RWTH Aachen

Department of Computer Science
Technical Report

A Local Greibach Normal Form for Hyperedge Replacement Grammars

Christina Jansen, Jonathan Heinen,
Joost-Pieter Katoen and Thomas Noll

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2011-04

RWTH Aachen · Department of Computer Science · January 2011

A Local Greibach Normal Form for Hyperedge Replacement Grammars

Christina Jansen, Jonathan Heinen, Joost-Pieter Katoen, and Thomas Noll

Software Modeling and Verification Group
RWTH Aachen University, Germany
http://moves.rwth-aachen.de/

Abstract. Heap-based data structures play an important role in modern programming concepts. However standard verification algorithms cannot cope with infinite state spaces as induced by these structures. A common approach to solve this problem is to apply abstraction techniques. Hyperedge replacement grammars provide a promising technique for heap abstraction as their production rules can be used to partially abstract and concretise heap structures. To support the required concretisations, we introduce a normal form for hyperedge replacement grammars as a generalisation of the Greibach Normal Form for string grammars and the adapted construction.

1 Introduction

The verification of programs that use pointers to implement dynamic data structures is a highly challenging and important task, as memory leaks or dereferencing null pointers can cause great damage especially when software reliability is at stake. As objects can be created at runtime, dynamic data structures induce a possibly infinite state space and therefore cannot be handled by standard verification algorithms. Abstraction techniques such as shape analysis [18] that yield finite representations for these data structures are a common way to address this problem. Other popular techniques are separation logic [10, 14] (duality with HRGs is observed in [4]) and regular tree automata theory [2].

Our approach is to verify pointer-manipulating programs using *hyperedge replacement grammars* (HRGs) [7]. Dynamic memory allocation and destructive updates are transcribed on hypergraphs representing heaps. Production rules of the HRG reflect employed data structures. Terminal edges model variables and pointers, whereas nonterminal edges represent abstract parts of a heap. Thus, hypergraphs are heap configurations that are partially concrete and partially abstract, such that heap fragments relevant for the current program state are concrete while a finite heap representation is archived. Concretisation of abstract heap fragments is obtained by classical forward grammar rule application, abstraction by backward application. This use of HRGs has been first proposed by us in [15]; tool support and the successful verification of the Deutsch-Schorr-Waite tree traversal algorithm have been reported in [8]. Graph grammars for heap verification have also been advocated in, e.g. [18, 13, 11, 1, 12]. Primarily this yields a rather intuitive and easy-to-grasp heap modelling approach, where no abstract program semantics is needed. In particular, it avoids a (often tedious) formal proof how this relates to a concrete semantics, see e.g. [3]. Pointer statements such as assignments and object creation are realised on concrete subgraphs only. Thus if pointer assignments "move" program variables too close to abstract graph fragments, a local concretisation is carried out. To enable this, our heap abstraction HRGs are required to be in a specific form that is akin to the well-known Greibach normal form (GNF) for string grammars.

In [15] we proposed to use the GNF introduced in [6] for this purpose, restricting manageable data structures to ones where each object is referenced by a bounded number of objects. This paper defines a normal form for HRGs as a generalisation of the original GNF for string grammars. Compared to [6], it allows us to model data structures without restrictions to referencing and in general results in grammars with less and smaller production rules. Furthermore our normal form allows to adapt the well-known GNF transformation algorithm for string to graph grammars. We present the adapted construction and its correctness in Section 3. In Section 2, the above concepts are formalised, we consider a notion of typing for HRGs, and provide all relevant theoretical results.

2 Preliminaries

Given a set S, $S^\star$ is the set of all finite sequences (strings) over S including the empty sequence ε. For $s \in S^\star$, the length of s is denoted by $|s|$, the set of all elements of s is written as $[s]$, and by $s(i)$ we refer to the i-th element of s. Given a tuple $t = (A, B, C, ...)$ we write A_t, B_t etc. for the components if their names are clear from the context. Function $f \restriction S$ is the restriction of f to S. Function $f : A \to B$ is lifted to sets $f : 2^A \to 2^B$ and to sequences $f : A^\star \to B^\star$ by point-wise application. We denote the identity function on a set S by id_S.

2.1 Heaps and Hypergraphs

The principal idea behind our *Juggrnaut* framework [8, 15] is to represent (abstract) heaps as hypergraphs.

Definition 1 (Hypergraph). *Let Σ be a finite ranked alphabet where $rk : \Sigma \to \mathbb{N}$ assigns to each symbol $a \in \Sigma$ its rank $rk(a)$. A* (labelled) hypergraph *over Σ is a tuple $H = (V, E, att, lab, ext)$ where V is a set of vertices and E a set of hyperedges, $att : E \to V^\star$ maps each hyperedge to a sequence of attached vertices, $lab : E \to \Sigma$ is a hyperedge-labelling function, and $ext \in V^\star$ a (possibly empty) sequence of distinct external vertices.*

For $e \in E$, we require $|att(e)| = rk(lab(e))$ and let $rk(e) = rk(lab(e))$. The set of all hypergraphs over Σ is denoted by HG_Σ.

Hypergraphs are graphs with edges as proper objects which are not restricted to connect exactly two vertices. Two hypergraphs are *isomorphic* if they are identical modulo renaming of vertices and hyperedges. We will not distinguish between isomorphic hypergraphs.

To set up an intuitive heap representation by hypergraphs we consider finite ranked alphabets $\Sigma = Var_\Sigma \uplus Sel_\Sigma$, where Var_Σ is a set of variables, each of rank one and Sel_Σ a set of *selectors* each of rank two. We model heaps as hypergraphs over Σ. Objects are represented by vertices, and pointer variables and selectors by edges connected to the corresponding object(s) where selector edges are understood as pointers from the first attached object to the second one. To represent abstract parts of the heap, we use nonterminal edges i.e. with labels from an additional set of *nonterminals* N of arbitrary rank (and we let $\Sigma_N = \Sigma \cup N$). The connections between hyperedges and vertices are called *tentacles*.

Example 1. A typical implementation of a doubly-linked list consists of a sequence of list elements connected by next and previous pointers and an additional list object containing pointers to the *head* and *tail* of the list. We consider an extended implementation where each list element features an additional pointer to the corresponding list object. Fig. 1 depicts a hypergraph representation of a doubly-linked list. The three circles are vertices representing objects on the heap. Tentacles are labeled with their ordinal number. For the sake of readability, selectors (*head* and *tail*) are depicted as directed edges. A variable named *list* referencing the list object is represented as an edge of rank one. The L-labeled box represents a nonterminal edge of rank three indicating an abstracted doubly-linked list between the first and second attached vertex, where each abstracted list element has a pointer to the list object. In Section 2.2 we will see how abstract structures are defined.

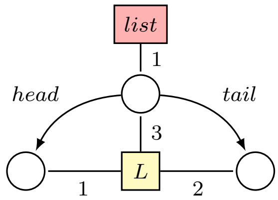


Fig. 1. Heap as hypergraph

Note that not every hypergraph represents a feasible heap: it is necessary that each variable and each a-selector (for every $a \in Sel_\Sigma$) refers to at most one object. Therefore we introduce *heap configurations* as restricted hypergraphs:

Definition 2 (Heap Configuration). *$H \in HG_{\Sigma_N}$ is a* heap configuration *if:*

1. $\forall a \in Sel_\Sigma, v \in V_H : |\{e \in E_H \mid att(e)(1) = v, lab(e) = a\}| \leq 1$, *and*
2. $\forall a \in Var_\Sigma : |\{e \in E_H \mid lab(e) = a\}| \leq 1$

We denote the set of all heap configurations over Σ_N by HC_{Σ_N}. If a heap configuration contains nonterminals it is abstract*, otherwise* concrete*.*

2.2 Data Structures and Hyperedge Replacement Grammars

As pointed out earlier, both abstraction and concretisation are transformation steps on the hypergraph representation of the heap. We use *hyperedge replacement grammars* for this purpose, implementing abstraction and concretisation as backward and forward application of replacement rules respectively.

Definition 3 (Hyperedge Replacement Grammar). *A* hyperedge replacement grammar *(HRG) over an alphabet Σ_N is a set of production rules of the form $X \to H$, with $X \in N$ and $H \in HG_{\Sigma_N}$ where $|ext_H| = rk(X)$. We denote the set of hyperedge replacement grammars over Σ_N by HRG_{Σ_N}.*

Example 2. Fig. 2 specifies an HRG for doubly-linked lists. n, p stand for *next* and *previous* while l is the pointer to the corresponding list object shared by all elements. *head* and *tail* (cf. Fig. 1) do not occur as they are not abstracted.

The HRG derivation steps are defined through hyperedge replacement.

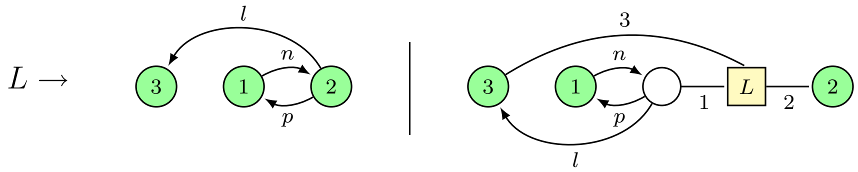


Fig. 2. A grammar for doubly-linked lists

Definition 4 (Hyperedge Replacement). *Let $H, K \in HG_{\Sigma_N}, e \in E_H$ a nonterminal edge with $rk(e) = |ext_K|$. W.l.o.g. we assume that $V_H \cap V_K = E_H \cap E_K = \emptyset$ (otherwise the components in K have to be renamed). The* substitution of e by K, *$H[K/e] = J \in HG_{\Sigma_N}$, is defined by:*

$$V_J = V_H \cup (V_K \setminus [ext_K]) \qquad E_J = (E_H \setminus \{e\}) \cup E_K$$
$$lab_J = (lab_H \upharpoonright (E_H \setminus \{e\})) \cup lab_K \qquad ext_J = ext_H$$
$$att_J = att_H \upharpoonright (E_H \setminus \{e\}) \cup mod \circ att_K$$

with $mod = \mathrm{id}_{V_J} \cup \{[ext_K(1) \mapsto att_H(e)(1), ..., ext_K(rk(e)) \mapsto att_H(e)(rk(e))\}$.

Example 3. Reconsider the hypergraph H of Fig. 1 as well as the second rule in Fig. 2, denoted by $L \rightarrow K$. In H we replace the nonterminal edge e labelled with L by K, which yields $H[K/e]$. This is possible since $rk(L) = |ext_K| = 3$. Replacing the L-edge we merge external node $ext(1)$ with the node connected to the first tentacle, $ext(2)$ with the second and so on. The resulting graph is shown in Fig. 3.

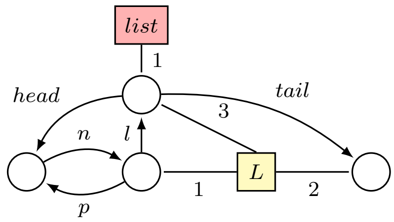


Fig. 3. Hyperedge replacement

Definition 5 (HRG Derivation). *Let $G \in HRG_{\Sigma_N}$, $H, H' \in HG_{\Sigma_N}$, $p = X \rightarrow K \in G$ and $e \in E_H$ with $lab(e) = X$. H* derives H' by p *iff H' is isomorphic to $H[K/e]$. $H \stackrel{e,p}{\Longrightarrow} H'$ refers to this derivation. Let $H \stackrel{G}{\Longrightarrow} H'$ if $H \stackrel{e,p}{\Longrightarrow} H'$ for some $e \in E_H, p \in G$. If G is clear from the context $\Rightarrow^\star$ denotes the reflexive-transitive closure.*

The definition of HRGs does not include a particular starting graph. Instead, it is introduced as a parameter in the definition of the generated language.

Definition 6 (Language of a HRG). *Let $G \in HRG_{\Sigma_N}$ and $H \in HG_{\Sigma_N}$. $L_G(H) = \{K \in HG_\Sigma \mid H \Rightarrow^\star K\}$ is the* language *generated from H using G.*

We write $L(H)$ instead of $L_G(H)$ if G is clear from the context. To define the language of a nonterminal we introduce the notion of a *handle* which is a hypergraph consisting of a single hyperedge attached to external vertices only.

Definition 7 (Handle). *Given $X \in \Sigma$ with $rk(X) = n$, an X-*handle *is the hypergraph $X^\bullet = (\{v_1, ..., v_n\}, \{e\}, [e \mapsto v_1...v_n], [e \mapsto X], v_1...v_n) \in HG_\Sigma$.*

Thus $L(X^\bullet)$ is the language induced by nonterminal X. For $H \in HC_{\Sigma_N}$, $L(H)$ denotes the set of corresponding concrete heap configurations. Note that it is not guaranteed that $L(H) \subseteq HC_\Sigma$, i.e., $L(H)$ can contain invalid heaps.

Definition 8 (Data Structure Grammar). *$G \in HRG_{\Sigma_N}$ is called a* data structure grammar *(DSG) over Σ_N if $\forall X \in N : L(X) \subseteq HC_{Sel_\Sigma}$. We denote the set of all data structure grammars over Σ_N by DSG_{Σ_N}.*

Theorem 1. *It is decidable whether a given HRG is a DSG.*

Later we will see that DSGs are still too permissive for describing heap abstraction and concretisation. In Section 2.4, we will therefore refine this definition to so-called *heap abstraction grammars*.

2.3 Execution of Program Statements

The overall goal of our framework is to reduce the large or even infinite program state spaces induced by dynamic data structures. To this aim, heap configurations are partially abstracted by backward application of replacement rules. As long as pointer manipulations are applied to concrete parts of the heap, they can be realised one-to-one. In order to avoid the need of defining an abstract semantics we avoid manipulations on abstract parts by applying local concretisation steps before. As pointers are not dereferenced backwards, restricting the dereferencing depth to one reduces the (potentially) affected parts of the heap to those nodes that are directly reachable from variable nodes by *outgoing edges*.

Definition 9 (Outgoing Edges). *Let $H \in HC_\Sigma$, $v \in V_H$. The set of* outgoing edges *at vertex v in H is defined as: $out(v) = \{e \in E_H \mid att(e)(1) = v\}$.*

In abstract heap configurations, variable vertices can have abstracted outgoing edges derivable at connected nonterminal tentacles.

Definition 10 (Tentacle). *Let $X \in N_\Sigma$, $i \in [1, rk(X)]$ the pair (X, i) is a tentacle. (X, i) is a* reduction tentacle *if, for all $H \in L(X^\bullet)$, $out(ext_H(i)) = \emptyset$.*

Example 4. Reconsider the grammar of Fig. 2. $(L, 3)$ is a reduction tentacle, as no outgoing terminal edges are derivable at external vertex 3.

A heap configuration is *inadmissible* if variable nodes are connected to non-reduction tentacles.

Definition 11 (Admissibility). *For $H \in HC_{\Sigma_N}$, $e \in E_H$, and $i \in \mathbb{N}$, the pair (e, i) is called a* violation point *if $(lab(e), i)$ is not a reduction tentacle and $\exists e' \in E_H : lab(e') \in Var_\Sigma \wedge att(e')(1) = att(e)(1)$. H is called* admissible *if it contains no violation point, and* inadmissible *otherwise.*

Heap manipulations may introduce violation points. This inadmissibility can be resolved by *concretisation*, that is, by considering all possible replacements of the corresponding edge. Notice that concretisation generally entails nondeterminism, viz. one successor state for each applicable replacement rule.

Example 5. On the left side of Fig. 4, an inadmissible heap configuration is depicted. While its *list* object is only connected to concrete edges and reduction tentacle $(L, 3)$, there is a violation point at the shaded $(L, 1)$-tentacle. Concretisation by applying both production rules of the grammar given in Fig. 2 results in the two admissible configurations on the right.

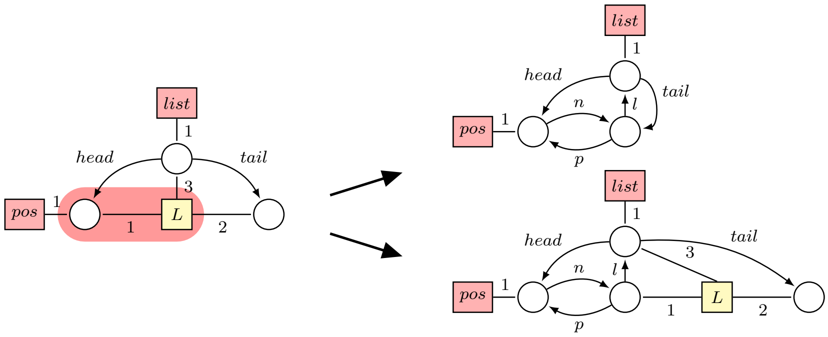


Fig. 4. Concretisation of inadmissible heap configurations

Theorem 2. *For $G \in HRG_{\Sigma_N}, H \in HC_{\Sigma_N}, e \in E_H$, $X \in N$ with $X = lab(e)$:*

$$L(H) = \bigcup_{\forall X \to K \in G} L(H[K/e])$$

This theorem follows directly from the confluence property of HRGs [17].

While concretisation is realised by standard, forward application of production rules, abstraction is handled by *backward* rule application. Thus we call $H \in HC_{\Sigma_N}$ an abstraction of $H' \in HC_{\Sigma_N}$ if $H \Rightarrow^\star H'$. Obviously $L(H') \subseteq L(H)$ and therefore abstraction leads to an over-approximation of the state space. The latter fact together with Theorem 2 yields the soundness of our heap abstraction approach. We apply the principle *"Abstract if possible – Concretise when necessary"* to obtain the best possible results in terms of the size of the resulting state space.

2.4 Heap Abstraction Grammars

As mentioned before, DSGs are not sufficient in our setting. Additional restrictions that ensure termination and correctness of the abstraction technique are listed and discussed in detail below.

Definition 12 (Heap Abstraction Grammar). *$G \in DSG_{\Sigma_N}$ is a* heap abstraction grammar (HAG) *over Σ_N if:*

(1)	*G is productive*	$\forall X \in N : L(X^\bullet) \neq \emptyset$
(2)	*G is increasing*	$\forall X \to H \in G : \lvert E_H \rvert \leq 1 \Rightarrow H \in HG_\Sigma$
(3)	*G is typed*	*see below*
(4)	*G is locally concretisable*	*see below*

We denote the set of all heap abstraction grammars over Σ_N by HAG_{Σ_N}.

Productivity (1) is a well-known notion from string grammars, ensuring that each abstract configuration represents at least one concrete configuration. A rule is *increasing* if its right-hand side is "bigger" than the corresponding handle. *Increasing grammars* (2) guarantee termination of abstraction, as applying rules backwards reduces the size of the heap representation. We call a grammar *typed* (3) if every concrete vertex has a well-defined type as induced by the set of outgoing edges.

Definition 13 (Typedness). *$G \in DSG_{\Sigma_N}$ is* typed *if* $\forall X \in N, i \in [1, rk(X)].$ $\forall H_1, H_2 \in L(X^\bullet) :\ type(X,i) := out_{H_1}(ext_{H_1}(i)) = out_{H_2}(ext_{H_2}(i))$.

As DSGs restrict the number of outgoing edges to a finite set of selectors, every untyped nonterminal can be replaced by a typed one for each derivable type.

Theorem 3. *It is decidable whether a HRG is typed. For any untyped DSG an equivalent typed DSG can be constructed.*

Local concretisability (4) ensures that admissibility of a heap configuration can be established within one concretisation step.

Example 6. Fig. 5(left) reconsiders the original heap configuration given in Fig. 4 with variable *pos* set to the tail of the list. This leads to an inadmissible configuration and two corresponding concretisations, cf. Fig. 5(right). While the first concretisation is an admissible configuration, the second one remains inadmissible. Successive concretisations would lead to further inadmissible configurations. Ignoring the second rule yields termination but is unsound as Theorem 2 requires concretisations by every corresponding rule.

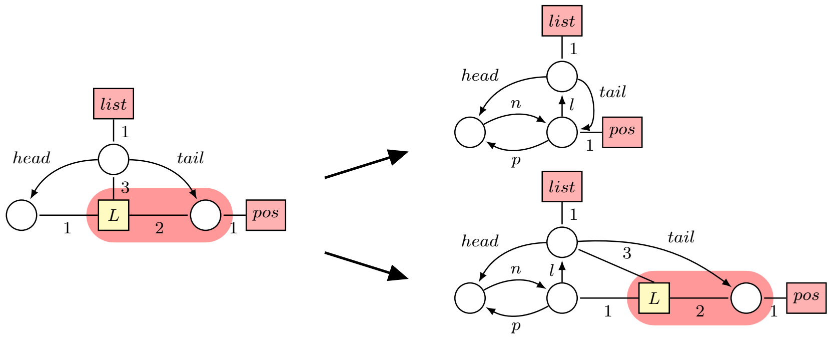


Fig. 5. Inadmissible concretisation

Let $G \in HRG_{\Sigma_N}$ with $p = X \to H \in G$. $(X,i) \to_p (Y,j)$ denotes that (X,i) can be replaced by (Y,j), i.e. $ext_H(i)$ is connected to an (Y,j)-tentacle.

Example 7. For p the second rule of Fig. 2 it holds that $(L,3) \to_p (L,3)$ and $(L,3) \to_p (l,2)$ denoting an incoming l-selector edge.

For simplicity we use G^X as the set of all rules $X \to H \in G$ and $\overline{G^X} = G \backslash G^X$.

Definition 14 (Local Concretisability). *$G \in HRG_{\Sigma_N}$ is locally concretisable if for all $X \in N$ there exist grammars $G_1^X, \cdots, G_{rk(X)}^X \subseteq G^X$ such that:*

1. $\forall i \in [1, rk(X)]$, $L_{G_i^X \cup \overline{G^X}}(X^\bullet) = L_G(X^\bullet)$
2. $\forall i \in [1, rk(X)]$, $a \in type(X,i), p \in G_i^X : (X,i) \to_p (a,1)$

Theorem 4. *For each DSG an equivalent HAG can be constructed.*

Property (1) can be achieved easily by removing non-productive rules, typedness (3) by introducing new, typed nonterminals. The *Local Greibach Normal Form* presented in the next section ensures properties (2) and (4), cf. Theorem 6.

3 Local Greibach Normal Form

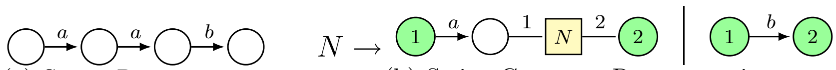
The Greibach Normal Form (GNF) for string grammars restricts production rules to the form $X \rightarrow aN_1 \dots N_k$ such that using left derivation only a word $w \in \Sigma^n N^\star$ is derived after n derivation steps. Thus terminal words are constructed from left to right extending a terminal prefix by one symbol each step.

Up to now there exists no uniform notion of a GNF for graph grammars. In contrast to [6], where graph derivation is from outside to inside, we consider a generalisation of GNF for strings where one-sided derivation is of interest.

Definition 15 (Local Greibach Normal Form). *$G \in DSG_{\Sigma_N}$ is in* local Greibach Normal Form (LGNF) *if for every non-reduction tentacle (X, i) there exists $G_i^X \subseteq G^X$ with:*

1. $L_{G_i^X \cup \overline{G^X}}(X^\bullet) = L_G(X^\bullet)$
2. *$\forall\, p \in G_i^X$: $(X, i) \rightarrow_p (Y, j)$ implies $Y \in \Sigma$ or (Y, i) is a reduction tentacle.*

Example 8. Strings can be uniquely represented by HGs containing chains of terminal edges only, production rules can be translated to HRGs analogously [7]. In Fig. 6, graph representations for word $w = aab$ and string grammar $N \rightarrow aN \mid b$ are given. As nonterminals are of rank two and $(X, 2)$ is a reduction tentacle for each nonterminal X exactly one G_i^X remains namely G_1^X containing the string GNF rules.



(a) String Representation (b) String Grammar Representation

Fig. 6. String Graphs

The LGNF for DSG G is established by merging corresponding sets G_i^X, constructed in four steps along the lines of the GNF construction for string grammars: Assume a total order on the non-reduction tentacles $T_1, \dots, T_n$ over N. Starting at the lowest tentacle, (1) every rule p implicating $T_i \rightarrow_p T_j$ with $j < i$ is eliminated, then (2) local recursion is removed. In a next step (3) all rules are brought into LGNF using simple hyperedge replacements. Finally (4) rules for nonterminals introduced during the construction are transformed. In the following we guide through the four construction steps and define them in detail.

For each non-reduction (X, i)-tentacle we initialise the set $G_i^X = G^X$ and define an ordinal inducing an ordering $T_1, \dots, T_n$ on non-reduction tentacles T_i.

Step 1: Elimination of rules. We first eliminate the rules $p = X \rightarrow H \in G_i^X$ with $(X, i) = T_k \rightarrow_p T_l$, $l < k$. Let $T_l = (Y, j)$, $e \in E_H$ with $lab(e) = Y$ and $att(e)(j) = ext(i)$. Then p is replaced by the set $\{X \rightarrow H[K/e] \mid Y \rightarrow K \in G_j^Y\}$. Theorem 2 states that this procedure does not change the language.

Lemma 1. *Let $G \in HRG_{\Sigma_N}$. For a grammar G' originating from G by eliminating a production rule, it holds that $L_G(H) = L_{G'}(H)$, $\forall H \in HG_{\Sigma_N}$.*

Note that G_j^Y does not contain any rule p with $T_l \rightarrow_p T_m$, $m < l$, as they are removed before. Thus after finitely many steps all corresponding rules are eliminated.

Step 2: Elimination of local recursion. After step 1, rules p with $T_i \rightarrow_p T_i$ remain.

Definition 16 (Local Recursion). *Let* $G \in \mathrm{HRG}_{\Sigma_N}$*,* $X \in N$*,* $i \in [1, rk(X)]$*.* G *is* locally recursive *at* (X, i) *if there exists an rule* p *with* $(X, i) \rightarrow_p (X, i)$*.*

Let $G_r^X \subseteq G_i^X$ be the set of all rules locally recursive at (X, i). To remove local recursion in $p = X \rightarrow H \in G_i^X$ we introduce a new nonterminal B'_j, a recursive rule $B'_j \rightarrow J_n$ and an exit rule $B'_j \rightarrow J_t$. J_t corresponds to graph H, where edge e causing local recursion is removed. We also remove all external nodes singly connected to e ($V_R = \{v \in [ext_H] \mid \forall e' \in E_H : v \in [att_H(e')] \Rightarrow e = e'\}$). By removing border-edge e, its previously connected internal nodes move to the border and get external. Thus $V_{ext} = ([att_{H_j}(e)] \cup [ext_{H_j}]) \cap V_{J_t}$ is the set of arbitrary ordered external nodes.

J_n extends J_t by an additional edge e' labelled by B'_j. As this edge models the structure from the other side, it is connected to the remaining external nodes of H that will not be external any longer. Note that the rank of B'_j is already given by J_t and therefore introduced gaps in the external sequence are filled by new external nodes that are connected to edge e' ($\mathit{fill}(i) = ext_{J_t}(i)$ *if* $ext_{J_t}(i) \in V_H$*, a new node otherwise*). To build up the same structure as (X, i) "from the other side" edge e' has to be plugged in correctly:

$$plug(g) = \begin{cases} ext_{H_j}(y) & \text{if } ext_{J_n}(g) \in V_{J_t} \\ ext_{J_t}(g) & \text{otherwise} \end{cases}, \text{ with } att_{H_j}(e)(y) = ext_{J_t}(g).$$

$B'_j \rightarrow J_t$ with:

$$\begin{aligned} V_{J_t} &= V_{H_j} \setminus V_R(e) \\ E_{J_t} &= E_{H_j} \setminus \{e\} \\ lab_{J_t} &= lab_{H_j} \upharpoonright E_{J_t} \\ att_{J_t} &= att_{H_j} \upharpoonright E_{J_t} \\ ext_{J_t} &\in V_{ext}^* \end{aligned}$$

$B'_j \rightarrow J_n$ with:

$$\begin{aligned} V_{J_n} &= V_{J_t} \cup [ext_{J_n}] \\ E_{J_n} &= E_{J_t} \cup \{e'\} \\ lab_{J_n} &= lab_{J_t} \cup \{e' \rightarrow B'_j\} \\ att_{J_n} &= att_{J_t} \cup \{e' \rightarrow plug\} \\ ext_{J_n} &= \mathit{fill} \end{aligned}$$

Newly introduced nonterminals are collected in a set N', $\Sigma' = \Sigma \cup N'$. For mirrored derivations, each terminal rule in G_i^X can be the initial one thus we add a copy, extended by an additional B'_j-edge, the G_i^X.

Lemma 2. *Let* $G \in DSG_{\Sigma_N}$*. For the grammar* G_i^X *over* $\Sigma' = \Sigma \cup \{B' \mid B'$ *newly introduced nonterminal*$\}$ *originating from* G *by eliminating the* (X, i)*-local recursion as described above, it holds that* $L_G(H) = L_{G_i^X}(H), \forall H \in HG_{\Sigma_N}$*.*

Example 9. The doubly-linked list HRG with production rules $L \rightarrow H \mid J$ given in Fig. 2 is locally recursive at $(L, 2)$. We introduce nonterminal B' and the rules $B' \rightarrow J_t \mid J_n$, cf. Fig. 7. The terminal J_t corresponds to J with removed L-edge and attached external node $ext(2)$. J_n is a copy of J_t with an additional B'-edge and replaced external node $ext(1)$. Intuitively, local recursion is eliminated by introducing new production rules which allow "mirrored" derivations.

Step 3: Generation of Greibach rules. Starting at the highest order tentacle, for each G_i^X LGNF can be established by elimination of every non-reduction (Y,j)-tentacle connected to external node i. That is because (Y,j) is of higher order and thus already in LGNF.

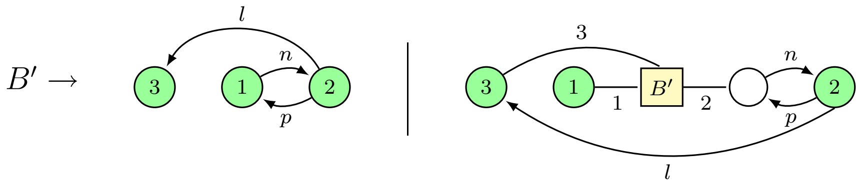


Fig. 7. Doubly-Linked Lists: $G_2^{B'}$

Step 4: Transforming new nonterminals to GNF . In the final step we apply steps one to three to the newly added nonterminals from step two. Obviously further nonterminals could be introduced. To avoid nontermination we merge nonterminals if the right-hand sides of the corresponding production rules are equal.

Theorem 5. *After finite many steps a nonterminal can be merged, thus the construction of LGNF terminates.*

Note that step 2 is nondeterministic as the order on external nodes can be chosen arbitrary. Unnecessary steps introduced by unsuitable orders can be avoided by considering permutations of external nodes isomorphic. If we reach an isomorphic nonterminal after arbitrary many steps all nonterminals in between represent the same language and thus can be merged as long as the rank of the nonterminals permit this.

Example 10. Applying steps 1 to 3 to $G_1^{B'}$ results in a new nonterminal B'' isomorphic to L. Thus we can merge L with B'' and even B' as the latter occurred between the formers.

Theorem 6. *Any DSG can be transformed into an equivalent DSG in LGNF.*

Note that LGNF directly implies the local concretisable property of HAGs. It additionally ensures the increasingness property, as every production rules belongs to at least one G_i^X composed by rules with terminal edges at external node $ext(i)$.

Lemma 3. *Each DSG in LGNF is increasing.*

While we restrict the normalisable grammars to DSGs here the procedure can easily be lifted to arbitrary bounded HRGs.

4 Related Work

The idea of using HRGs for verifying heap manipulating programs was proposed in [8, 15]. No technique for transforming a given HRG into a suitable grammar was provided though, instead using the GNF construction from [6] was proposed, allowing hypergraphs to be concretised from outer to inner. This generally resultsg in more and larger rules compared to our LGNF approach, as LGNF generalises GNF, i.e. every resulting grammar from [6] is in LGNF. Considering the example grammar for binary trees with linked frontier [6], its GNF consists of 135 production rules whereas our local Greibach construction results in 36 rules. In number of nodes and edges our largest rule is half the size of the normalised example rule given in [6]. Note that [6] restricts the input grammars to bounded ones, i.e. those allowing only boundedly many references to each object, excluding for instance rooted grammars like the one given in Fig. 2. Adapting the construction to HAGs without additional restrictions lead to a complex construction with poor results.

A further GNF approach for HRGs can be found in [5]. The basic idea is to use the string GNF construction on a linearisation of the considered HRG. It is however not clear how to re-obtain the HRG from the resulting linearisation. Further normal form constructions addressing node replacement can be found in [16] and [9].

5 Conclusion

This paper presented the theoretical underpinnings of heap abstraction using hyperedge replacement grammars (HRGs). We showed that concretisation and abstraction are naturally obtained by forward and backward rule application respectively. The main contribution is a Greibach normal form (GNF) together with a procedure to transform an HRG into (local) GNF.

Future work will concentrate on advancing our prototypical tool [8], incremental LGNF construction, and on the automated synthesis of heap abstraction grammars from program executions.

References

1. Bakewell, A., Plump, D., Runciman, C., Checking the Shape Safety of Pointer Manipulations. In: RelMiCS ’03, LNCS vol. 3051, pp. 48–61 (2004).
2. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T., Abstract Regular Tree Model Checking. In: ENTCS, vol. 149, pp. 37–48 (2006).
3. Distefano, D., Katoen, J.-P., Rensink, A., Safety and Liveness in Concurrent Pointer Programs. In. FMCO ’05, pp. 280–312 (2005).
4. Dodds, M., From Separation Logic to Hyperedge Replacement and Back. In: ICGT ’08, LNCS vol 5214, pp.484–486 (2008).
5. Dumitrescu, S., Several Aspects of Context Freeness for Hyperedge Replacement Grammars. In: W. Trans. on Comp., vol. 7, pp. 1594-1604, (2008).
6. Engelfriet, J., A Greibach Normal Form for Context-free Graph Grammars. In: ICALP ’92, LNCS vol. 623, pp. 138–149 (1992).
7. Habel, A., Hyperedge Replacement: Grammars and Languages. Springer-Verlag New York, Inc. (1992).
8. Heinen, J., Noll, T., Rieger, S., Juggrnaut: Graph Grammar Abstraction for Unbounded Heap Structures. In: TTSS ’09 (to be published in ENTCS).
9. Klempien-Hinrichs, R., Normal Forms for Context-Free Node-Rewriting Hypergraph Grammars. In: Math. Structures in Comp. Sci., vol. 12, pp. 135-148, (2002).
10. O’Hearn, P.W., Hongseok, Y., Reynolds, J.C., Separation and Information Hiding. In: POPL ’04, vol. 39, pp. 268–280, ACM Press (2004)
11. Rensink, A., Canonical Graph Shapes. In: ESOP ’04, LNCS vol. 2986, pp. 401–415 (2004).
12. Rensink, A., Distefano D., Abstract Graph Transformation. In: SVV ’05, ENTCS vol. 157, pp 39-59 (2006).
13. Rensink, A., Summary from the Outside In. In: AGTIVE’03, LNCS vol. 3062, pp. 486–488 (2004).
14. Reynolds, J.C., Separation Logic: A Logic for Shared Mutable Data Structures. In: LICS ’02, pp. 55–74, IEEE CS Press (2002).
15. Rieger, S., Noll, T., Abstracting Complex Data Structures by Hyperedge Replacement. In: ICGT ’08, LNCS vol. 5214, pp. 69-83 (2008).
16. Rozenberg, G., Welzl, E., Boundary NLC Graph Grammars-Basic Definitions, Normal Forms, and Complexity. In: Inf. Control, vol. 69, pp. 136-167 (1986).

17. Rozenberg, G., Handbook of Graph Grammars and Computing by Graph Transformation: vol. I. Foundations, World Scientific Publishing Co., Inc. (1997).
18. Sagiv, M., Reps, T., Wilhelm, R., Parametric Shape Analysis via 3-Valued Logic. In: ACM Trans. Program. Lang. Syst., vol. 24, pp. 217-298 (2002).

A Decidability of DSG

Theorem 1 in section 2 states that it is decidable whether a grammar $G \in HRG_{\Sigma_N}$ is a DSG. We will proof the decidability for productive grammars. For completeness we start with a procedure to transform an arbitrary HRG into an equivalent productive one.

A nonterminal $X \in N$ is non-productive iff $L(N^\bullet) = \emptyset$. For a grammar G we collect productive nonterminals in the set *productive*. We initialise the set with all nonterminals $X \in N$, where at least one X-rules has a terminal graph as right-hand side, i.e. $X \to H$, with $H \in HG_\Sigma$. Nonterminals X are added iteratively if there is a rule $X \to H$ such that all nonterminals occuring in H are productive, i.e. if $\{lab(e) \,|\forall e \in E_H : lab(e) \in N\} \subseteq \mathit{productive}$ holds. Since the set of nonterminals is finite this procedure will terminate eventually. We then remove the non-productive nonterminals from the alphabet and any rule containing at least one of them from the grammar. This will not change the language of G as each of these removed rules yields hypergraphs containing the non-productive nonterminal Y and therefore cannot be applied to derive a terminal graph. The resulting Grammar is productive.

To check if a productive grammar $G \in \mathrm{HRG}_{\Sigma_N}$ is a DSG_{Σ_N} we check for each rule $X \to H \in G$ and node $v \in V_H$ if v is connected to a variable, i.e.:

$$\exists e \in E_H, lab(e) \in \mathit{Var}_\Sigma, att(e)(1) = v, \tag{1}$$

or if an outgoing selector can be derived more then once at v, i.e.:

$$\begin{aligned}\exists e_1, e_2 \in E_H : lab(e_1) = X \wedge lab(e_2) = Y \wedge att(e_1)(i) = att(e_2)(j) = v \\ \wedge term((X,i)) \cap term((Y,j)) \neq \emptyset.\end{aligned} \tag{2}$$

where $term(T_i)$ recursively determines the set of derivable selectors for each tentacle $T_i = (X, i)$:

$$term((X,i)) = \begin{cases} \{X\} & \text{if } X \in \Sigma \wedge i = 1 \\ \emptyset & \text{if } X \in \Sigma \wedge i = 2 \\ \bigcup\limits_{\forall T_j : (X,i) \to_p T_j} term(T_j) & \text{otherwise} \end{cases}$$

As the sets $term(T_i) \subseteq \Sigma$ are bounded the sets can be calculated incrementally. If none of the properties (1) and (2) holds G is a DSG, otherwise it is not.

B Typedness for DSGs

We split Theorem 3 into two lemma (Lemma 5, 6) and proof them seperately. For simplicity we introduce the notion of typedness on hypergraphs and relate them to the notion of typedness of HRGs introduced in Definition 13.

Definition 17 (Type of a Hypergraph). *Let $H \in \mathrm{HG}_{\Sigma_N}$, $i \in [1, |ext_H|]$ the type-sequence of H, $type(H) = type(H)(1) \dots type(H)(|ext_H|)$ is defined by: $type(H)(i) = out_H(ext_H(i))$.*

The set of types of a nonterminal is defined as $types(N) = \{type(H) \mid H \in L(N)\}$.

Lemma 4. *A grammar $G \in \mathrm{DSG}_{\Sigma_N}$ is typed iff for any nonterminal $X \in N$ it holds that $|types(N)| = 1$.*

Lemma 4 directly holds by definition of typedness.

Lemma 5. *It is decidable whether a HRG is typed.*

Proof. We can check if a DSG G is typed by checking if for each $X \in N$, $i \in [1, rk(X)]$ and any pair of rules $X \to H_1 \mid H_2$ the set of concretisable outgoing edges is equal:

$$\bigcup_{\forall T_i \in Tent_{H_1}(ext_{H_1}(i))} term(T_i) = \bigcup_{\forall T_i \in Tent_{H_2}(ext_{H_1}(i))} term(T_i), \tag{3}$$

where $Tent_H(v) = \{(Y, i) \,|\, \forall Y \in N, i \in [1, rk(Y)], \exists e \in E_H : lab_H(e) = Y \wedge att_H(e)(j) = v\}$ is the set of tentacles connected to node v. If property (3) holds for each $X \in N$, $i \in [1, rk(X)]$ and any pair of rules $X \to H_1 \mid H_2$, G is typed, otherwise it is not.

Lemma 6. *For any untyped DSG G a typed DSG G' can be constructed, where the languages of G and G' are equivalent, i.e. $\bigcup_{X \in N} L_G(X^\bullet) = \bigcup_{X \in N'} L'_G(X^\bullet)$.*

Proof. We transform a given grammar $G \in \mathrm{DSG}_{\Sigma_N}$ into an equivalent typed grammar $G' \in \mathrm{DSG}_{\Sigma_{N'}}$ with $N' = \{(X, T) \in N \times \mathcal{P}(\Sigma) \mid T \in types(X)\}$ with $rk((X, T)) = rk(X)$, by replacing every nonterminal with a set of corresponding typed ones. We first generate a set of right-hand side graphs with replaced nonterminals for each nonterminal $X \in N$: $R^X = \{t(H) | \; X \to H \in G\}$, with $t(H)$ the set of graphs derived from H by replacing old nonterminals from N by new ones from N':

$$t(h) = \{H[e_1/X_1{}^\bullet] \dots [e_k/X_k{}^\bullet] \mid \{e_1 \dots e_k\} = E^N, \forall X_i = (lab(e_i), T) \in N'\},$$
$$T \in types(lab(e_i))),$$

where $E^N = \{e \in E_H \mid lab(e) \in N\}$ is the set of edges labelled with nonterminals. The grammar G' is defined as follows:

$$G' = \{(X, T) \to H \mid X \in N' \wedge H \in R^X : type(H) = T\}$$

The definition of G' obviously implies typedness.

It remains to show, that the languages of G and G' are equivalent, i.e

$$\forall H \in \mathrm{HG}_\Sigma : \exists X \in N : X^\bullet \Rightarrow^* H \Longleftrightarrow \exists X \in N' : ({}^\bullet X) \Rightarrow^* H$$

"$\Longleftarrow$": Let $(X,T)^\bullet[e_1/K_1]\ldots[e_n/K_n] = H$ a derivation for a given HG $H \in \mathrm{HG}_\Sigma$ in G'. We obtain the derivation $X[e'_1/K'_1]\ldots[e'_n/K'_n] = H$ in G by simply replacing any occurring edge labelled by a nonterminal $(X,T) \notin N$ by the corresponding untyped nonterminal, i.e. $K'_i = (V_{K_i}, E_{K_i}, lab_{K'_i}, att_{K_i}, ext_{K_i})$, where $lab_{K'_i}(e) = lab_{K_i}(e)$ if $lab_{K_i}(e) \in N$ and $lab_{K'_i}(e) = X$ if $lab_{K_i}(e) = (X,T)$. The resulting replacement sequence is valid, since typed edges and hypergraphs are of the same rank as the corresponding untyped ones. Furthermore the replacement sequence is a valid derivation in G, as each rule $(X,T) \to t(H) \in G'$ is a copy from a rule $X \to H \in G$ except from nonterminal relabelling.

"$\Longrightarrow$": Starting from an arbitrary derivation $X^\bullet[e_1/K_1]\ldots[e_n/K_n] = H$ in G we construct a corresponding derivation $(X,T)^\bullet[e'_1/K'_1]\ldots[e'_n/K'_n] = H$ in G'. Analogous to above we use graphs $K'_i = (V_{K_i}, E_{K_i}, lab_{K'_i}, att_{K_i}, ext_{K_i})$ but here we have to determine a suitable type for each nonterminal edge. Note that within the derivation each introduced nonterminal edge $e'_j \in E^N_{K'_i}$ will be replaced lateron as the resulting hypergraph H is a terminal graph, i.e. $H \in HG_\Sigma$. We will give e'_j an appropriate label $(lab_{K_i}(e_j), type(K'_j))$ such that the j-th replacement can be realised properly, i.e. $lab_{K'_i}(e'_j) = (lab_{K_i}(e), type(K'_j))$, thus all nonterminal edges $e \in K'_i$ are labelled with corresponding typed nonterminals. The types of all K'_i are determined from right to left, starting with the terminal graph K'_n. As each rule in G has a copy in G' for any combination of edge labels extended by types and as $lab(e_i) \to K_i \in G$ there is also a rule $lab(e_i) \to K_i$ for each $i \in [1\ldots n]$.

C Local Greibach Normal Form

The most demanding part of the Greibach Normal Form construction is the elimination of local recursion. New nonterminals and production rules are generated during this process, while the language must not change.

We show the following statement implying Lemma 2.

Lemma 7. *Let $G \in DSG_{\Sigma_N}$. For the grammar G^X_i over $\Sigma' = \Sigma \cup \{B' \mid B'$ newly introduced nonterminal$\}$ originating from G by eliminating the (X,i)-local recursion, it holds that $\forall H \in HG_{\Sigma_N}, K \in HG_\Sigma : (H \overset{G}{\Rightarrow} K) \Leftrightarrow (H \overset{G^X_i}{\Rightarrow} K)$.*

Proof. Note that, as HRGs are context free, for any $H \in HG_{\Sigma_N}$, and $e_1, e_2 \in E_H$ $H[e_1/K_1][e_2/K_2] = H[e_2/K_2][e_1/K_1]$. Therefore we can reorder replacement sequences as long as we ensure, that edges are not replaced before they are introduced, i.e. $e_j \in E_{K_i} \Longrightarrow j > i$. For any replacement $[e/K]$ of a sequence we define the set of successors of $[e/K]$ recursively by

$$successors([e/K]) = \bigcup_{e' \in E^N_K} successors([e'/K']) \cup E^N_K.$$

We split G^X_i into the set of old production rules P_{old}, i.e. productions which are adopted from G without changings, and the set of new production rules P_{new}, i.e. productions generated during elimination of local (X,i)-recursion. We know that all rules from P_{new} are of the form $X \to H_B$ or $B \to J_n \mid J_t$, where E_{X_n} contains an edge labelled with B.

"$\Longleftarrow$" : We reorder a given derivation $H \Rightarrow^* K : H[e_1/K_1]\dots[e_n/K_m] = K$ such that each $[e_i/K_i]$, with $lab(e_i) \to K_i \in P_{new}$ is directly succeeded by the elements of $\{[e/K] \in successors([e_i/K_i]) \mid lab(e) \to K \in P_{new}\}$, i.e. the successors contained in P_{new}. This yields the following derivation:

$$H \underbrace{[e_1/K_1]\dots[e_{i-1},K_{i-1}]}_{\in P_{old}} \underbrace{[e_i/K_i]\dots[e_j/K_j]}_{\in P_{new}} [e_j/K_{j+1}]\dots = K$$

Note that there can be more then one or none subderivations $[e_i/K_i]\dots[e_j/K_j]$ from P_{new}. (*) The replacements before the first of such subderivations do not contain any rules from P_{new} and thus no edge labelled with B is introduced. Therefore $H[e_1/K_1]\dots[e_{i-1},K_{i-1}] \in HG_{\Sigma_N}$ can be derived in G accordingly.

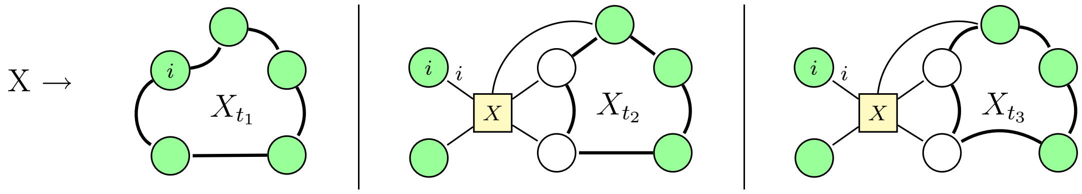


Fig. 8.

Consider the grammar G given in Figure 8. By construction we know that P_{new} is of the form given in Figure 9 and the subderivation has the following form:

$$H[e_i/K][e_{i+1}/J_{n_1}]\dots[e_{j-1}/J_{N_{j-2}}][e_j/J_t] = K' \in HG_{\Sigma_N}.$$

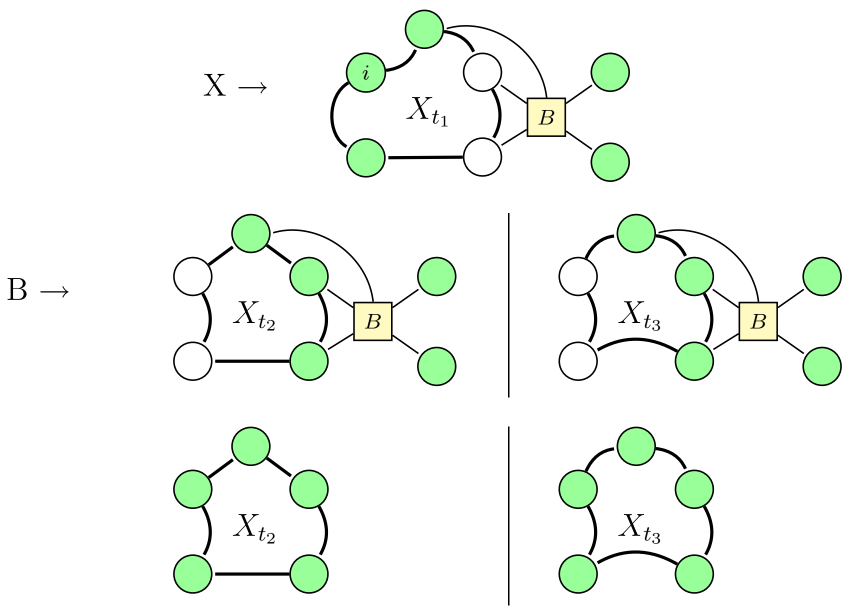


Fig. 9.

We show by induction over the amount n of $B \to J_n$ rules, that for any such sequence a corresponding derivation $H \Rightarrow^* K'$ in G exists.

(IB) n = 0: We consider the derivation $H_1 \overset{X \to H_B}{\Rightarrow} H_2 \overset{B \to J_t}{\Rightarrow} K'$ of G_i^X, see Figure 9 for rules. The corresponding derivation in G is given by $H_1 \overset{X \to X_n}{\Rightarrow} H_2' \overset{X \to X_t}{\Rightarrow} K'$, where $X \to X_n$ generates the nodes and edges from $B \to J_t$ while the remaining graph is generated by $X \to X_t$.

(IH) A corresponding derivation can be found for an arbitrary but fixed number n of rule applications $B \to J_n$, $n \in \mathbb{N}$.

(IS) $n \to n+1$: Consider the derivation

$$H \overset{X \to H_B}{\Rightarrow} H_1 \overset{B \to J_{n_1}}{\Rightarrow} \dots \overset{B \to J_{n_{n+1}}}{\Rightarrow} H_{n+1} \overset{B \to J_t}{\Rightarrow} K'$$

By (IH) we know that

$$H \overset{X \to H_B}{\Rightarrow} H_1 \overset{B \to J_{n_1}}{\Rightarrow} \dots \overset{B \to J_{n_n}}{\Rightarrow} H_n \overset{B \to J_t}{\Rightarrow} = K'$$

can be simulated in G via

$$H \overset{X \to X_n}{\Rightarrow} H_1' \overset{X \to X_{n_1}}{\Rightarrow} \dots \overset{X \to X_{n_n}}{\Rightarrow} H_n' \overset{X \to X_t}{\Rightarrow} K'.$$

We extend this derivation by inserting the rule $X \to X_{n_{n+1}}$ corresponding to $B \to J_{n_{n+1}}$ at the beginning of the derivation. See Figure 10 for the correspondence of the G_i^X- and the G-derivation. Note that graph parts derived during the i-th derivation step of G_i^X, are derived in G by derivation step $(n+1) - (i-1) = n+2-i$, where $n+1$ is the overall number of derivation steps.

$$H \overset{X \to X_n}{\Rightarrow} H_1' \overset{X \to X_{n_{n+1}}}{\Rightarrow} H_2' \overset{X \to X_{n_1}}{\Rightarrow} \dots \overset{X \to X_{n_n}}{\Rightarrow} H_{n+1}' \overset{X \to X_t}{\Rightarrow} K'.$$

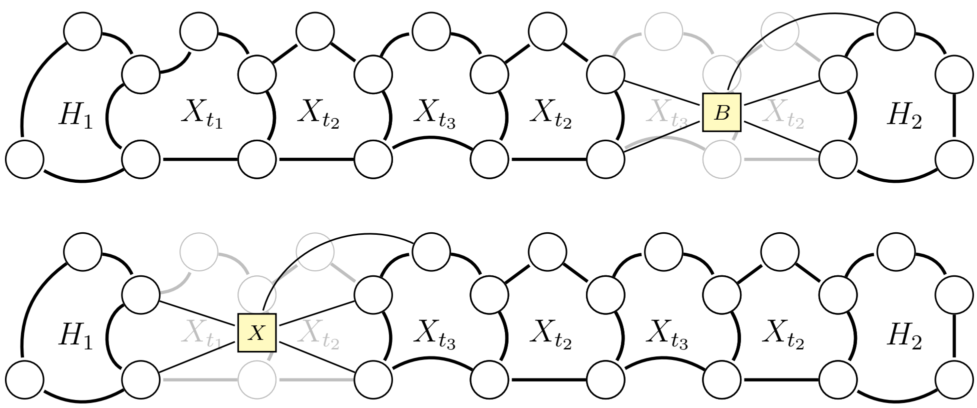


Fig. 10.

"$\Longrightarrow$" : Analogous, by considering subderivations of local (X, i)-recursive rules.

The remaining derivation $H[e_j/K_{j+1}]\cdots = K$ is handled exactly the same way, starting from (*). Thus every $K \in HG_\Sigma$ derivable in G_i^X is derivable in G.

During the LGNF construction further nonterminals are introduced when removing local recursion. As for this newly introduced nonterminals LFNG has to established, we have to make sure, that only finitely many new nonterminals are generated. We achieve this by merging nonterminals with equal right-hand sides.

Theorem 7. *After finitely many steps a nonterminal can be merged, thus the construction of LGNF terminates.*

Proof (Cyclic nonterminal reappearance). By construction we know that for every locally recursive rule exactly two production rules for the new nonterminal B are generated. The first is obtained from the respective original rule, where the locally recursive edge and corresponding external nodes are removed. The second is a copy of the first rule with an additional nonterminal edge labelled with B on the right-hand side. This means that the subgraph used as basis for the construction of production rules for new nonterminals remains the same. In addition exactly one edge (labelled with B) is used for construction. As there exist only finally many possibilities to combine subgraph and edge eventually a nonterminal will be produced whose associated rules coincide with those of an already existing nonterminal (up to renaming). Thus we can merge these nonterminals as any further constructed production rules would be redundant.

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from `http://aib.informatik.rwth-aachen.de/`**. To obtain copies consult the above URL or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email:** `biblio@informatik.rwth-aachen.de`

2008-01 * Fachgruppe Informatik: Jahresbericht 2007
2008-02 Henrik Bohnenkamp, Marielle Stoelinga: Quantitative Testing
2008-03 Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, Harald Zankl: Maximal Termination
2008-04 Uwe Naumann, Jan Riehme: Sensitivity Analysis in Sisyphe with the AD-Enabled NAGWare Fortran Compiler
2008-05 Frank G. Radmacher: An Automata Theoretic Approach to the Theory of Rational Tree Relations
2008-06 Uwe Naumann, Laurent Hascoet, Chris Hill, Paul Hovland, Jan Riehme, Jean Utke: A Framework for Proving Correctness of Adjoint Message Passing Programs
2008-07 Alexander Nyßen, Horst Lichter: The MeDUSA Reference Manual, Second Edition
2008-08 George B. Mertzios, Stavros D. Nikolopoulos: The λ-cluster Problem on Parameterized Interval Graphs
2008-09 George B. Mertzios, Walter Unger: An optimal algorithm for the k-fixed-endpoint path cover on proper interval graphs
2008-10 George B. Mertzios, Walter Unger: Preemptive Scheduling of Equal-Length Jobs in Polynomial Time
2008-11 George B. Mertzios: Fast Convergence of Routing Games with Splittable Flows
2008-12 Joost-Pieter Katoen, Daniel Klink, Martin Leucker, Verena Wolf: Abstraction for stochastic systems by Erlang's method of stages
2008-13 Beatriz Alarcón, Fabian Emmes, Carsten Fuhs, Jürgen Giesl, Raúl Gutiérrez, Salvador Lucas, Peter Schneider-Kamp, René Thiemann: Improving Context-Sensitive Dependency Pairs
2008-14 Bastian Schlich: Model Checking of Software for Microcontrollers
2008-15 Joachim Kneis, Alexander Langer, Peter Rossmanith: A New Algorithm for Finding Trees with Many Leaves
2008-16 Hendrik vom Lehn, Elias Weingärtner and Klaus Wehrle: Comparing recent network simulators: A performance evaluation study
2008-17 Peter Schneider-Kamp: Static Termination Analysis for Prolog using Term Rewriting and SAT Solving
2008-18 Falk Salewski: Empirical Evaluations of Safety-Critical Embedded Systems
2008-19 Dirk Wilking: Empirical Studies for the Application of Agile Methods to Embedded Systems
2009-02 Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Quantitative Model Checking of Continuous-Time Markov Chains Against Timed Automata Specifications

2009-03 Alexander Nyßen: Model-Based Construction of Embedded Real-Time Software - A Methodology for Small Devices
2009-04 Daniel Klünder: Entwurf eingebetteter Software mit abstrakten Zustandsmaschinen und Business Object Notation
2009-05 George B. Mertzios, Ignasi Sau, Shmuel Zaks: A New Intersection Model and Improved Algorithms for Tolerance Graphs
2009-06 George B. Mertzios, Ignasi Sau, Shmuel Zaks: The Recognition of Tolerance and Bounded Tolerance Graphs is NP-complete
2009-07 Joachim Kneis, Alexander Langer, Peter Rossmanith: Derandomizing Non-uniform Color-Coding I
2009-08 Joachim Kneis, Alexander Langer: Satellites and Mirrors for Solving Independent Set on Sparse Graphs
2009-09 Michael Nett: Implementation of an Automated Proof for an Algorithm Solving the Maximum Independent Set Problem
2009-10 Felix Reidl, Fernando Sánchez Villaamil: Automatic Verification of the Correctness of the Upper Bound of a Maximum Independent Set Algorithm
2009-11 Kyriaki Ioannidou, George B. Mertzios, Stavros D. Nikolopoulos: The Longest Path Problem is Polynomial on Interval Graphs
2009-12 Martin Neuhäußer, Lijun Zhang: Time-Bounded Reachability in Continuous-Time Markov Decision Processes
2009-13 Martin Zimmermann: Time-optimal Winning Strategies for Poset Games
2009-14 Ralf Huuck, Gerwin Klein, Bastian Schlich (eds.): Doctoral Symposium on Systems Software Verification (DS SSV'09)
2009-15 Joost-Pieter Katoen, Daniel Klink, Martin Neuhäußer: Compositional Abstraction for Stochastic Systems
2009-16 George B. Mertzios, Derek G. Corneil: Vertex Splitting and the Recognition of Trapezoid Graphs
2009-17 Carsten Kern: Learning Communicating and Nondeterministic Automata
2009-18 Paul Hänsch, Michaela Slaats, Wolfgang Thomas: Parametrized Regular Infinite Games and Higher-Order Pushdown Strategies
2010-02 Daniel Neider, Christof Löding: Learning Visibly One-Counter Automata in Polynomial Time
2010-03 Holger Krahn: MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering
2010-04 René Wörzberger: Management dynamischer Geschäftsprozesse auf Basis statischer Prozessmanagementsysteme
2010-05 Daniel Retkowitz: Softwareunterstützung für adaptive eHome-Systeme
2010-06 Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Computing maximum reachability probabilities in Markovian timed automata
2010-07 George B. Mertzios: A New Intersection Model for Multitolerance Graphs, Hierarchy, and Efficient Algorithms
2010-08 Carsten Otto, Marc Brockschmidt, Christian von Essen, Jürgen Giesl: Automated Termination Analysis of Java Bytecode by Term Rewriting
2010-09 George B. Mertzios, Shmuel Zaks: The Structure of the Intersection of Tolerance and Cocomparability Graphs

2010-10 Peter Schneider-Kamp, Jürgen Giesl, Thomas Ströder, Alexander Serebrenik, René Thiemann: Automated Termination Analysis for Logic Programs with Cut

2010-11 Martin Zimmermann: Parametric LTL Games

2010-12 Thomas Ströder, Peter Schneider-Kamp, Jürgen Giesl: Dependency Triples for Improving Termination Analysis of Logic Programs with Cut

2010-13 Ashraf Armoush: Design Patterns for Safety-Critical Embedded Systems

2010-14 Michael Codish, Carsten Fuhs, Jürgen Giesl, Peter Schneider-Kamp: Lazy Abstraction for Size-Change Termination

2010-15 Marc Brockschmidt, Carsten Otto, Christian von Essen, Jürgen Giesl: Termination Graphs for Java Bytecode

2010-16 Christian Berger: Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles

2010-17 Hans Grönniger: Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten

2010-18 Ibrahim Armaç: Personalisierte eHomes: Mobilität, Privatsphäre und Sicherheit

2010-19 Felix Reidl: Experimental Evaluation of an Independent Set Algorithm

2010-20 Wladimir Fridman, Christof Löding, Martin Zimmermann: Degrees of Lookahead in Context-free Infinite Games

* These reports are only available as a printed version.

Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.