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 based on separation logic [10, 14] (duality with hyperedge replacement grammars is observed in [4]) and regular tree automata [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 achieved. 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, 12, 11, 1, 13]. 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. The full version of this paper [1] contains the omitted proofs.

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 $t = (A, B, C, \dots)$ we write A_t, B_t etc. for the components if their names are clear from the context. Function $f \upharpoonright S$ 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 pairwise 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.

[1] Technical Report AIB-2011-04 available from http://aib.informatik.rwth-aachen.de

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.

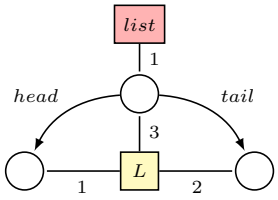


Fig. 1. Heap as hypergraph

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.

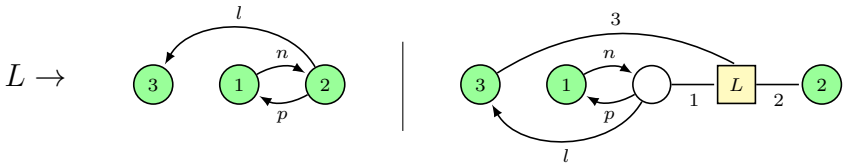


Fig. 2. A grammar for doubly-linked lists

The HRG derivation steps are defined through hyperedge replacement.

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

Example 3. Reconsider the hypergraph H of Fig. 1 as well as the second rule in Fig. 2, denoted by $L \to K$. 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.

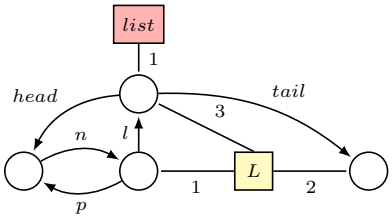


Fig. 3. Hyperedge replacement

Definition 5 (HRG Derivation). *Let $G \in HRG_{\Sigma_N}$, $H, H' \in HG_{\Sigma_N}$, $p = X \to K \in G$ and $e \in E_H$ with $lab(e) = X$. H* derives *H' by p iff H' is isomorphic to $H[K/e]$. $H \overset{e,p}{\Longrightarrow} H'$ refers to this derivation. Let $H \overset{G}{\Longrightarrow} H'$ if $H \overset{e,p}{\Longrightarrow} H'$ 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 $X \in N$ with $rk(X) = n$, an* X-handle *is the hypergraph $X^\bullet = (\{v_1, \ldots, v_n\}, \{e\}, [e \mapsto v_1 \ldots v_n], [e \mapsto X], v_1 \ldots v_n) \in HG_\Sigma$.*

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 $X \in N$, $i \in [1, rk(X)]$, 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.

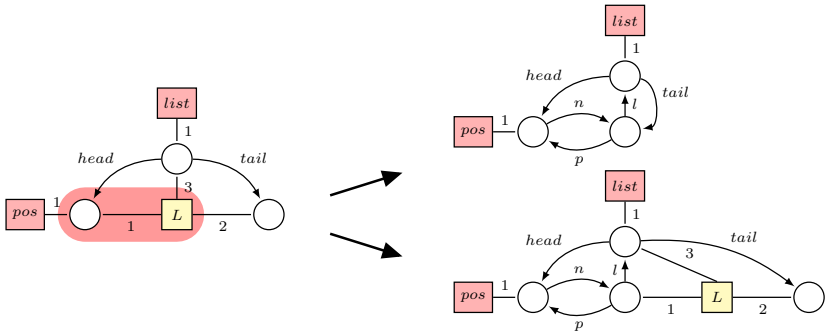


Fig. 4. Concretisation of inadmissible heap configurations

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 [16].

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 terminal or "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* $\forall X \in N, i \in [1, rk(X)]$, $\exists type(X,i) \subseteq \Sigma : \forall H \in L(X^\bullet) :\ type(X,i) = out_H(ext_H(i))$.

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.

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 a (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 $\overline{G^X} = G \backslash G^X$.

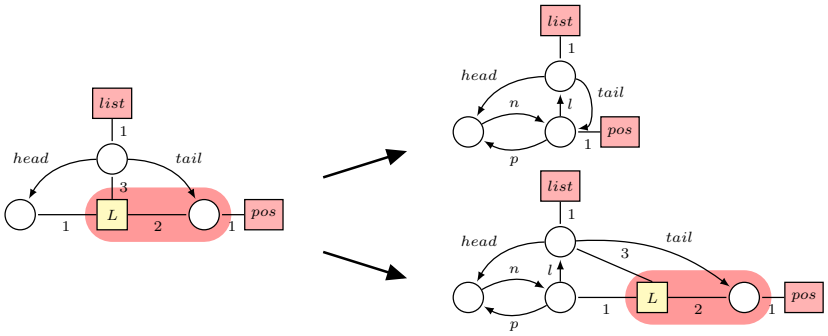


Fig. 5. Inadmissible concretisation

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

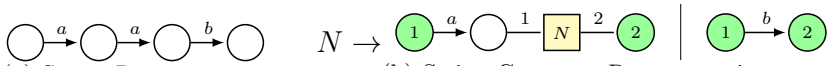
The Greibach Normal Form (GNF) for string grammars restricts production rules to the form $X \to aN_1 \dots N_k$ such that using left derivation only a word $w \in \Sigma^n N^\star$ is derived after n derivation steps. Thus terminal words are constructed from left to right extending a terminal prefix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

Up to now the normal form given in [6] is the only notion of a GNF for graph grammars widely known and accepted. 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. $\forall\, p \in G_i^X\colon (X, i) \to_p (Y, j)$ *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 $X \to aX \mid b$ 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 $T_1, \ldots, T_n$ over N. For increasing $i \in [1, n]$ (1) every rule p, such that $T_i \to_p T_j$ 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 we define an ordering $T_1, \ldots, T_n$ on non-reduction tentacles.

Step 1: Elimination of rules. We first eliminate the rules $p = X \to H \in G_i^X$ with $(X, i) = T_k \to_p T_l$, $l < k$. 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 $\{X \to H[K/e] \mid Y \to K \in G_j^Y\}$. 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 $L_G(H) = L_{G'}(H)$ for all $H \in HG_{\Sigma_N}$.*

Note that G_j^Y does not contain any rule p with $T_l \to_p T_m$, $m < l$, 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 $T_i \to_p T_i$ 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 a rule p with $(X, i) \to_p (X, i)$.*

Let $G_r^X \subseteq G_i^X$ be the set of all rules locally recursive at (X, i). To remove local recursion in $p = X \to H \in G_i^X$ we introduce a new nonterminal B'_j, a recursive rule $B'_j \to J_n$ and an exit rule $B'_j \to J_t$. 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' ($fill(i) = ext_{J_t}(i)$ *if* $ext_{J_t}(i) \in att_H(e)$, *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:

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

$B'_j \to J_t$ with:

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

$B'_j \to J_n$ with:

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

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

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

Example 9. The doubly-linked list HRG with production rules $L \to H \mid J$ given in Fig. 2 is locally recursive at $(L, 2)$. We introduce nonterminal B' and the rules $B' \to J_t \mid J_n$, cf. Fig. 7. The terminal right-hand side 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.

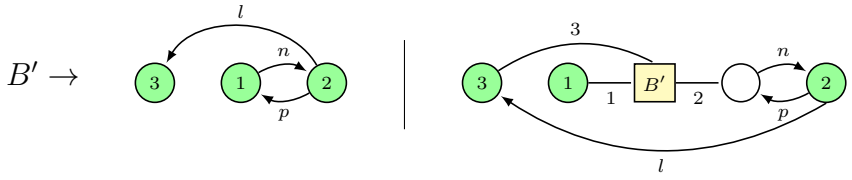


Fig. 7. Doubly-Linked Lists: $G^{B'}_2$

Step 3: Generation of Greibach rules. Starting at the highest order tentacle, for each G^X_i 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.

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 arbitrarily. 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 rule 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 results 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 degree 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 leads 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 [17] 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. Volume 3051, Springer (2003) 48–61
2. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract Regular Tree Model Checking. ENTCS **149** (2006) 37–48
3. Distefano, D., Katoen, J.P., Rensink, A.: Safety and Liveness in Concurrent Pointer Programs. In: FMCO. Volume 5 (2005) 280–312
4. Dodds, M.: From Separation Logic to Hyperedge Replacement and Back. In: ICGT '08. Volume 5214 (2008) 484–486
5. Dumitrescu, S.: Several Aspects of Context Freeness for Hyperedge Replacement Grammars. W. Trans. on Comp. **7** (2008) 1594–1604
6. Engelfriet, J.: A Greibach Normal Form for Context-free Graph Grammars. In: ICALP '92. Volume 623 (1992) 138–149
7. Habel, A.: Hyperedge Replacement: Grammars and Languages. Springer-Verlag New York (1992)
8. Heinen, J., Noll, T., Rieger, S.: Juggrnaut: Graph Grammar Abstraction for Unbounded Heap Structures. TTSS '09 (to be published in ENTCS) (2009)
9. Klempien-Hinrichs, R.: Normal Forms for Context-Free Node-Rewriting Hypergraph Grammars. Math. Structures in Comp. Sci. **12** (2002) 135–148
10. O'Hearn, P.W., Hongseok, Y., Reynolds, J.C.: Separation and Information Hiding. POPL '04 **39** (2004) 268–280
11. Rensink, A.: Canonical Graph Shapes. In: ESOP '04. Volume 2986 of LNCS (2004) 401–415
12. Rensink, A.: Summary from the Outside In. AGTIVE'03 **3062** (2004) 486–488
13. Rensink, A., D., D.: Abstract Graph Transformation. SVV '05 **157** (2006) 39–59
14. Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. In: LICS '02 (2002) 55–74
15. Rieger, S., Noll, T.: Abstracting Complex Data Structures by Hyperedge Replacement. In: ICGT '08. Volume 5214 (2008) 69–83
16. Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformation: vol. World Scientific Publishing Co., Inc., River Edge, NJ, USA (1997)
17. Rozenberg, G., Welzl, E.: Boundary NLC Graph Grammars-Basic Definitions, Normal Forms, and Complexity. Inf. Control **69** (1986) 136–167
18. Sagiv, M., Reps, T., Wilhelm, R.: Parametric Shape Analysis via 3-Valued Logic. ACM Trans. Program. Lang. Syst. **24** (2002) 217–298