- $N \subseteq \Sigma$ is a set of nonterminals,
- $T = \Sigma \setminus N$ is a set of terminals,
- P is a set of productions of the form $X \to H$ with $X \in N$ and hypergraph H with $|ext_H| = rk(X)$.
- $S \in N$ is the starting symbol.

Example

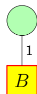
$$N = \{B\}$$
$$T = \{r, l\}$$
$$S = B$$



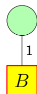
Derivation

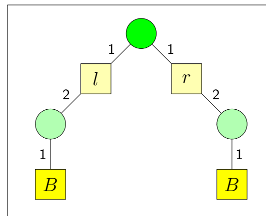


Derivation (Select Nonterminal)

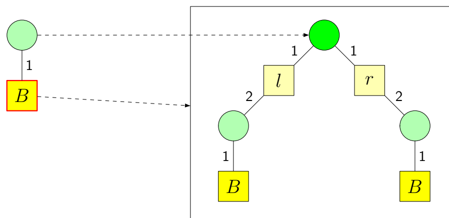


Derivation (Select Rule)

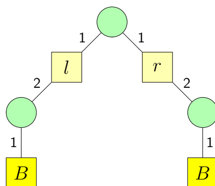




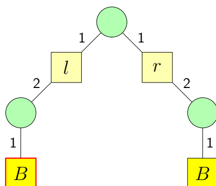
Derivation (Match)



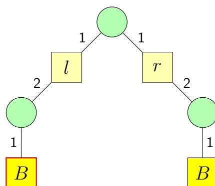
Derivation (Replace)

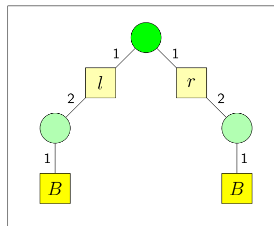


Derivation (Select Nonterminal)

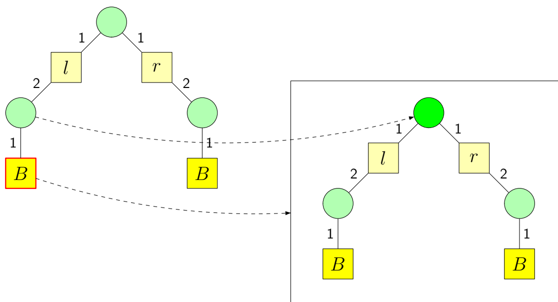


Derivation (Select Rule)

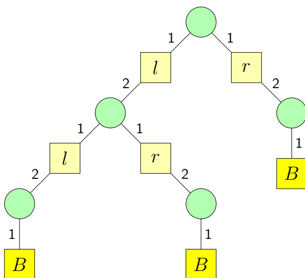




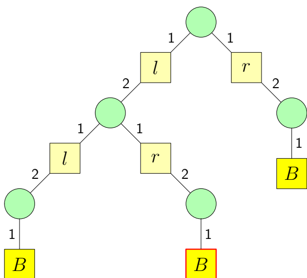
Derivation (Match)



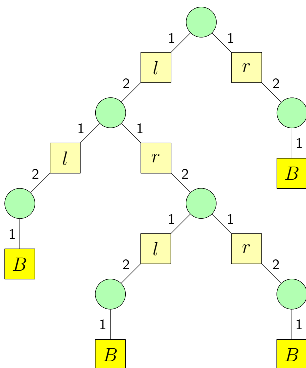
Derivation (Replace)



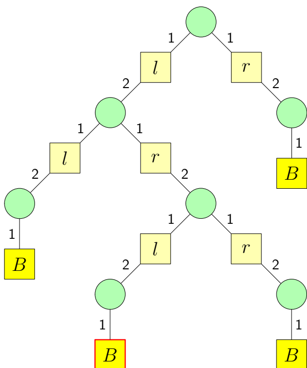
Derivation (Select Nonterminal)



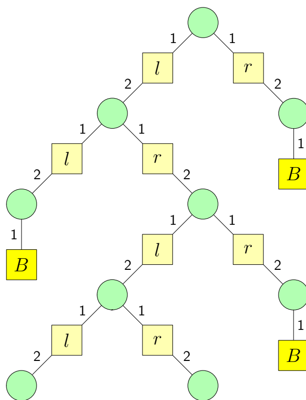
Derivation (Replace)



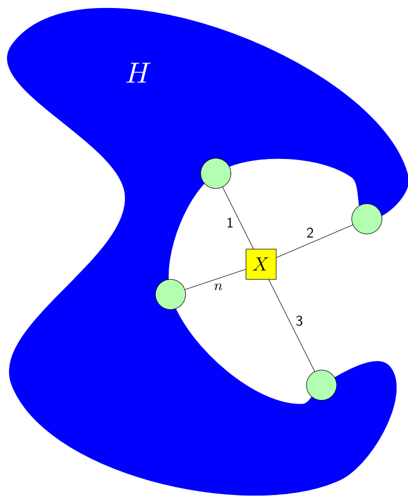
Derivation (Select Nonterminal)



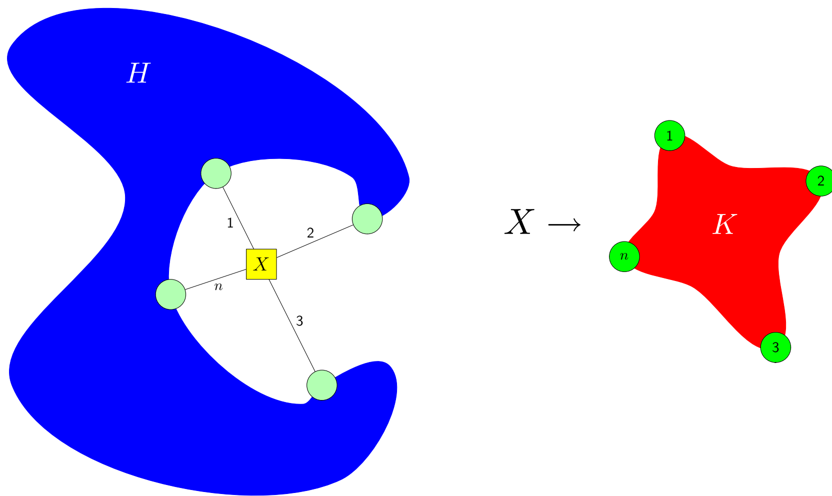
Derivation (Replace with Terminal Rule)



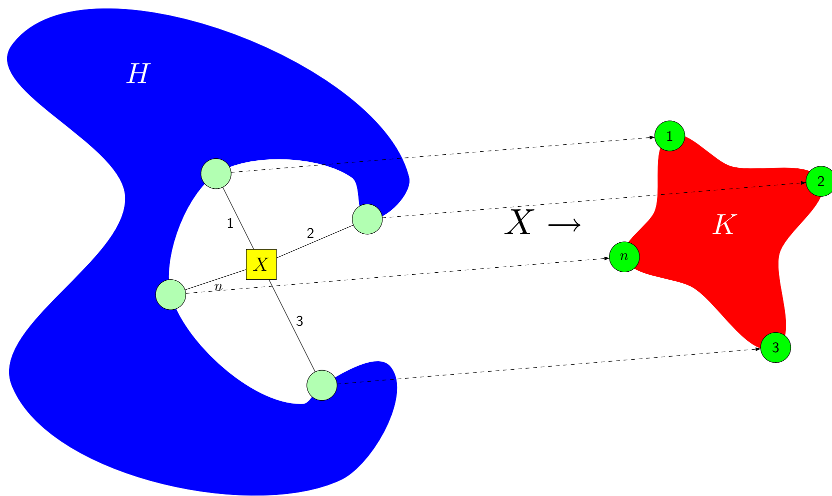
Rule Application (General Case)



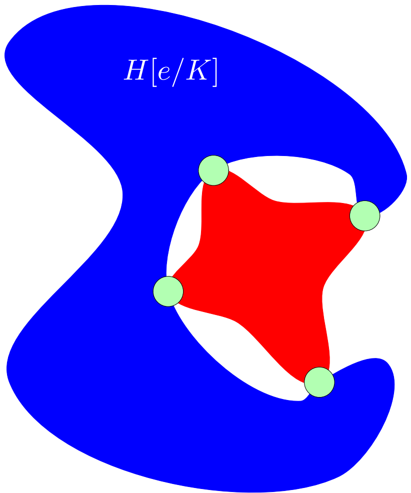
Rule Application (General Case)



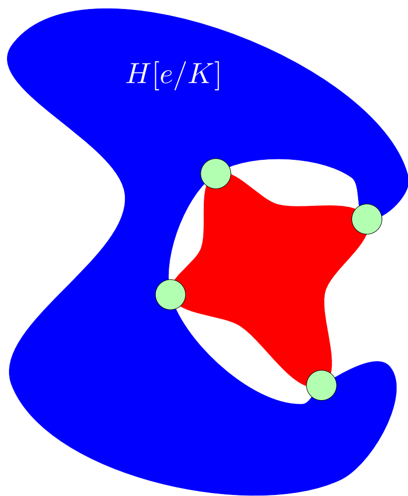
Rule Application (General Case)



Rule Application (General Case)



Rule Application (General Case)



Some Properties

Sequentialization/Parallelization:

$H[e_1/H_1, ..., e_n/H_n] =$
$H[e_1/H_1]...[e_n/H_n]$

Confluence:

$H[e_1/H_1][e_2/H_2] =$
$H[e_2/H_2][e_1/H_1]$

Associativity:

$H[e_1/H_1][e_2/H_2] =$
$H[e_1/H_1[e_2/H_2]]$

Overview

1. Introduction
2. Hypergraph Reduction
3. Heap Configurations
4. Modeling Program Semantics
5. Conclusion

Hypergraph Reduction

Aim: Abstraction

- HRGs for generating graphs
- Here: opposite approach, reduce size of graphs maintaining structural information

$\Rightarrow$ Backward application of grammar rules

Hypergraph Reduction

Aim: Abstraction

- HRGs for generating graphs
- Here: opposite approach, reduce size of graphs maintaining structural information

$\Rightarrow$ Backward application of grammar rules

Problems

- Determining applicable rules
- Finding the redex
- No confluence, reduction might “get stuck” / different results
- No termination guaranteed (depending on grammar)

Hypergraph Embedding

Embedding $\iota : K \rightarrowtail H$

HG K is embedded in HG H if there is $\iota = (\iota_V, \iota_E)$ with

- $\iota_V : V_K \to V_H$ surjective on ext_K, bijective on $V_K \setminus [ext_K]$

Hypergraph Embedding

Embedding $\iota : K \rightarrowtail H$

HG K is embedded in HG H if there is $\iota = (\iota_V, \iota_E)$ with

- $\iota_V : V_K \to V_H$ surjective on ext_K, bijective on $V_K \setminus [ext_K]$
- $\iota_E : E_K \to E_H$ bijective

Hypergraph Embedding

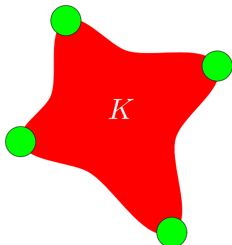
Embedding $\iota : K \rightarrowtail H$

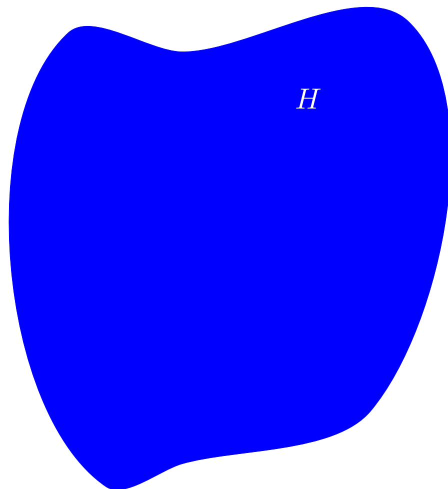
HG K is embedded in HG H if there is $\iota = (\iota_V, \iota_E)$ with

- $\iota_V : V_K \to V_H$ surjective on ext_K, bijective on $V_K \setminus [ext_K]$
- $\iota_E : E_K \to E_H$ bijective
- functions retain ℓ and att
  (labeling & graph structure retained)

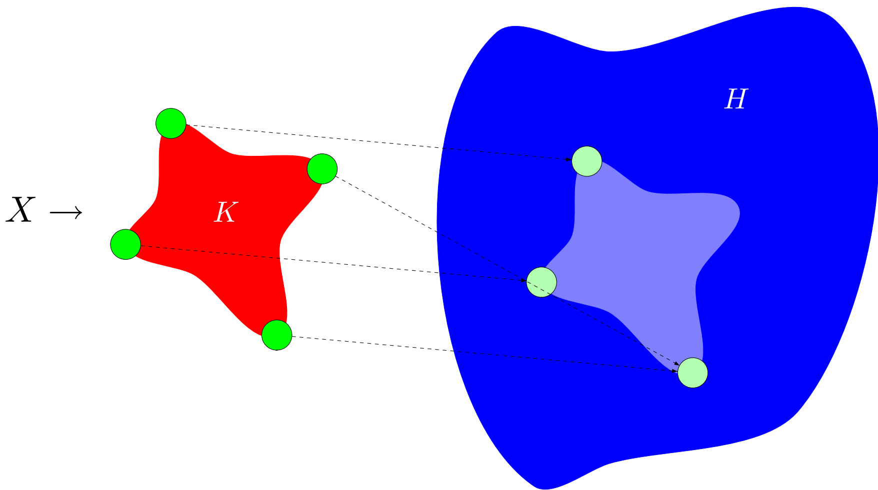
Backward Rule Application

$X \rightarrow$



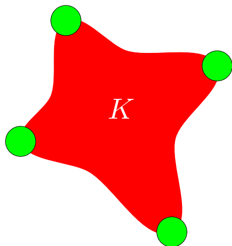


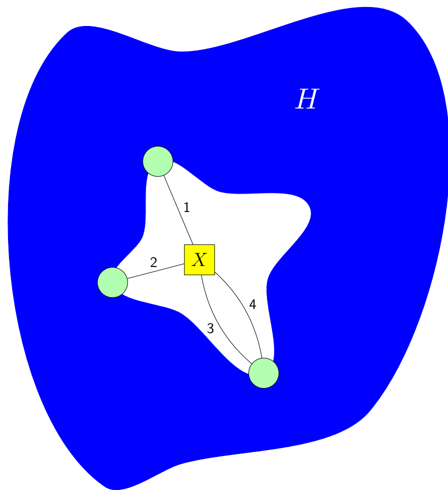
Backward Rule Application (Embedding)



Backward Rule Application (Backward Replace)

$X \rightarrow$





Termination & Matching

Increasing HRG

A HRG $G = (N, T, P, S)$ is increasing iff for all $X \to H \in P$:

$$rk(X) < |V_H| \vee |E_H| > 1$$

Increasingness clearly implies finiteness of all backward derivations.

Termination & Matching

Increasing HRG

A HRG $G = (N, T, P, S)$ is increasing iff for all $X \to H \in P$:

$$rk(X) < |V_H| \vee |E_H| > 1$$

Increasingness clearly implies finiteness of all backward derivations.

How to match and select rules?

Define rule selector as partial function $\sigma(P, H) = (p, \iota)$ where $p = X \to K \in P$ and $\iota : K \rightarrowtail H$ such that $\exists H'$ with $H \xLeftarrow{p,\iota} H'$.

Termination & Matching

Increasing HRG

A HRG $G = (N, T, P, S)$ is increasing iff for all $X \to H \in P$:

$$rk(X) < |V_H| \vee |E_H| > 1$$

Increasingness clearly implies finiteness of all backward derivations.

How to match and select rules?

Define rule selector as partial function $\sigma(P, H) = (p, \iota)$ where $p = X \to K \in P$ and $\iota : K \rightarrowtail H$ such that $\exists H'$ with $H \stackrel{p,\iota}{\Longleftarrow} H'$.

Details? $\Rightarrow$ future work

Overview

1. Introduction
2. Hypergraph Reduction
3. Heap Configurations
4. Modeling Program Semantics
5. Conclusion

Heaps as Hypergraphs

Modeling

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

Heaps as Hypergraphs

Modeling

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

Ensuring Finiteness

- Employ abstraction by (backward) application of HRG rules
- Enforce bound on the number of nodes

$\Rightarrow$ special sink node

Heap Configurations (formally)

Definition

Let G be a HRG. A heap configuration is of the form $\gamma = (H, Var, sink, F)$ with

- hypergraph H
- program variables $Var \subseteq T_G$
- sink node $sink$
- set of flags $F \subseteq \mathrm{Flags}$

Heap Configurations (formally)

Definition

Let G be a HRG. A heap configuration is of the form $\gamma = (H, Var, sink, F)$ with

- hypergraph H
- program variables $Var \subseteq T_G$
- sink node $sink$
- set of flags $F \subseteq \mathrm{Flags}$

Isomorphism

We do **not** distinguish isomorphic heap configurations.

Heap Configurations (formally)

Definition

Let G be a HRG. A heap configuration is of the form $\gamma = (H, Var, sink, F)$ with

- hypergraph H
- program variables $Var \subseteq T_G$
- sink node $sink$
- set of flags $F \subseteq \text{Flags}$

Isomorphism

We do **not** distinguish isomorphic heap configurations.

Boundedness

$\gamma = (H, Var, sink, F)$ is k–bounded if $|V_H| \leq k$.

Enforcing Boundedness

Idea

Summarize nodes in $sink$ if abstraction through HRG does not suffice for size reduction.

Enforcing Boundedness

Idea

Summarize nodes in $sink$ if abstraction through HRG does not suffice for size reduction.

Heap Compactor κ

$\kappa(\gamma, k) = \gamma'$ where $\gamma = (H, ...)$ and $\gamma' = (K, ...)$ s.t.:

- If $|V_H| > k : |V_K| = k$ and ...
- If $|V_H| \leq k : K = H$.

Enforcing Boundedness

Idea

Summarize nodes in $sink$ if abstraction through HRG does not suffice for size reduction.

Heap Compactor κ

$\kappa(\gamma, k) = \gamma'$ where $\gamma = (H, \dots)$ and $\gamma' = (K, \dots)$ s.t.:

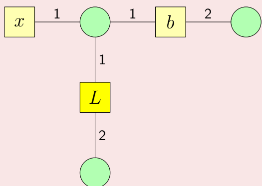
- If $|V_H| > k : |V_K| = k$ and ...
- If $|V_H| \leq k : K = H$.

Strategies

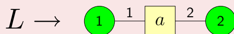
- Summarize preferably connected nodes
- Summarize nodes as far away from variables as possible
- Call heap compactor always after abstraction

Partial Concretization

Problem: What is $x.a$?



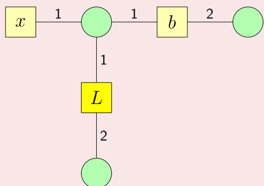
$L \longrightarrow$

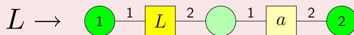
$L \longrightarrow$ 

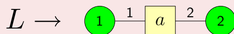
Idea: Apply rules nondeterministically until $x.a$ can be "resolved"

Partial Concretization

Problem: What is $x.a$?



$L \longrightarrow$ 

$L \longrightarrow$ 

Idea: Apply rules nondeterministically until $x.a$ can be "resolved"
$\Rightarrow$ Possibly infite loop (analogous to left recursion by CFGs)

Partial Concretization

Solution

Introduce normal form analogous to Greibach NF for CFGs:

Partial Concretization

Solution

Introduce normal form analogous to Greibach NF for CFGs:

For all rules $X \rightarrow H \in P$ the following holds:

$$\forall e \in E_H : [ext_H] \cap [att_H(e)] \neq \emptyset \Rightarrow \ell(e) \in T$$

Partial Concretization

Solution

Introduce normal form analogous to Greibach NF for CFGs:

For all rules $X \rightarrow H \in P$ the following holds:

$$\forall e \in E_H : [ext_H] \cap [att_H(e)] \neq \emptyset \;\Rightarrow\; \ell(e) \in T$$

Consequence

- Single rule application suffices
- Expressive power equivalent?

Overview

1 Introduction

2 Hypergraph Reduction

3 Heap Configurations

4 Modeling Program Semantics

5 Conclusion

Expressions and Assignments

Multivalued Expression Semantics

- pointer expressions yield a set of nodes, due to *sink* node
- boolean expressions cannot always be evaluated to a single value (due to pointer sem.)

$\Rightarrow$ introduction of nondeterminism

Expressions and Assignments

Multivalued Expression Semantics

- pointer expressions yield a set of nodes, due to *sink* node
- boolean expressions cannot always be evaluated to a single value (due to pointer sem.)

$\Rightarrow$ introduction of nondeterminism

Steps for Execution of Assignments

1. removing flags from heap

Expressions and Assignments

Multivalued Expression Semantics

- pointer expressions yield a set of nodes, due to *sink* node
- boolean expressions cannot always be evaluated to a single value (due to pointer sem.)

$\Rightarrow$ introduction of nondeterminism

Steps for Execution of Assignments

1. removing flags from heap
2. partial concretization, if nonterminal edge in range (nondet.)

Expressions and Assignments

Multivalued Expression Semantics

- pointer expressions yield a set of nodes, due to *sink* node
- boolean expressions cannot always be evaluated to a single value (due to pointer sem.)

$\Rightarrow$ introduction of nondeterminism

Steps for Execution of Assignments

1. removing flags from heap
2. partial concretization, if nonterminal edge in range (nondet.)
3. execution of the actual assignment (nondet.)

Expressions and Assignments

Multivalued Expression Semantics

- pointer expressions yield a set of nodes, due to *sink* node
- boolean expressions cannot always be evaluated to a single value (due to pointer sem.)

⇒ introduction of nondeterminism

Steps for Execution of Assignments

1. removing flags from heap
2. partial concretization, if nonterminal edge in range (nondet.)
3. execution of the actual assignment (nondet.)
4. garbage collection

Expressions and Assignments

Multivalued Expression Semantics

- pointer expressions yield a set of nodes, due to *sink* node
- boolean expressions cannot always be evaluated to a single value (due to pointer sem.)

$\Rightarrow$ introduction of nondeterminism

Steps for Execution of Assignments

1. removing flags from heap
2. partial concretization, if nonterminal edge in range (nondet.)
3. execution of the actual assignment (nondet.)
4. garbage collection
5. re–abstraction using HRG rules backwards

Expressions and Assignments

Multivalued Expression Semantics

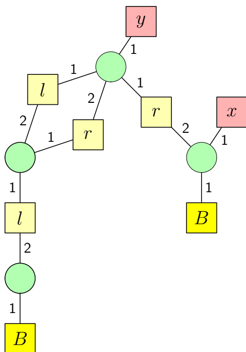
- pointer expressions yield a set of nodes, due to *sink* node
- boolean expressions cannot always be evaluated to a single value (due to pointer sem.)

$\Rightarrow$ introduction of nondeterminism

Steps for Execution of Assignments

1. removing flags from heap
2. partial concretization, if nonterminal edge in range (nondet.)
3. execution of the actual assignment (nondet.)
4. garbage collection
5. re–abstraction using HRG rules backwards
6. call to compactor to obtain k–bounded representation

Example

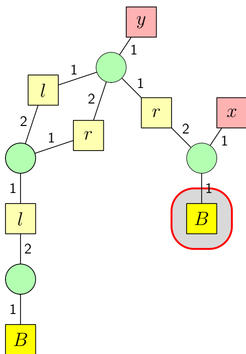


Instructions:

new(x.r);

Steps:

Example



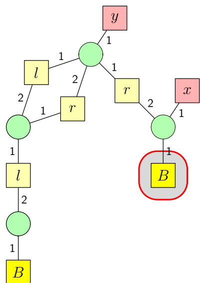
Instructions:

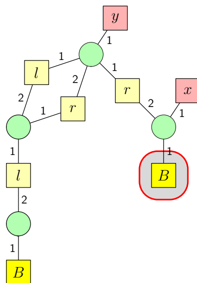
new(x.r);

Steps:

1. Heap Concretizer

Example





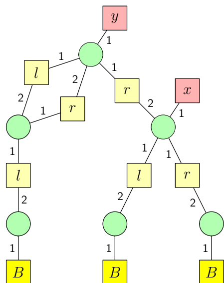
Instructions:

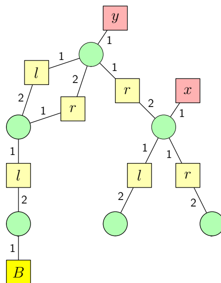
new(x.r);

Steps:

1. Heap Concretizer

Example





Instructions:

new(x.r);

Steps:

1. Heap Concretizer

Example

Instructions:

new(x.r);

Steps:

1. Heap Concretizer
2. Execution

Example

Instructions:

new(x.r);

Steps:

1. Heap Concretizer
2. Execution

Example

Instructions:

new(x.r);

Steps:

1. Heap Concretizer
2. Execution
3. Garbage Collector

Example

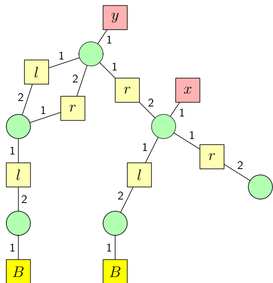
Instructions:

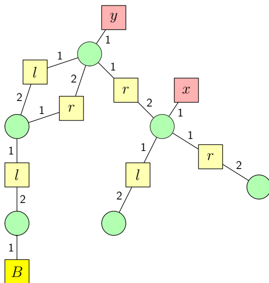
new(x.r);

Steps:

1. Heap Concretizer
2. Execution
3. Garbage Collector

Example





Instructions:

new(x.r);

Steps:

1. Heap Concretizer
2. Execution
3. Garbage Collector
4. Heap Abstractor

Example

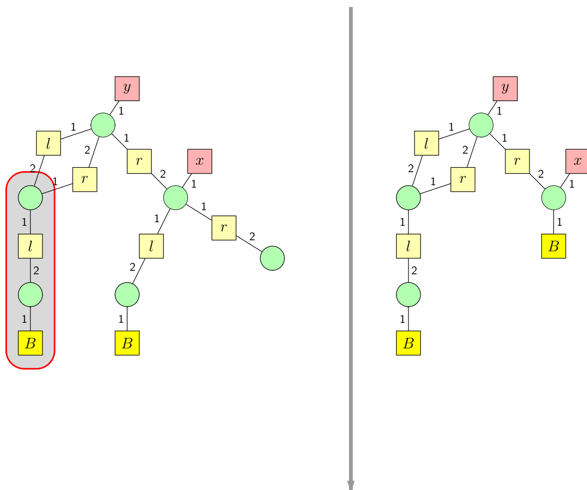
Instructions:

new(x.r);

Steps:

1. Heap Concretizer
2. Execution
3. Garbage Collector
4. Heap Abstractor

Example



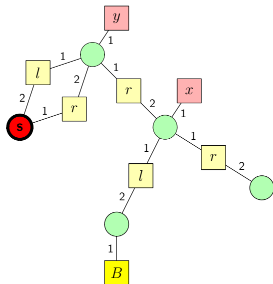
Instructions:

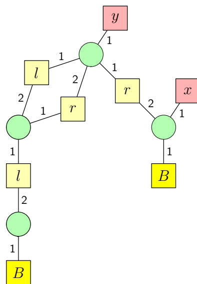
new(x.r);

Steps:

1. Heap Concretizer
2. Execution
3. Garbage Collector
4. Heap Abstractor
5. Heap Compactor

Example





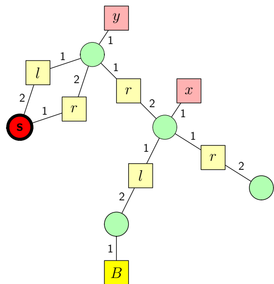
Instructions:

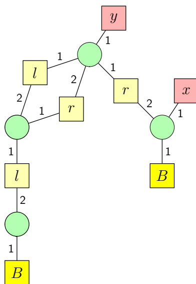
new(x.r);

Steps:

1. Heap Concretizer
2. Execution
3. Garbage Collector
4. Heap Abstractor
5. Heap Compactor

Example





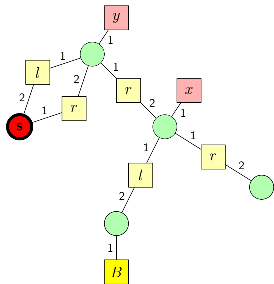
Instructions:

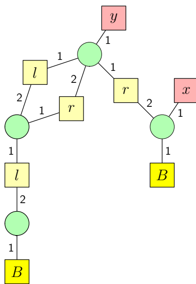
new(x.r);

x = y;

Steps:

Example





Instructions:

new(x.r);

x = y;

Steps:

1. Heap Concretizer

Example

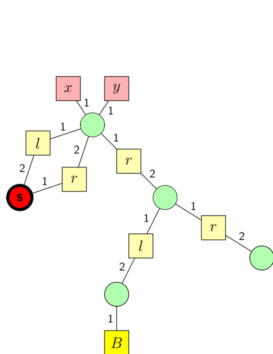
Instructions:

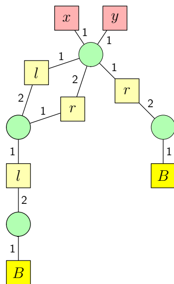
new(x.r);
x = y;

Steps:

1. Heap Concretizer
2. Execution

Example





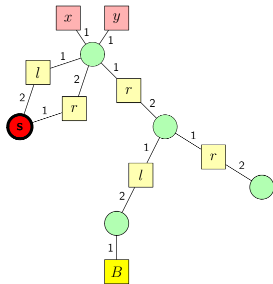
Instructions:

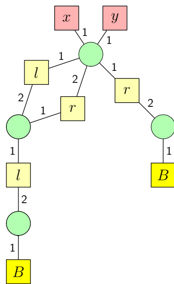
new(x.r);
x = y;

Steps:

1. Heap Concretizer
2. Execution

Example





Instructions:

new(x.r);
x = y;

Steps:

1. Heap Concretizer
2. Execution
3. Garbage Collector
4. Heap Abstractor
5. Heap Compactor

Overview

1. Introduction
2. Hypergraph Reduction
3. Heap Configurations
4. Modeling Program Semantics
5. Conclusion

Summary

- Abstraction framework using graph grammars
- Customization for many data structures possible (and necessary)
- Supports dynamic memory allocation, dynamic threading, destructive updates.

Summary

- Abstraction framework using graph grammars
- Customization for many data structures possible (and necessary)
- Supports dynamic memory allocation, dynamic threading, destructive updates.

To do

- Handling of graph matching
- Rule selection heuristics
- Graph grammars for common data structures
- Tests with examples
- Logic
- Verification framework
- Implementation

Thank you for your attention!