RWTH Aachen University
Chair for Software Modeling and Verification

Bachelor-Thesis

Heap Abstraction Beyond Context-freeness

Hannah Arndt

March 15, 2016

First Supervisor: Thomas Noll
Second Supervisor: Joost-Pieter Katoen

Advisor: Christoph Matheja

Abstract. This thesis introduces and analyses variants of hyperegde replacement grammars in the context of heap abstraction. In particular automata controlled, Indian parallel and indexed grammars are studied. Automata controlled grammars are especially considered for their potential to ensure backward-confluence of grammars. Indian parallel and indexed grammars are both considered for their capability to express balanced trees. While Indian parallel grammars turn out to lack an appropriate abstraction mechanism, indexed grammars reveal a high potential to be of practical relevance. This thesis analyses all three variants regarding the decidability of the emptiness problem and the existence of an abstraction mechanism that ensures safety. It also includes a detailed example that shows how balancedness of AVL trees is preserved by a standard insertion algorithm.

Table of Contents

1 Introduction

The formal verification of programs is an important field of research in computer science, as software has increasing relevance for both the economy and our daily life. When the potential financial loss caused by errors becomes very large or even lives depend on the reliability of software (for example in case it controls a surgical robot) it is important to aim at a formal proof of correctness in addition to testing. While testing can only detect errors but never ensure the absence of them, verification works the other way around: One tries to show that a piece of software satisfies a specification.

Many real life programs use pointers, especially to build dynamic data structures, such as lists and trees. It is therefore particularly interesting to consider such pointer-programs. Typical properties of interest for these programs are the absence of memory leaks and null dereferencing or the preserving invariants for the data structure, e.g. absence of cycles.

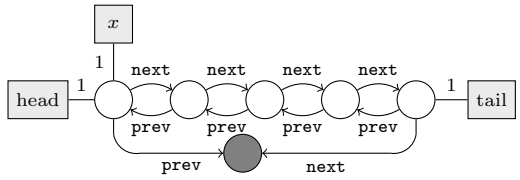
A major challenge is that dynamic data structures, as they typically occur in pointer programs, induce a potentially infinite state space since objects are created at runtime and the input to the program might be of arbitrary size.

Consider for example a sorting algorithm on a doubly-linked list where we want to ensure that no null pointers are dereferenced during the execution. Such an algorithm should be able to work with lists of arbitrary length and an error might occur only at lists with very specific lengths. Even if we ignore the infinite combinations of data in a list of given length, it is impossible to test all lists of arbitrary length for the desired property.

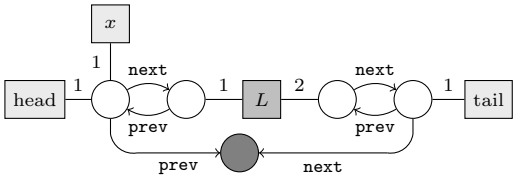
This thesis is based on an approach by Heinen et al. that represents the heap as a hypergraph. Figure 1a displays the representation of a doubly-linked list with five elements where the first one is referenced by two variables *head* and x and the last one by a variable *tail*. Variables are represented by hyperedges of rank one (displayed by squares). Objects, in this case the list elements, are represented by vertices. The references between the objects, in the example *next* and *prev*, are represented by hyperedges of rank two (usually represented by labelled arrows). An additional vertex (displayed in dark grey) represents the null references.

As we have already discussed, it is not possible to consider all lists of different lengths explicitly, therefore one must consider abstract heaps that represent several concrete ones at the same time. Heinen et al. do this by abstracting subgraphs that are currently not directly referenced by program variables and can therefore not be accessed by the program in the next step, into place-holders as is shown exemplary in Figure 1b. These place-holders are again hyperedges, displayed by squares that are linked to their adjacent vertices. The idea is that L represents a middle segment for the given list of arbitrary length. If the program under consideration moves x one element further, L expands again thereby considering the two cases that the middle segment represented an empty list or did contain more vertices. Figure 2 shows these two cases.

The process of concretising place-holders to sub-graphs (where the subgraphs may contain place-holders of their own) is controlled by a context-free



(a) Concrete heap



(b) Abstract heap

Fig. 1: Heap as hypergraph

hyperedge replacement grammar, where the non-terminals are used as placeholders. The grammar in the backward direction is used to abstract sub-graphs into non-terminals. An example grammar with two rules is shown in Figure 3. The first rule formalizes the concretisation in Figure 2a. The two vertices enumerated by one and two, respectively, are identified with the vertices attached to the L-symbol in Figure 1b by the links enumerated with one and two. Therefore, the *next*- and *prev*-edges that link the vertices in the rule, are inserted between the corresponding vertices in the list. The same procedure is repeated with the second rule yielding the hypergraph in Figure 2b.

All concepts that have been addressed here are formulated and explained in more detail in Section 3

Similar to context-free string grammars, the application of a rule to a non-terminal depends only on the non-terminal, specified by the left-hand side of a rule. On the one hand, context-free hyperedge replacement grammars are well studied and yield several good properties [8,6] but on the other hand they also restrict the expressiveness [14]. For example, complete or balanced trees cannot be expressed, since the grammar has no access to the depth of different sub-trees while generating a branch. Several important data structures, such as AVL and red-black trees are therefore out of scope of the original approach.

In this thesis we study three derivation controlled variations of context-free hyperedge replacement grammars that were originally studied for string grammars and discuss whether they can extend the approach by Heinen et al. in a useful way.

First, we study grammars where the derivation is controlled by a deterministic finite automaton (Section 6) and we observe that this extension can po-

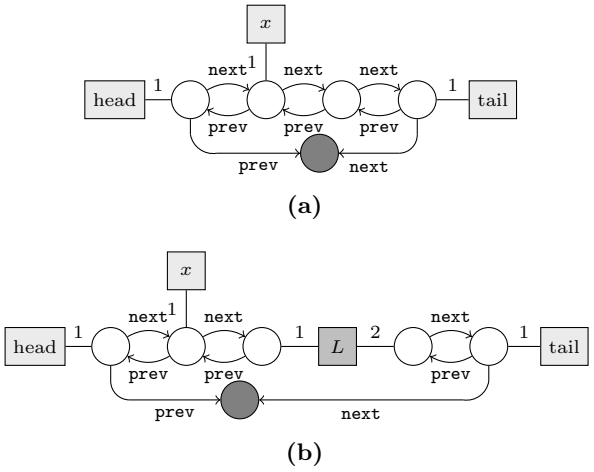


Fig. 2: Moving x forward and expanding L

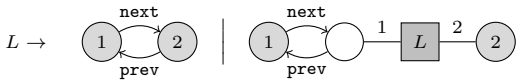


Fig. 3: Grammar for doubly-linked list

tentially contribute to establish backward-confluence, i.e. make sure that the abstraction has a unique result.

We then consider Indian parallel grammars (Section 7), a variation where all non-terminals with the same label have to be replaced in parallel by the same rule. This variation is capable of expressing complete trees, but it turns out to be unsuited for heap abstraction.

Finally, we turn to indexed grammars. They are not only capable of expressing complete, but also balanced trees as they occur in AVL-trees. We show that indexed grammars preserve many interesting properties of context-free hyperedge replacement grammars and we consider indexed grammars therefore a natural extension of the approach by Heinen et al. [11]. We complete the section on indexed grammars by giving a detailed example how indexed grammars can be used to show that the balancing of AVL-trees is preserved by a standard insertion algorithm.

2 Related Work

This thesis studies extensions and variations of the heap abstraction approach by Heinen et al. [11]. They have introduced a symbolic execution on abstract graphs using context-free hyperedge replacement grammars as we have shortly discussed

in the introduction. More detail on this approach is provided in Section 3 since it is essential for this thesis.

The trade-off between desirable properties and expressiveness of grammars occurs also for string grammars [4]. There are many extensions of context-free string grammars, for example by providing further mechanisms to control the derivation [4, Chapter 1] or by adding some amount of context sensitivity [4, Chapter 2]. The Handbook of Formal Languages [5] provides a comprehensive survey for controlled string grammars.

The literature contains some work that transfers such variants to HRGs. There, the focus lies on general properties like the expressiveness or decidability of the emptiness and membership problem. Habel [8, Chapter VIII] has studied application conditions such as occurrence checks and parallel derivations. Parallel controlled grammars are one possibility to express complete trees and we study in Section 7 whether they can be used to verify algorithms on such data structures. Kreowski [14] has additionally mentioned the possibility to control the derivation with a regular language. While he combines this with other control mechanisms, we study hyperedge replacement grammars where a regular language is the only control mechanism in Section 6. There is also more specialized work on this field: For example Maneth [15] has studied the expressive power of two variants of cooperating distributed hyperedge replacement grammars which can be seen as a special case of parallel grammars. His work also considers the relation between hyperedge replacement grammars and their string counterparts and links them using their derivation trees. We also make use of this connection in Section 4 to show that the emptiness problem is decidable for all considered grammars.

The main contribution of this thesis is that we show that indexed grammars can be used to model complex operations on dynamic data structures. In particular, we hope that our analysis provides the foundations to improve the approach by Heinen et al. to handle algorithms on balanced trees. There are only very few approaches that are able to do this so far:

Manna et al. [16] present a term algebra together with a size function that defines the black depth of a term. This term algebra together with presburger arithmetic ($\langle \mathbb{Z}; 0, +, < \rangle$) yields a decidable theory of red-black trees. It can then be used to verify programs manually in a Hoare-like approach.

Rugina [17] has presented an approach based on shape analysis that includes quantitative properties to address the balancing property of AVL-trees. They can do the analysis fully automatic, notably they do not need to provide loop-invariants manually which is a disadvantage of other approaches [16,9]. On the other hand their approach is specialised to the verification of AVL-trees and therefore limited to this special data-structure.

Habermehl et al. [9] have presented yet an other approach based on tree automata with size constraints (TASC). The languages of TASCs are shown to be closed under union, intersection and complement and further emptiness is decidable. Using a restricted class rTASC that preserves the most important properties, they can validate Hoare triples where $\langle A \rangle P \langle A' \rangle$ holds if A' accepts a tree t' if and only if there is a tree t accepted by A and t' is obtained by

applying the program P to t. Though this approach is sound, it has several disadvantages: Pre- and postconditions as well as loop invariants have to be provided manually as TASCs which is a very complex task. Further, program segments that temporarily break the tree structure cannot be verified in a step-by-step manner, but have to be considered atomic.

3 Foundations

The notations and definitions in this section are based on the paper on heap abstraction by Heinen et al. [11]. Some basic notations for functions and sequences that we use throughout this paper can be found in Appendix A.

3.1 Context-free Hyperedge Replacement Grammars

We first introduce the concepts of hypergraphs and hyperedge replacement grammars, as they have been developed independently from heap abstraction (originally under the name of plex grammars by Feder [7]) and have various applications, e.g. to generate chemical structures or electric circuits [7].

Hypergraphs A hypergraph is a more general form of graph, where edges do not necessarily link two but any number of vertices. The edges are labelled by symbols from an alphabet Σ and the label determines how many adjacent vertices the hyperedge has. We say the alphabet is *ranked* as each symbol is assigned a natural number. Hyperedges are denoted by squares which are attached to the adjacent vertices by enumerated links. Hyperedges that link exactly two vertices may also be written as an arrow from the first to the second adjacent vertex with their label written above them. Two examples can be seen in Figure 1: While the edges labelled by *next*, *prev* and L have two adjacent vertices, the edges labelled by *head*, *tail* and x have only one.

Further, hypergraphs have a (possibly empty) sequence of pairwise distinct external vertices. The length of this sequence is called the *type* of the hypergraph. The hypergraphs in Figure 1 have no external vertices, but the graphs on the left-hand side in the grammar in Figure 3 have each two, denoted by the numbers written within them.
In the following we define hypergraphs formally:

Definition 1 (Hypergraph). Let Σ be a finite ranked alphabet, i.e. each symbol $X \in \Sigma$ is assigned a rank $rk(X) \in \mathbb{N}_0$.

A *hypergraph* is a tuple $H = (V, E, att, lab, ext)$, where V is a set of vertices and E a set of hyperedges, the function $lab : E \to \Sigma$ maps every hyperedge to its label, and $att : E \to V^*$ assigns each hyperedge the sequence of its attached vertices, such that $rk(e) := |att(e)| = rk(lab(e))$. The ith adjacent vertex to hyperedge e is $att(e, i)$.

$ext \in V^*$ is a possibly empty sequence of pairwise distinct external vertices. The length of the sequence ext, i.e. the number of external vertices, is called the

type of H, written $type(H)$. The ith external vertex is referred to by $ext(i)$ for $i \in [1, type(H)]$.

The *set of all hypergraphs* over an alphabet Σ is HG_Σ.

Consider for example the right-hand side of the second rule in the HRG for doubly-linked lists in Figure 3. Here, $\Sigma = \{next, prev, L\}$ and $rk(next) = rk(prev) = rk(L) = 2$. We have three vertices, i.e. $V = \{v_1, v_2, v_3\}$ (arbitrarily enumerated from left to right) and three edges, i.e. $E = \{e_{next}, e_{prev}, e_L\}$ (named by their labels for convenience), where for example $lab(e_{next}) = next$. As an example of the attachment function we have here $att(e_{next}) = v_1v_2$. Finally, the sequence of external vertices is v_1v_3.

During the course of this thesis, the prefix "hyper-" will often be omitted to increase readability. If not explicitly stated otherwise, graph and edge will refer to hypergraph and hyperedge, respectively.

Hyperedge Replacement Grammars In the introduction we have seen that non-terminals can be replaced by graphs and vice versa. Let us first specify what it formally means to replace an edge by a graph:

Definition 2 (Hyperedge replacement). Let $H \in \mathrm{HG}_{N\cup T}$ be a graph and $e \in E_H$ one of its (non-terminal) edges. Let $K \in \mathrm{HG}_{N\cup T}$ with $type(K) = rk(e)$ be the graph to be inserted. W.l.o.g. let H and K be disjoint, i.e. $V_H \cap V_K = E_H \cap E_K = \emptyset$. The replacement of e by K, denoted by $H[K/e]$, is the graph $H' \in \mathrm{HG}_{N\cup T}$ defined by

$$
\begin{aligned}
V_{H'} &= V_H \cup (V_K \backslash [ext_K]) & E_{H'} &= (E_H \backslash \{e\}) \cup E_K \\
lab_{H'} &= (lab_H \upharpoonright (E_H \backslash \{e\})) \cup lab_K & ext_{H'} &= ext_H \\
att_{H'} &= att_H \upharpoonright (E_H \backslash \{e\}) \cup (mod \circ att_K)
\end{aligned}
$$

where $mod = id_{V_{H'}} \cup \{ext_K(1) \mapsto att_H(e, 1), ..., ext_K(rk(e)) \mapsto att_H(e, rk(e))\}$.

For example replacing the L-edge in Figure 1b with the first graph on the right hand side in Figure 3 yields the graph in Figure 2a. The edge e_L is removed from the graph, then the hypergraph K is added disjointly to H. Finally, the ith external vertex of K is identified with the ith tentacle of e_L.

Hyperedge replacement grammars specify rules which edges can be replaced by which hypergraphs. The derivation terminates if the hypergraph is fully concrete, i.e. if all its edges are labelled by terminals.

Definition 3 (Hyperedge replacement grammar (HRG)). A *hyperedge replacement grammar* is a tuple $G = (N, T, P)$, where T and N denote the terminals and non-terminals. Together they form the set of edge labels, i.e. all intermediate graphs are in $\mathrm{HG}_{N\cup T}$. P is a finite set of production rules of the form $p : X \to H$, where $X \in N$ is a non-terminal edge label and $H \in HG_{N\cup T}$ is a hypergraph over the alphabet $N \cup T$. We write $lhs(p)$ for the left-hand side X and $rhs(p)$ for the right-hand side H of the rule. The rank of the left and the type of the right-hand side of the rule must agree, i.e. $rk(X) = type(H)$.

Since HRGs distinguish edges with terminal and non-terminal labels, we define two sets

$E^N = \{e \in E \mid lab(e) \in N\}$, the set of all edges labelled by a non-terminal. We refer to them as *non-terminal edges*.

$E^T = \{e \in E \mid lab(e) \in T\}$, the set of all edges labelled by a terminal. Accordingly referred to as *terminal edges*.

The mechanism of applying production rules is edge replacement:

Definition 4 (Rule application). Let G be an HRG and $H \in HG_{N \cup T}$ a hypergraph with a hyperedge e labelled by the non-terminal X. Then applying a production rule $p : X \to K$ of G yields $H' \cong H[K/e]$, written as $H \Rightarrow_{p,e} H'$. If the identity of the edge or the rule is irrelevant it can be omitted ($H \Rightarrow_p H', H \Rightarrow H'$).

The reflexive and transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$.

The *language* of a hypergraph H with respect to grammar G is $\mathcal{L}(H) = \{K \in \mathrm{HG}_T \mid H \Rightarrow^* H'\}$ the set of terminal hypergraphs derivable from H.

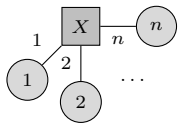


Fig. 4: A handle for n-ary symbol X

Abstraction In the introduction we have only vaguely stated that sub-graphs that are currently not directly accessible for the program are stored in a non-terminal to reduce the state space, i.e. the number of different (abstract) heaps that have to be considered. We now formalise how this is done using an HRG.

For this, we need a graph that consists of exactly one hyperedge labelled by a specific non-terminal and its adjacent vertices, which are at the same time the external vertices of the graph.

Definition 5 (Handle). For an edge label X of rank n, an *X-handle* is a hypergraph $X^\bullet = (\{v_1, \ldots, v_n\}, \{e\}, \{e \mapsto v_1 \ldots v_n\}, \{e \mapsto X\}, v_1 \ldots v_n)$. Further, $X^\circ = (\{v_1, \ldots, v_n\}, \{e\}, \{e \mapsto v_1 \ldots v_n\}, \{e \mapsto X\}, \epsilon)$ is a handle without external vertices.

During the abstraction process, an 'occurrence' (embedding) of the left-hand side of a rule is replaced by the handle corresponding to its right-hand side. Therefore, we introduce an embedding, i.e. a mapping from the components of a 'smaller' graph K to a sub-graph of the 'big' graph H.

Definition 6 (Embedding). Given hypergraphs $K, H \in \mathrm{HG}_{N \cup T}$, an *embedding emb* of K in H is a pair of mappings $emb_V : V_K \to V_H$ and $emb_E : E_K \to E_H$ with the following properties:

$$
\begin{aligned}
emb_V(v) &\neq emb_V(V') && \forall v \in V_K, v' \in V_K \backslash [ext_K] \text{ with } v \neq v' \\
emb_E(e) &\neq emb_E(e') && \forall e, e' \in E_K \text{ with } e \neq e' \\
emb_V(v) &\notin [ext_H] && \forall v \in V_K \backslash [ext_K] \\
lab_K(e) &= lab_H(emb_E(e)) && \forall e \in E_K \\
emb_V(att_K(e)) &= att_H(emb_E(e)) && \forall e \in E_K \\
e \notin emb_E(E_K) &\Rightarrow [att_H(e)] \cap emb_V(V_K) = \emptyset && \forall e \in E_H,
\end{aligned}
$$

i.e. the mappings are injective, inner vertices of K are not mapped to external vertices in H and the functions *att* and *lab* are preserved.

We further define how such an embedding is replaced by a handle:

Definition 7 (Replace Mapping). Let $H, K \in \mathrm{HG}_{N \cup T}$ be hypergraphs, *emb* an embedding from K to H and $X \in N$ a non-terminal edge label. H' is isomorphic to $\mathit{replace}(K, H, \mathit{emb}, X^\bullet)$ if and only if

$$\begin{aligned} V_{H'} &= V_H \backslash \mathit{emb}_V(V_K \backslash [\mathit{ext}_K]) \\ E_{H'} &= (E_H \backslash \mathit{emb}_E(E_K)) \uplus \{e\},\ e \text{ being a fresh edge} \\ \mathit{lab}_{H'} &= (\mathit{lab}_H \restriction E_{H'}) \cup \{e \mapsto X\} \\ \mathit{att}_{H'} &= (\mathit{att}_H \restriction E_{H'}) \cup \{e \mapsto \mathit{emb}_V(\mathit{ext}_K)\} \\ \mathit{ext}_{H'} &= \mathit{ext}_H. \end{aligned}$$

For example the graph in Figure 1a (aside from the x-edge) is obtained by replacing an embedding of the second rule in the grammar in Figure 3 in the graph in Figure 2b.

With this notion, abstraction corresponds to reverse application of production rules, i.e. replacing an embedding by a correspondingly labelled handle.

Definition 8 (Abstraction). Let $G = (N, T, P)$ be an HRG, $H, H' \in \mathrm{HG}_{N \cup T}$ hypergraphs and $p : X \to K$ a rule in P. H' is obtained from H by abstraction with p, also written as $H \Leftarrow H'$, if and only if *emb* is an embedding of K in H and $H' \cong \mathit{replace}(K, H, \mathit{emb}, X^\bullet)$.

We now turn to the specific approach presented by Heinen et al.[11]:

3.2 Pointer Manipulations and Symbolic Execution

Heap structures such as lists and trees can be represented as hypergraphs, where a vertex is interpreted as an object and binary terminal edges represent references between them. We can further model program variables that point at some object by unary edges, labelled with the variable and adjacent to the respective vertex. One additional vertex represents the *null* reference, i.e. all references and variables that are assigned *null* point to this vertex that has no outgoing edges.

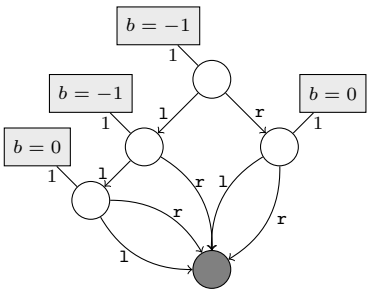


Fig. 5: A heap with data

If necessary, objects can have explicit data. This is realised via edge-labels, thus it is only possible if the range of values is finite (and should be very restricted in practice). Use cases include red/black-labels for red-black trees and balancing information in $\{-1, 0, 1\}$ for AVL trees.

This data is also modelled by a unary hyperedge with the appropriate label and adjacent to the vertex this data belongs to.

Figure 5 shows an example that makes use of this option. Each node except the null node is assigned its balancing. This is for example necessary for the AVL algorithm to determine where the balancing has to be re-established (for more details see Section 8.6).

Not all hypergraphs can be interpreted as valid heaps: For instance, if x is a variable-label, a graph where two edges are labelled with x cannot be valid since this would imply that x points to two objects at the same time. However, it is decidable whether a graph corresponds to a heap and it is also decidable if a given HRG produces only valid heaps [11].

For the following, we assume that we have a set of variables $Var = \{x_1, x_2, \dots\}$ of rank one, a set of references $Ref = \{s_1, s_2, \dots\}$ of rank two, and a set of data fields $Data = \{d_1, d_2, \dots\}$ with restricted domains Dom_1, Dom_2,... The set of terminal edge labels becomes thus $T = Var \cup Ref \cup \biguplus_i Dom_i$. For readability, we denote labels from each Dom_i in the form "$d_i = data$". Their rank is also one.

We can perform pointer manipulations on a graph representing a heap. Possible statements for this are $x = P$, $x.s = P$ and $new(x)$, where P is either *null*, x or $x.s$. Further, $x.d = v$ is a possible statement, if v is in the appropriate domain.

These statements transform the heap according to the usual semantics in common pointer based languages like Java. We give here two examples how pointer manipulation impacts the graph. The definitions for all other cases can be found in Heinen et al. [11].

For a hypergraph H, we write $H[\mu]$ to denote that the statement μ is applied to H.

$$H[x = null] = \begin{cases} H & \text{if } x \text{ references null} \\ (V, E\backslash\{e\}, att \restriction (E\backslash\{e\}), lab \restriction (E\backslash\{e\}), \epsilon) & \text{if } lab(e) = x \end{cases}$$

The formula states that, unless x already is *null*, the edge labelled with x is removed from the graph.

$$H[x.d_i = v] = \begin{cases} \text{undefined} & \text{if } x \text{ references null} \\ (V, (E\backslash\{e\}) \cup \{e'\}, att \restriction (E\backslash\{e\}) \cup \{e' \mapsto u\}, & \\ lab \restriction (E\backslash\{e\}) \cup \{e' \mapsto \text{"}d_i = v\text{"}\}), \epsilon) & \text{where } lab(e) \in Dom_i \\ \quad \text{and there is an edge } e_x \text{ labelled with } x \text{ such that } e \text{ and} & \\ \quad e_x \text{ are attached to the vertex } u.\ e' \text{ is a fresh edge.} & \end{cases}$$

Here, the previous d-edge on the vertex that x points to is replaced by one that is labelled with d and the correct value v.

During the symbolic execution on abstract heaps, it is important that all vertices directly accessible by the program are concrete. The program can access objects a variable points to and those referenced by this object. In the hypergraph representation, these are the vertices that are attached by outgoing edges.

Definition 9 (Outgoing Edge). For a vertex v, the edge e is called *outgoing* if $att(e, 1) = v$, i.e. if v is the first adjacent vertex to e.

A hypergraph $H \in \mathrm{HG}_{N \cup T}$ is called *admissible* if and only if $\mathcal{L}(H)$ contains only valid heaps and all vertices where a variable-labelled edge is pointing to have either no adjacent edges with non-terminal label, or the adjacent non-terminals cannot produce outgoing terminal edges to this vertex.

As long as a heap is admissible, it does not matter whether we first apply pointer manipulations or concretise it. This is formalized by the following lemma:

Lemma 1. *[11] For an admissible heap configuration $H \in HG_{N \cup T}$, a pointer manipulation $\mu \in \{x = P,\ x.s = P,\ new(x),\ x.d_i = v_i$, where P is either null, x or $x.s$ and $v_i \in Dom_i\}$ it holds that the set $\{H'[\mu] : H' \in \mathcal{L}(H)\}$ is the same as $\mathcal{L}(H[\mu])$.*

Proof. The lemma has been shown for pointer manipulations $x = P$, $x.s = P$, and $new(x)$, where P is either *null*, x or $x.s$ [11]. The case $x.d = v$ does not change this behaviour. By assuring an admissible heap it is clear that no production rule can create the same data-field to the affected vertex. Since all production rules are context-free, pointer manipulations and production rules are also commutable in this case. □

One has to distinguish between concrete and abstract semantics. The concrete semantics of a program in the context of this approach are the heaps in hypergraph representation obtained from applying the pointer manipulations to the input. For this approach, the abstract semantic is very similar, the only difference being that the hypergraph may now contain non-terminals to abstract parts of the heap and the semantic includes all hypergraphs that can be derived from the obtained non-terminal graph.

Independent from the concrete approach, it is necessary for correction that the abstract semantics over-approximate the concrete one, i.e. that the result of the concrete semantics is a subset of the abstract one. This means that any property shown for the abstract result holds in particular also for the concrete semantics.

4 Derivation Trees

Hyperedge replacement grammars are intuitively very similar to context-free string grammars and also all variants we are going to study have their counterpart as a string-replacement system. It is therefore possible to transfer at least some results from string grammars to the respective HRG.

In this section we establish that the emptiness problem for hyperedge replacement grammars can be reduced to the emptiness problem for string grammars. We show that for any HRG a corresponding CFG can be constructed, such that the sets of derivation trees contain isomorphic trees. The result then follows immediately since the language of a grammar is empty if and only if the set of derivation trees is empty.

In the following we use X_i to denote a non-terminal and α_i for terminal symbols. Furthermore, we use p_i for production rules of an HRG and p'_i for those of a CFG to keep them separated. Accordingly, we mark all other components of string grammars with a prime.

In the context of derivation trees we use trees in the form of terms, i.e. a tree consists of an n-ary function symbol (representing the root) with sub-trees as its arguments.

Definition 10 (Tree). Let a be an n-ary function symbol and $t_1, \ldots, t_n$ be trees (in term representation). Then $t = a(t_1, \ldots, t_n)$ is a tree with root a and sub-trees $t_1, \ldots, t_n$. Note, that the arity is zero if a is a leave.

Derivation Trees for HRGs To define derivation trees for HRGs, we require that edges with non-terminal label of a hypergraph are ordered in some arbitrary but fixed way. In particular, the non-terminal hyperedges on the right-hand side of production rules are ordered and this order is preserved when applying the rule.

Definition 11 (Derivation Tree). [6] Let $G = (N, T, P)$ be a context-free HRG. Then $t = p(t_1, \ldots, t_n)$ is a *derivation tree* for non-terminal X if and only if $p \in P$ is a production rule of the form $X \to H \in HG_{N \cup T}$ and for each $i \in \{1, \ldots, n\}$, t_i is a derivation tree for the label of the ith non-terminal edge of H. Leaves are thus labelled with rules where the right-hand side is terminal.

The set of all derivation trees for a non-terminal X in G is denoted by $D(G, X)$.

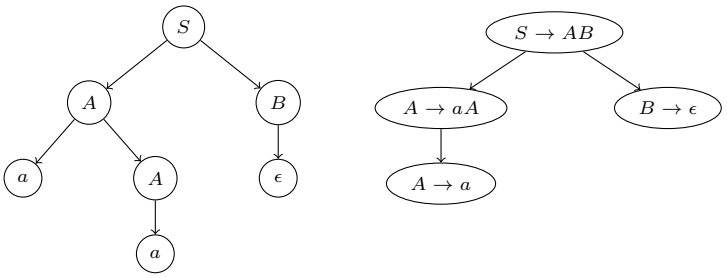
Each derivation tree $t \in D(G, S)$ corresponds to a hypergraph $K \in \mathcal{L}(S^\circ)$, which is derived from S° by applying the productions as specified in the tree. We express this formally by the function $\mathit{yield} : D(G, S) \to \mathcal{L}(X^\circ)$ that is defined recursively as follows:

Definition 12 (Function yield). [6] The hypergraph corresponding to the derivation tree $t \in D(G)$ is

$$\mathit{yield}(p(t_1, \ldots, t_n)) = \mathit{rhs}(p)[\mathit{yield}(t_1)/e_1, \ldots, \mathit{yield}(t_n)/e_n],$$

where $e_1, \ldots, e_n$ are the (ordered) non-terminal hyperedges of $\mathit{rhs}(p)$.

We see immediately that the language of an HRG G with initial graph S° is non-empty, iff the set of derivation trees $D(G, S) \neq \emptyset$ [6], since for each hypergraph in the language there is a derivation tree and vice versa. For a hypergraph $H \in \mathrm{HG}_{N \cup T}$ the language $\mathcal{L}(H)$ with respect to G is non-empty if $\mathcal{L}(S^\circ) \neq \emptyset$ for all non-terminals X occurring in H.



(a) Common derivation tree **(b)** Rule labelled derivation tree

Fig. 6: Derivation Trees for the word aa

Derivation Trees for CFGs In the literature (see Hopcroft et al. [12]), derivation trees are usually defined in a slightly different way than presented here: This definition allows CFG derivation trees to be more conveniently compared to the derivation trees of HRGs. To avoid confusion, we call our variation *rule labelled* derivation trees and the others *common* ones.
We first recall the traditional definition of derivation trees for CFGs:

Definition 13 (Derivation Tree for CFG). [12] A tree t is a *(common) derivation Tree* of a CFG $G' = (N', T', P')$ if all inner nodes are labelled with non-terminals $X' \in N'$, all leaves are labelled with terminals $\alpha' \in T'$ or the empty sequence ϵ and if $\alpha'_1 \dots \alpha'_{k_1} X'_1 \alpha'_{k_1+1} \dots X'_n \alpha'_{k_n+1} \dots \alpha'_{k_{n+1}}$ are the labels of the children of a vertex with label X' from left to right, then there is a rule $p : X' \to \alpha'_1 \dots \alpha'_{k_1} X_1 \alpha'_{k_1+1} \dots X'_n \alpha'_{k_n+1} \dots \alpha'_{k_{n+1}} \in P'$. Further a node labelled with ϵ must be the only child of some node labelled with X' and there must be a rule $X \to \epsilon \in P'$. The word corresponding to this tree is obtained when reading the labels of the leaves from left to right.

Instead of labelling the nodes with symbols from the alphabet, rule labelled derivation trees use instead the rules themselves:

Definition 14 (Rule Labelled Derivation Tree). Let $G' = (N', T', P')$ be a context-free string grammar. The *rule labelled derivation trees* for $X' \in N'$ are all trees $p'(t'_1, \dots, t'_n)$ where $p' \in P'$, $lhs(p') = X'$ and for $1 \leq i \leq n$: t'_i is a derivation tree for X'_i (the ith non-terminal symbol of $rhs(p')$).

Rule labelled derivation trees have the advantage that we can see immediately which rules have been applied. This simplifies notation when we apply them to derivation controlled grammars. On the other hand, it is slightly less straightforward to construct the derived word. Let us shortly define how to extract the word from the tree:

Definition 15. The word corresponding to a rule labelled derivation tree t is defined inductively. For a leaf labelled by $p \in P$, $w(t)$ is $rhs(p)$ (As it was a

leaf, the *rhs* of p must consist of terminals only). For a tree $t = p(t_1, \dots t_n)$: $w(t) = rhs(p)[w(t_1)/X_1, \dots, w(t_n)/X_n]$.

Figure 6 shows a short example for both kinds of derivation trees. They are based on a context-free grammar with the following rules:

$$\begin{array}{ll} S \to AB & \\ A \to aA & B \to bB \\ A \to a & B \to \epsilon \end{array}$$

We can observe that the rule labelled derivation tree uses one layer less, because the terminals are already encoded in the rules.

Lemma 2. *Rule labelled derivation trees and common derivation trees for CFGs are equivalent, i.e. they can be transformed into each other, such that the corresponding word is the same.*

Proof. We start with the proof by showing that each common derivation tree can be transformed in a rule labelled one by an induction on the depth of the derivation tree:

IB: (depth = 1) Let $X \in N$ be the label of the root and $\alpha_1, \dots, \alpha_n \in T$ the labels of the leaves. Then there is a rule $p : X \to \alpha_1 \dots \alpha_n \in P$. The corresponding rule labelled derivation tree is p. The corresponding word is in both cases $\alpha_1 \dots \alpha_n$.

IH: Derivation trees of depth $d < n$ can be transformed to rule labelled derivation trees, such that the corresponding word is the same.

IS: Let t be a derivation tree of depth n. Let $X \in N$ be the label of the root and $\alpha_1, \dots \alpha_{k_1}, X_1, \alpha_{k_1+1}, \dots, X_n, \alpha_{k_n+1}, \dots, \alpha_{k_{n+1}}$ ($X_i \in N$ and $\alpha_j \in T$) its children. Then there is a rule

$$p : X \to \alpha_1 \dots \alpha_{k_1} X_1 \alpha_{k_1+1} \dots X_n \alpha_{k_n+1} \dots \alpha_{k_{n+1}}.$$

Further, all nodes labelled by some $\alpha_j \in T$ are leaves and all labelled by some $X_j \in N$ are the roots of sub trees t_i of depth smaller than n. By induction hypothesis, those sub trees can be transformed to rule labelled ones such that the corresponding word is preserved. As the word corresponding to a derivation tree t with root p is the concatenation of the terminal segments and the words corresponding to its sub-trees

$$\begin{aligned} &w(p(t_1, \dots, t_n)) = rhs(p)[X_1/w(t_1), \dots X_n/w(t_n)]) \\ &= \alpha_1, \dots \alpha_{k_1}, w(t_1), \alpha_{k_1+1}, \dots, w(t_n), \alpha_{k_n+1}, \dots, \alpha_{k_{n+1}}. \end{aligned}$$

Hence, by induction hypothesis the words are the same.

The other direction is a similar induction, but note that the induction base is now depth 0 since the leaves labelled by terminals in the common derivation trees are implicitly contained in the rules.

IB: (depth = 0) Let p be the label of a rule labelled derivation tree of depth 0. Then $p \in P$ is a rule of form $X \to \alpha_1 \dots \alpha_n, \alpha_i \in T$. This corresponds to a common derivation tree with root X and children $\alpha_1, \dots, \alpha_n$. The corresponding word is for both $\alpha_1 \dots \alpha_n$.

IH: Rule labelled derivation trees for CFG G of depth $d < n$ can be transformed to common derivation trees for G such that the corresponding word is the same.

IS: Let $p(t_1, \dots, t_n)$ be a rule labelled derivation tree for the CFG G of depth n, where p is a production rule of G in the form

$$p : X \to \alpha_1 \dots \alpha_{k_1} X_1 \alpha_{k_1+1} \dots X_n \alpha_{k_n+1} \dots \alpha_{k_{n+1}}.$$

Then the corresponding common tree consists of the root X with sub-trees α_i that are leaves, while the t_is are transformed according the the induction hypothesis to sub-trees with root X_i.

□

Theorem 1. *For any context-free HRG G a CFG G' can be constructed in time and space $\mathcal{O}(|P| \cdot max_{p \in P}\{rhs(p)\})$, such that $D(G, X) = D(G', X')$, up to node renaming, for all non-terminals X in G.*

Proof. We first construct for each HRG a suitable CFG and then show that the corresponding derivation trees are isomorphic.

Contruction: Let $G = (N, T, P)$ be an HRG. We construct the CFG $G' = (N, \{a\}, P')$. For all non-terminals $X \in N$ we ignore the *type* when using them in G' and treat them just as simple non-terminal symbols. For our construction we require just one terminal symbol a. For all rules $p : X \to K \in \mathrm{HG}_{N \cup T}$ let $X_1, \dots, X_n$ be the set of non-terminals appearing in K in some arbitrary but fixed order (this order was specified when defining derivation trees for HRGs). We add the rule $p' : X \to X_1 \dots X_n$ to P'. For rules $p : X \to K \in \mathrm{HG}_T$ we add $p' : X \to a$ instead.

Remark 1. By this construction, different rules of G can be mapped to identical rules in G', but they are still distinguished rules. For instance, $p_1 : X \to K \in \mathrm{HG}_T$ is mapped to $p'_1 : X \to a$ and $p_2 : X \to K' \in \mathrm{HG}_T$ is mapped to $p'_2 : X \to a$. This is of importance as we apply this theorem to automata controlled HRGs in Section 6, where the derivation sequence is controlled by a regular language over the set of rules.

Complexity: We have to consider each rule in the original HRG. The construction of the new rule runs once over the right-hand side of the rule, transforming each non-terminal edge to a non-terminal symbol and discarding everything else. Therefore, the construction takes time and space $\mathcal{O}(|P| \cdot max_{p \in P}\{rhs(p)\})$ and is thus linear in the size of the HRG.

Correctness: We now show for each non-terminal X in N that $D(G, X) = D(G', X')$. For the first direction let t be a derivation tree in $D(G, X)$ for some $X \in N$. Induction on the depth of t:

IB: (*depth* $= 0$) If t has depth 0 its root (and only node) is labelled by some rule $p \in P$ of the form $p : X \to K \in HG_T$. By construction we have a rule $p' : X \to a \in P'$ and the tree with a single node labelled by p' is in $D(G', X)$ and isomorphic to t (up to renaming).

IH: For all $X \in N$ we have that for all trees of depth at most n in $D(G, X)$ there is an isomorphic tree in $D(G', X)$.

IS: Let $t = p(t_1, \ldots, t_n) \in D(G, X)$ be a derivation tree of depth n. Then p is of the form $X \to H \in HG_{N \cup T}$, $X_1, \ldots, X_n$ is the sequence of non-terminal edges in H and $t_i \in D(G, X_i)$ for all $i \in \{1, \ldots, n\}$. By induction hypothesis, for each i between 1 and n there is a tree t'_i isomorphic to t_i in $D(G', X_i)$ and by construction there is a production rule $p' : X \to X_1 \ldots X_n$ in P'. Thus, $p'(t'_1, \ldots, t'_n)$ which is isomorphic to t, is in $D(G', X)$.

For the other direction, let t' be a derivation tree in $D(G', X)$ for some $X \in N$. We again proceed by an induction on the depth:

IB: (*depth* $= 0$) If t' has depth 0, its root (and only node) is labelled by some rule $p' \in P'$ of the form $p' : X \to a$. Then by construction there is a rule $p : X \to H \in HG_T$ and the tree with a single node labelled by p is in $D(G, X)$ and isomorphic to t' (up to renaming).

IH: For all $X \in N$ and for all trees of depth at most n in $D(G', X)$ there is an isomorphic tree in $D(G, X)$.

IS: Let $t' = p'(t'_1, \ldots, t'_n) \in D(G', X)$ be a derivation tree of depth n. Then p' is of the form $p' : X \to X_1 \ldots X_n$ in P' and $t_i \in D(G', X_i)$ for all $i \in \{1, \ldots, n\}$. By induction hypothesis, for each i between 1 and n there is a tree t_i isomorphic to t'_i in $D(G, X_i)$ and by construction there is a production rule $X \to H \in HG_{N \cup T}$, with $X_1, \ldots, X_n$ being the sequence of non-terminal edges in H. Thus, $p(t_1, \ldots, t_n)$ which is isomorphic to t' is in $D(G, X)$.

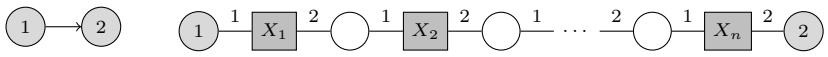
□

The converse direction holds as well:

Theorem 2. *For each CFG G' one can construct an HRG G in time and space* $\mathcal{O}(|P| \cdot max_{p \in P}\{rhs(p)\})$ *such that the corresponding derivation trees are isomorphic.*

Proof. For this case we only discuss the construction since the proof of correctness is very similar to the one for the previous theorem.

Construction: We consider two kinds of graphs given in Figure 7. The first consists of two connected vertices that are external (see Figure 7a). We refer to it as K_t and use it for all kinds of terminal sequences. The second consists of n non-terminal edges $e_1, \ldots, e_n$ labelled by $X_1, \ldots, X_n$ and $n+1$ vertices between them, with the ones at the very left and very right being external (see Figure 7b). We set the ordering of non-terminal edges from left to right. For rules in the CFG of the form $p' : X \to \alpha_1 \ldots X_1 \ldots X_n \ldots \alpha_n$ we add a rule $p : X \to K_n$ to the



(a) K_t **(b)** Linear graph with n non-terminal edges

Fig. 7: The two kinds of graphs used to construct an HRG from a CFG

HRG where K_n is as described above and in Figure 7b. For terminal rules we add a rule $p : X \to K_t$. Remark 1 holds also here.

Complexity: We have to consider each rule in the original CFG. The construction of the new rule runs once over the right-hand side of the rule, transforming non-terminal symbols to edges and discarding terminal ones. Therefore, the construction takes time and space $\mathcal{O}(|P| \cdot max_{p \in P}\{rhs(p)\})$ and is thus linear in the size of the CFG.

Correctness: The rank of the non-terminals agrees with the type of the hypergraphs. The constructed HRG is therefore valid. Furthermore, all constructed rules have the same sequence of non-terminals as their counterpart. Analogous to Theorem 1, the corresponding derivation trees are isomorphic. □

5 Controlled Grammars: An Overview

In the following chapters we analyse three different variations of HRGs that in some way control the derivation. All three variations have already been introduced for string grammars and can be found in [5].

In each case, after defining the respective variation, we first show that emptiness is decidable which is mainly based on the decidability for the string case and the results from Section 4 on the equivalence of derivation trees to lift this to HRGs. The decidability of emptiness is important in the context of heap abstraction, because it is possible that grammars produce 'dead ends', i.e. derivation sequences that never lead to a concrete hypergraph. In the context of heap abstraction we use the grammar to alternately concretise and abstract graphs, Thus, we have to decide whether a graph is part of a productive derivation sequence. i.e. whether a terminal graph can be derived from it. The possibility to decide emptiness provides us with an appropriate test.

A main idea in the approach of Heinen et al. to ensure that the state space is finite is to use the given grammars also for abstraction. Therefore, we give for each variation an abstraction mechanism and show that it yields an over-approximation and is therefore safe. For this we use the following result from Heinen et al.:

Theorem 3 (Correctness). *[11] Let $H \in HG_{N \cup T}$ be a valid heap configuration. For concretisation function con with $\mathcal{L}(H) = \bigcup_{H' \in con(H)} \mathcal{L}(H')$ and abstraction function abs, such that $\forall H' \in abs(H) : \mathcal{L}(H) \subset \mathcal{L}(H')$, the abstract semantic is an over-approximation of the concrete one.*

We also consider whether the proposed abstraction mechanism is powerful enough to serve the objective of obtaining a finite state space.

Each section will be concluded with some further remarks on their usability for heap abstraction.

6 Automata Controlled Grammars

The first variant we introduce are so called regular controlled grammars in a variation we call *automata controlled grammars*.

A regular controlled HRG extends hyperedge replacement grammars by a regular set $R \subseteq P^*$ of permitted derivation sequences. That is, a derivation $H \Rightarrow_{p_1} ... \Rightarrow_{p_n} H'$ is valid if and only if the sequence of applied rules $p_1...p_n$ is contained in the set R.

Regular controlled grammars have been studied for the string case [4, Chapter 1], where they are quite interesting as they can express several languages not expressible with context-free grammars. Kreowski has transferred them to HRGs [14]. But here, it is not yet clear whether regular controlled HRGs have more expressive power than context-free ones. This might at first be somewhat surprising, but it turns out, that the typical examples of non-context-free string languages that can be defined by regular controlled grammars, e.g. $\{a^n b^n c^n \mid n \in \mathbb{N}\}$, can also be defined by context-free string generating HRGs [8, chapter V]. However, they do have different properties and we analyse in the following section whether that makes them useful for heap abstraction.

6.1 Definitions

An automata controlled HRG uses a DFA (see Appendix A) to define which derivations are valid:

Definition 16 (Automata controlled grammar). An *automata controlled HRG* is a tuple $G = (N, T, P, \mathcal{A})$, where $\mathcal{A} = (Q, P, \delta, F)$ is a DFA over the alphabet P (i.e. the set of rules) and the other components are as in Definition 3.

The idea is that derivation sequences are valid if accepted by $\mathcal{A}$. Since DFAs define exactly the regular sets, these grammars have the same expressive power as regular controlled ones. Their advantage is that the automaton can be executed in parallel to the derivation, i.e. we consider pairs of (non-terminal) derived hypergraphs and automaton states, which is important when we define the abstraction mechanism.

Definition 17 (Derivation). Let $G = (N, T, P, \mathcal{A})$ be an automaton controlled grammar, $H, H' \in \mathrm{HG}_{N \cup T}$ hypergraphs and $q, q' \in Q$ states in the automaton. We have $(H, q) \Rightarrow_p (H', q')$ for a rule $p \in P$ if and only if $H' \Rightarrow_p H$ by application of p and $\delta(q, p) = q'$.
The transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$. The set of pairs (H', q') that are derivable from (H, q) is $con(H, q) = \{(H', q') \mid (H, q) \Rightarrow^* (H', q')\}$.

The rule application on the graphs themselves are in principle the same as in context-free grammars (see Definition 2) and the automaton is executed in parallel, but rule p can only be applied in a given state q, if the transition function $\delta(q, p)$ is defined (Since it is a partial function this is not necessarily the case for all combinations). Note that the mapping from graphs to states is not unique since a graph may be reached by different derivation sequences.

The language of such a grammar starting from a graph H and a state q is defined as follows:

Definition 18 (Language). Let $G = (N, T, P, \mathcal{A})$ be an automaton controlled grammar, $H \in \mathrm{HG}_{N \cup T}$ a hypergraph and q a state in $\mathcal{A}$. The *language* $\mathcal{L}(H, q)$ is the set of all graphs $K \in \mathrm{HG}_T$ such that there is a derivation starting from H and q, obtaining K and reaching a final state in $\mathcal{A}$. Thus,

$$\begin{aligned}\mathcal{L}(H, q) &= \{K \in \mathrm{HG}_T \mid (H, q) \Rightarrow^* (K, q^*) \text{ with } q^* \in F\} \\ &= \{K \in \mathrm{HG}_T \mid \exists w = p_1 ... p_n \in P^*, \mathcal{A}_q \text{ accepts } w \text{ and } H \Rightarrow_{p_1} ... \Rightarrow_{p_n} K\}.\end{aligned}$$

6.2 Emptiness

As explained in Section 5, the decidability of emptiness is an important property in the context of heap abstraction. For automata controlled HRGs the emptiness problem is decidable although it suffers from a bad complexity.

Theorem 4. *For an automaton controlled HRG G and a hypergraph H it is decidable whether $\mathcal{L}(H) = \emptyset$, but NP hard.*

Remark 2. The only known algorithm for this problem is even in EXP-TIME [5].

Proof. Decidability: When the automaton of an automata controlled HRG is ignored, it is equivalent to a context-free HRG. For those the emptiness problem is reduced to the emptiness of string grammars according to Theorem 1 since they can be constructed in a way such that the set of derivation trees is equivalent. For any given derivation tree it can be checked if there is at least one possible derivation encoded in this tree that is accepted in the automaton (the set of encoded derivations is the set of all permutations of node-labels that respect the partial order given by the tree, i.e. the rule-label of some node can only appear in the sequence after the rule labelling its parent). This check has equal results on the derivation trees of HRGs and CFGs (if the renaming of rules is applied to the alphabet of the automaton). The set of valid derivation trees with respect to the controlling automaton is therefore empty for the automata controlled HRGs if and only if it is empty for the automata controlled HRGs. Since emptiness is decidable for regular controlled string grammars, it is also decidable for automata controlled HRGs.

Complexity: The emptiness problem for regular controlled string grammars is NP-hard [5]. We have EMPTY(CFG) $\leq_P$ EMPTY(HRG), since a corresponding grammar with isomorphic derivation trees whose language is empty

if and only if the originals' language is empty can be constructed in $\mathcal{O}(|P|)$ - Theorem 2. Therefore, the emptiness problem for automata controlled HRGs is also NP-hard. □

6.3 Abstraction

A central aspect of the symbolic execution proposed by Heinen et al. [11] is the use of HRGs for abstraction. In contrast to context-free HRGs, regular controlled HRGs have more than one reasonable abstraction mechanism. In this thesis, we introduce and analyse one possibility where we see the potential that it is useful to ensure backward-confluence, i.e. to ensure that the abstraction has a unique result. Here, each abstraction consists of the backwards application of a rule and a backwards step in the associated automaton, i.e. we abstract a graph H with a rule p by using an incoming p-edge in the current automaton state backwards and by embedding and replacing the left-hand side of the rule.

Definition 19 (Abstraction). Let $G = (N, T, P, \mathcal{A})$ be an automaton controlled grammar, $H, H' \in \mathrm{HG}_{N \cup T}$ hypergraphs, q, q' states of $\mathcal{A}$ and $p : X \to K$ a rule in P. (H', q') is obtained from (H, q) by abstraction with p, also written as $(H, q) \Leftarrow_p (H', q')$, if and only if $\delta(q', p) = q$, *emb* is an embedding of K in H and $H' \cong \mathit{replace}(K, H, \mathit{emb}, X^\bullet)$. We define $\mathit{abs}(H, q) = \{(H', q') \mid \exists p_1 ... p_n \in P^+ : (H, q) \Leftarrow_{p_1} ... \Leftarrow_{p_n} (H', q')\}$, the set of pairs (H', q') that can be abstracted from (H, q) in at least one step.

This abstraction mechanism on the one hand reduces the set of production rules that can be used for abstraction, since in a given state there are not necessarily incoming edges for all production rules. On the other hand the number of possible abstraction results in one step is also increased, as there might be more than one incoming edge labelled with p (the backwards direction of an DFA is not necessarily deterministic). It thus depends strongly on the chosen automaton whether backward-confluence is influenced in a positive or negative way.

Example Before we proof the correctness of this abstraction mechanism, we illustrate in a short example how the approach can be used.

When we use hyperedge replacement grammars for heap abstraction we want to abstract the heaps as much as possible. A heap is called *fully abstracted* if it cannot be further abstracted. It is very inconvenient if we can obtain different fully abstracted graphs from one input graph by the choice of the embeddings, especially if the number of different graphs is very large or even infinite.

Definition 20 (backward-confluence). [11] An HRG G is called *backward-confluent* if the set of fully abstracted graphs

$$\mathit{fullAbstr}(H)_G = \{K \in \mathrm{HG}_{N \cup T} \mid K \Rightarrow^* H \wedge \forall X \to I \in G : \mathit{emb}(I, K) = \emptyset\}$$

is a singleton for every $H \in \mathrm{HG}_{N \cup T}$.

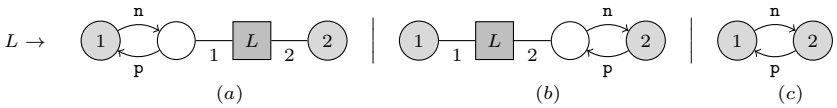
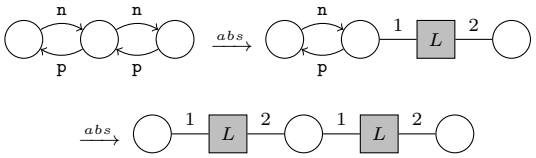


Fig. 8: Grammar for doubly-linked lists

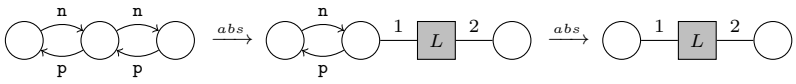
It is therefore desirable that HRGs are backward-confluent. The following example shows that automaton controlled HRGs can help to reach this property:

Figure 8 shows an HRG for doubly-linked lists. Unfortunately, it is not backward-confluent as Figure 9 shows: In one case (Figure 9a) the second rule is embedded twice so that the resulting graph consists of two adjacent L-edges. Since the grammar does not provide any rules with a right-hand side like this, the abstraction terminates at this point. In the other case (Figure 9b), rule c is used only once. This corresponds better to the way derivations of this grammar happen which is why one might perceive it as more 'correct' (though of course both variants are by definition correct abstractions). As a result, we have abstracted the given graph to two different fully abstract graphs, which is something we would like to avoid.

Usually this problem is solved by introducing an additional rule that consists exactly of the two adjacent L-edges (the lower graph in Figure 9a), so that we can add an additional derivation step leading to the last graph in Figure 9b. Thus, backward-confluence is ensured.



(a) The 'wrong' abstraction



(b) The 'correct' abstraction

Fig. 9: Illustration why the HRG in Figure 8 is not backward-confluent

Regular controlled HRGs provide us with a second possibility to ensure backward-confluence. We can specify the controlling regular language as an au-

tomaton and accompany each abstraction step by a back-step in this automaton. Figure 10 shows an automaton that fulfils this task for the given grammar.

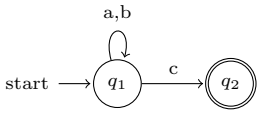


Fig. 10: Automaton that makes the grammar in Figure 8 backward-confluent

We see that the problematic abstraction in Figure 9 is no longer possible. In this simple example it is easy to see that the regular controlled HRG does not concretise in any other way than a context-free HRG consisting of the same rules (as rule c terminates the derivation anyway). But when analysing the abstraction steps in Figure 9 we get the following result: Initially we are in state q_2 (since the initial graph is terminal we used rule c and thus reached the accepting state). The first abstraction step is still possible, but leaves us in state q_1 (having used the c-edge backwards). In this state one may use only a- and b-edges for abstraction. This way the abstraction is backward-confluent without any new production rules added.

Correctness We now proceed by showing the correctness of the abstraction mechanism, i.e. by proving that it yields an over-approximation. For this we use Theorem 3. In the following we show their prerequisites.

First we show that pointer manipulations and derivation steps are commutable:

Lemma 3. *For an automaton controlled HRG G, an admissible heap configuration $H \in HG_{N \cup T}$ and a pointer manipulation $\mu \in \{x = P,\ x.s = P,\ new(x),\ x.d_i = v_i$, where P is either null, x or $x.s$ and $v_i \in Dom_i\}$ it holds that the set $\{H'[\mu] : H' \in \mathcal{L}(H)\}$ is the same as $\mathcal{L}(H[\mu])$, i.e. Lemma 1 holds also for automata controlled HRGs.*

Proof. The lemma holds for context-free HRGs, so we only argue that it does not interfere with the automaton. The automaton restricts the number of applicable rules but this is independent from any changes a pointer manipulation might apply to concrete parts of the heap. Further, it discards terminal graphs if it has not reached a final state. Since the derivation sequence is not influenced by pointer manipulations, this is again independent. Therefore, the lemma can be extended to automata controlled HRGs. □

Now we proceed to the main theorem:

Theorem 5. *Concretisation as defined in Definition 17 and Abstraction as defined in Definition 19 yields an over-approximation.*

Proof. Theorem 3 states that any concretisation function (con) and abstraction function (abs) where $\mathcal{L}(H) = \bigcup_{H' \in con(H)} \mathcal{L}(H')$ and $\mathcal{L}(H) \subseteq \mathcal{L}(H')\ \forall H' \in abs(H)$ yield such an over-approximation [11].

Let us show that those two properties hold:

1. $\mathcal{L}(H,q) = \bigcup_{(H',q') \in con(H,q)} \mathcal{L}(H',q')$:
   We start with the left-hand side of the equation, insert the definitions and concatenate the two sequences:

   $$\bigcup_{(H',q') \in con(H,q)} \mathcal{L}(H',q')$$
   $$= \{K \in \mathrm{HG}_T \mid \exists v = p_1 ... p_k \; \exists H' \in \mathrm{HG}_{N \cup T} \; \exists v' = p_{k+1} ... p_n :$$
   $$\delta^*(q,v) = q', \mathcal{A}_{q'} \text{ accepts } v' \text{ and } H \Rightarrow_{p_1} ... \Rightarrow_{p_k} H' \Rightarrow_{p_{k+1}} ... \Rightarrow_{p_n} K\}$$

   The intermediate graph H' can be omitted and from $\delta^*(q,v) = q'$ and $\mathcal{A}_{q'}$ accepts v' follows that $\mathcal{A}_q$ accepts $w = vv'$, yielding:

   $$\{K \mid \exists w = vv' = p_1 ... p_k p_{k+1} ... p_n : \mathcal{A}_q \text{ accepts } w \text{ and } H \Rightarrow_{p_1} ... \Rightarrow_{p_n} K\}$$

   This is equivalent to $\mathcal{L}(H,q)$.
2. Instead of $\mathcal{L}(H) \subseteq \mathcal{L}(H') \; \forall H' \in abs(H)$ we show that $(H,q) \Rightarrow (H',q')$ if and only if $(H',q') \Leftarrow (H,q)$, which is a stronger statement.
   Let $H, H' \in \mathrm{HG}_{N \cup T}$ be hypergraphs, q, q' states in the controlling automaton and $p \in P$ a production rule. It holds that $H \Rightarrow_p H'$ if and only if $H' \Leftarrow_p H$ [11]. Furthermore, both $(H,q) \Rightarrow_p (H',q')$ and $(H',q') \Leftarrow_p (H,q)$ imply and require $\delta(q,p) = q'$. Therefore, the equality is shown.

□

6.4 Discussion

Automata controlled HRGs seem to preserve some of the important properties of HRGs needed for heap abstraction. We have also seen in an example that they can be used to ensure backward-confluence of a given grammar by providing an appropriate automaton. It would be interesting to study whether this is possible for any HRG, especially those where backward-confluence cannot be established by simply introducing new rules. On the other hand the emptiness problem is NP-hard and it is further not possible to prevent unproductive branches by eliminating unproductive rules beforehand. Thus, verification using automata controlled HRGs to extend the approach by Heinen et al. [11] would probably consume more time resources than one with ordinary HRGs.

7 Indian Parallel Grammars

Indian parallel HRGs are one possibility to express full trees. They are transferred from Indian parallel string grammars[5,14]. The mechanism of hyperedge replacement is again the same as for context-free HRGs, but has the additional restriction, that all edges labelled by the same non-terminal have to be replaced in parallel, i.e. if a rule $p : X \to H$ is applied, it has to be applied to all occurrences of X.

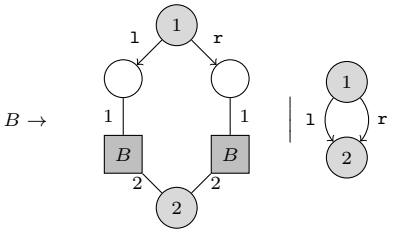


Fig. 11: Complete Binary Tree Grammar

Technically there is no difference between replacing edges in parallel or replacing them one after another, but the word parallel stresses that all replaced edges were contained in the original hypergraph.

Figure 12 shows an example derivation of the grammar in Figure 11. In each step all occurrences of B-edges are replaced using the first rule. As a result in each step all paths from the root have the same length. To terminate the derivation one would at some point replace all occurrences of B with the second rule, yielding a complete tree. As it is an Indian parallel grammar, it is not possible to terminate one of the paths with the second rule but leave others for further extension with the first rule.

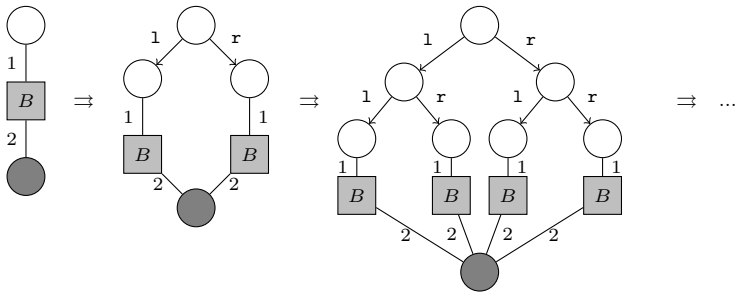


Fig. 12: Example Derivation

7.1 Definitions

In the following we give the formal definition for derivations and languages of Indian parallel grammars:

Definition 21 (Derivation). Let $G = (N, T, P)$ be an Indian parallel Grammar and $H, H' \in \mathrm{HG}_{N \cup T}$ be two hypergraphs. H' is derived from H, iff there is a rule $p : X \to H \in P$ such that H' is isomorphic to $H[K/e_1, \ldots, K/e_n] = (\ldots((H[K/e_1])[K/e_2])\ldots)[K/e_n]$, where $\{e_1, \ldots, e_n\}$ is the set of edges in H labelled with X. We denote by $H \rightrightarrows H'$ that H' is derived from H by an Indian parallel derivation. As usual $\rightrightarrows^*$ is the transitive and reflexive closure of $\rightrightarrows$. Furthermore, $con(H) = \{H' \in \mathrm{HG}_{N \cup T} \mid H \rightrightarrows^* H'\}$ is the set of all hypergraphs derivable from H.

The language of an Indian parallel grammar is defined as the set of all terminal graphs that can be derived from H in accordance to the Indian parallel mechanism.

Definition 22 (Language). Let $G = (N, T, P)$ be an Indian parallel grammar and $H \in \mathrm{HG}_{N \cup T}$ a hypergraph. The *language* of H with respect to G is $\mathcal{L}(H) = \{K \in \mathrm{HG}_T \mid H \rightrightarrows^* K\}$.

7.2 Emptiness

Emptiness for Indian parallel grammars is as difficult to decide as emptiness for common context-free grammars, because if there is at least one valid context-free derivation of a grammar, there is also one which uses always the same rule for each non-terminal and can thus be interpreted as parallel.

Theorem 6. *The emptiness problem for Indian parallel grammars is decidable, i.e. for each Indian parallel grammar G and axiom hypergraph $H \in HG_{N \cup T}$ it is decidable whether $\mathcal{L}(H) = \emptyset$ with respect to G, in time $\mathcal{O}(|N|^2 \cdot |P|)$.*

Proof. We proof the above theorem by showing that the marking algorithm (introduced by Bar-Hillel et al. in [2]) that decides emptiness for context-free (string) grammars is also correct for Indian parallel HRGs.

The marking algorithm constructs iteratively a set M that contains all non-terminals X so that a terminal graph (or string respectively) can be derived.

1. $M^0 = \emptyset$
2. $M^{i+1} = \{X \mid \exists p : X \to H : \forall Y \text{ with } \exists e \in H : lab(e) = Y \Rightarrow Y \in M^i\}$
3. If $M^k = M^{k+1}$ set $M = M^k$

By this we add in each step all non-terminals such that there is a rule applicable to this non-terminal and all non-terminals on its right-hand side are already marked, beginning with rules where the right-hand side is terminal in the first step. We claim that $\mathcal{L}(H)$ is non-empty with respect to G for some $H \in \mathrm{HG}_{N \cup T}$ if and only if there is a $k \in \mathbb{N}$ such that all non-terminals appearing in H are in M^k.

Correctness: We show that $\mathcal{L}(H) \neq \emptyset$ implies that all non-terminals appearing in H are in M and that, if all non-terminals appearing in H are in M, then $\mathcal{L}(H) \neq \emptyset$. Indian parallel grammars are a special case of context-free grammars, i.e. if $\mathcal{L}(H) \neq \emptyset$ with respect to an Indian parallel grammar G then the language is still not empty if the grammar is interpreted in the "normal" (non parallel) way. Since the algorithm is correct for context-free grammars [2] the first implication holds.

For the converse direction we use that the algorithm can be interpreted in a way that each (productive) non-terminal is assigned a rule, so that any derivation that applies only these assigned rules, yields a terminal graph. We assume that

there is some $k \in \mathbb{N}$ such that all non-terminals occurring in H are in M^k and show by an induction on k that $\mathcal{L}(H) \neq \emptyset$:
First we define GM^i as the set of hypergraphs K such that all non-terminals occurring in K are in M^i (GM corresponds to M in the same way). We show by induction that $H \in GM \implies \mathcal{L}(H) \neq \emptyset$:

IB: Let H be a hypergraph in GM^0. Then H must be terminal since M^0 is empty and thus $\mathcal{L}(H) = \{H\} \neq \emptyset$.
IH: If H is in GM^k then $\mathcal{L}(H) \neq \emptyset$.
IS: For a hypergraph H in GM^{k+1} we have the set $\{X_1, \ldots, X_n\}$ of non-terminals that are in $M^{k+1} \backslash M^k$. By induction on n we show that there is an Indian parallel derivation $H \rightrightarrows^* H'$ such that $H' \in GM^k$:
  IB: For $n = 1$ we have a rule $p : X_1 \to K$ such that $K \in GM^k$ otherwise X_1 would not be in M^{k+1}. If we apply p to all occurrences of X_1 we have a valid Indian parallel derivation and $H \rightrightarrows_p H'$ with $H' \in GM^k$.
  IH: If the number of non-terminals in H and in $M^{k+1} \backslash M^k$ is n, there is an Indian parallel derivation $H \rightrightarrows^* H'$ such that $H' \in GM^k$.
  IS: As X_{n+1} appears in M^{k+1} we have a rule $p : X_n \to K$ with $K \in GM^k$. Therefore, the application of p to all edges in H labelled with X_n $H \rightrightarrows_p H''$ is a valid Indian parallel derivation, H'' is in GM^{k+1} and the set of non-terminals in H'' and $M^{k+1} \backslash M^k$ is $\{X_1, \ldots, X_n\}$.

Therefore, we have shown that $\mathcal{L}(H) \neq \emptyset$ for all H in GM.

Complexity: M^{i+1} can be constructed from M^i in time $\mathcal{O}(|P| \cdot |N|)$, since for each rule we have to check for all non-terminals appearing on the right-hand side whether they are in M^i. The algorithm terminates if no new non-terminal is added, i.e. if $M^{k+1} = M^k$ for some k. As M increases in each step by at least one and its maximal size is $|N|$ (if all non-terminals are productive), it follows that the algorithm terminates after at most $|N|$ iterations. □

7.3 Abstraction

In this section we first show that an abstraction mechanism for Indian parallel grammars that yields an over-approximation is possible. We then discuss why this is nevertheless not feasible in practice.

A general requirement to have an over-approximation is that derivation steps and pointer manipulations are commutable.

Lemma 4. *For an Indian parallel grammar G, an admissible heap configuration $H \in HG_{N \cup T}$ and a pointer manipulation $\mu \in \{x = P,\ x.s = P,\ new(x),\ x.d_i = v_i$, where P is either null, x or $x.s$ and $v_i \in Dom_i\}$ it holds that the set $\{H'[\mu] : H' \in \mathcal{L}(H)\}$ is the same as $\mathcal{L}(H[\mu])$, i.e. Lemma 1 holds also for Indian parallel grammars.*

Proof. Indian parallel grammars are context-free in the sense that the only application condition for a rule is whether the non-terminal on its left-hand side is present in the graph. In particular it does not depend on any concrete parts of the graph. Pointer manipulations do not change any abstract parts of the heap. Furthermore, we can use that Indian parallel grammars are a special case of context-free ones and that the property holds for them (see Lemma 1). Thus, the effect of the given pointer manipulations does not depend on whether further derivation steps have been executed before they are applied or after, as long as the heap is admissible when the pointer manipulation is executed. □

Next, we propose an abstraction mechanism and show that it fulfils the requirements for an over-approximation.

Definition 23 (Abstraction). Let $G = (N, T, P)$ be an Indian parallel grammar, $H, H' \in \mathrm{HG}_{N \cup T}$ hypergraphs and $p : X \to K \in P$ a production rule. Let $E^X = \{e_1, \ldots, e_n\} = \{e \in E_H \mid lab(e) = X\}$ be the set of edges in H labelled with X. H' is abstracted from H using p if and only if there is a set of non-overlapping embeddings $\{emb_1, .., emb_m\}$ such that $\forall e \in E^X \ \exists i \in \{1, \ldots, m\} : e \in \mathrm{img}(emb_i)$, i.e. each occurrence of X in H is covered by one of the embeddings, and $H' \cong replace(K, replace(\ldots replace(K, H, emb_1, X^\bullet)\ldots), emb_m, X^\bullet)$. We denote that H' is abstracted from H by $H \leftleftarrows H'$ and use $\leftleftarrows^*$ for the reflexive and transitive closure of $\leftleftarrows$. Furthermore, $abs(H) = \{H' \mid H \leftleftarrows^* H\}$ denotes the set of all hypergraphs that can be obtained from H by abstraction.

Intuitively this abstraction ensures that if abstraction introduces a non-terminal X, all other occurrences of this X have been created by the same abstraction. Then the abstraction can be reversed by applying the rule again in the forward direction on all occurrences of X. To ensure this, all occurrences of X have to be covered by the embeddings of the right-hand side of the rule.

Theorem 7. *The concretisation and abstraction given by Definitions 21 and 23 respectively yield an over-approximation.*

Proof. We show that the prerequisites for Theorem 3 are met and use their result.

1. $\mathcal{L}(H) = \bigcup_{H' \in con(H)} \mathcal{L}(H')$: This first equality can be shown by inserting the definitions:

$$\begin{aligned}\bigcup_{H' \in con(H)} \mathcal{L}(H') &= \{K \in \mathrm{HG}_T \mid \exists H' \in con(H) : H' \rightrightarrows^* K\} \\ &= \{K \in \mathrm{HG}_T \mid \exists H' \in \mathrm{HG}_{N \cup T} : H \rightrightarrows^* H' \rightrightarrows^* K\} \\ &= \mathcal{L}(H).\end{aligned}$$

2. $\mathcal{L}(H) \subseteq \mathcal{L}(H') \ \forall H' = abs(H)$: Let H' be obtained from H via one abstraction step using some rule $p : X \to rhs(p)$. Then all occurrences of X must originate from the backwards-applying of rule p. The derivation mechanism

for Indian parallel grammars thus allows us to apply rule p again in the forward direction to the obtained graph yielding H once again (of course the application of other rules than p might be possible yielding different graphs. This is not forbidden by the prerequisites of the theorem).
If H' is obtained from H via k abstraction steps, there are k intermediate hypergraphs H_1 to H_k. Each of is obtained from its predecessor by a single abstraction step, where all occurrences of the *lhs*-non-terminal of the applied rule originated from this step and thus the backwards application of this rule. Therefore, the abstractions can be undone in the reverse order they were done.

Since all requirements of Theorem 3 are met, the result follows immediately. □

7.4 Discussion

Though we have seen that there is at least one abstraction mechanism that yields an over-approximation, it is probably not useful for heap abstraction in practise.

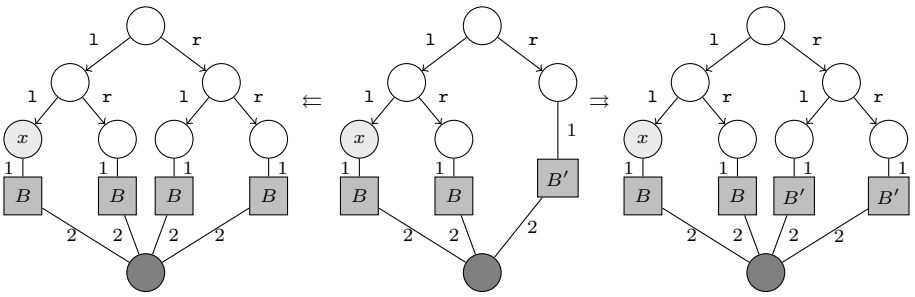


Fig. 13: Problematic Abstraction

The first problem is, that it either makes abstraction nearly impossible or over-approximates too much, i.e. desired properties like completeness of trees are likely lost after abstraction. We illustrate this on a short example. Consider the first graph in Figure 13 where one vertex is referenced by a variable x, indicated by an x written inside the vertex. This node must thus not be abstracted. We first try to abstract this graph with the grammar in Figure 11 Without additional rules added, the graph cannot be abstracted at all, because the B-edge below x has to stay and since all rules of the grammar have a B on their left-hand side, this means that they can either replace all B occurrences or not be used. This situation would apply almost always in the symbolic execution. In particular it would happen for all different depths of the tree therefore yielding an infinite state space.

As we have seen, the grammar without any additional rules to ease abstraction cannot be used for symbolic execution. In the following we observe that adding rules, though resolving this issue, introduces new severe problems:

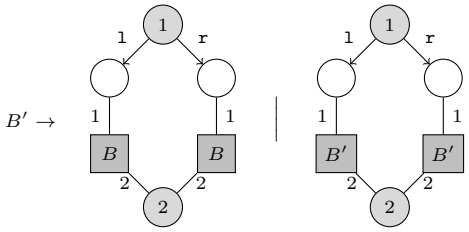


Fig. 14: Additional Rules to the grammar in Figure 11

Assume that we have the additional rules given in Figure 14. They alone are not enough to enable abstraction when holding on a vertex at arbitrary depth, but it is clear that something like the first rule is necessary to start the abstraction while keeping the fixed vertex and something like the second to continue this abstraction. Probably further rules that combine B and B' branches would also be useful, but we can show the principal problem with these two rules. The second graph in Figure 13 shows the result of applying the first rule backwards. Since B' did not occur previously in the graph, there are no restrictions to do so. The third graph is the result of applying the second rule in the forward direction. Though the graph itself is again complete we could now apply rules to the B and the B' edges independently, yielding a severely unbalanced tree.

The example illustrates a problem that arises in many if not all Indian parallel grammars. Either we cannot abstract X-labelled non-terminal edges when holding on to an adjacent vertex to one of these edges. Or we break the principle of parallel derivations by splitting the set of X-labelled edges so that they can be replaced independently. In the first case the symbolic execution yields an infinite state space, in the second it probably results in a false negative. Therefore, Indian parallel grammars, though yielding an important graph set, are unlikely to be useful for heap abstraction as presented by Heinen et al. [11].

8 Indexed Grammars

Indexed Grammars are an especially useful extension of HRGs. They preserve most of the positive properties and are capable of expressing useful data structures, e.g. balanced trees. For an introductory example see Figure 15, which we now discuss shortly: When ignoring the indices (the content in square brackets at the non-terminal edges) the grammar generates the set of all binary trees.

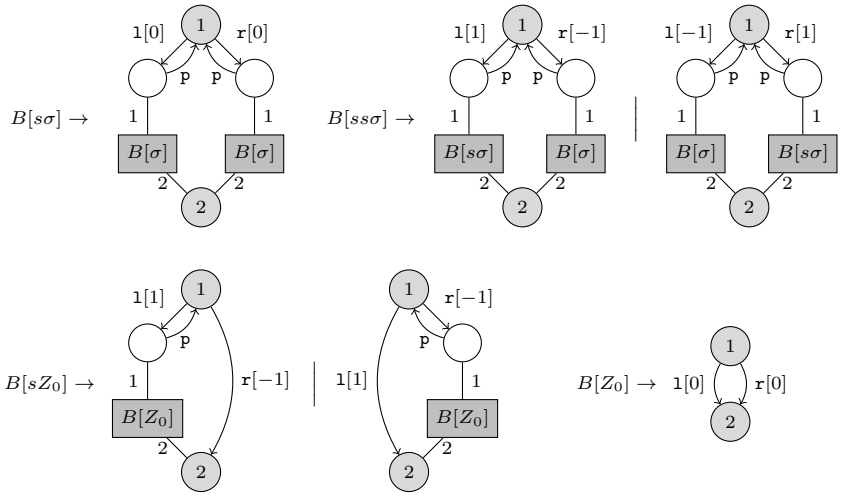


Fig. 15: Grammar for balanced trees

Let us have a look at the indices. In this grammar the indices can be interpreted as natural numbers encoding the height of the tree generated by the non-terminal edge. All rules create at least one sub-tree whose hight is exactly one less than the parent vertex, thus ensuring the correct height of the parent vertex. Additionally, the second and third rule allow one of the sub-trees to be one smaller in height than its sibling. Altogether the language of this grammar is the set of all balanced trees (the height of siblings differs by at most one). Further, the (terminal) left and right edges are annotated by an information about their relative height in comparison to their sibling. This is necessary to provide a path grammar for abstractions that begin at the root (see Figure 18).

In the following, we first introduce the new mechanisms formally and then analyse the properties of this grammar type focusing mainly on properties necessary for heap abstraction.

8.1 Definition

Since this new grammar type operates on indexed (and annotated) hypergraphs, let us first provide a definition:

Definition 24 (Indexed (Annotated) Hypergraph). A graph H is an *indexed (annotated) hypergraph* over the alphabets T and N if it is a non-terminal hypergraph where each non-terminal edge is additionally labelled by a stack from a finite alphabet F. Formally, an index function is added to the graphs: $ind : E^N \to F^*$. Additionally, the concrete elements can be annotated with elements from a set A. This is expressed by a partial function $ann : V \cup E^T \dashrightarrow A$ mapping concrete elements of the graph optionally to elements of A. Overall a

graph is altogether a tuple $H = (V, E, att, lab, ext, ind, (ann))$, where V, E, att, lab and ext are defined as usual.

We denote the *class of all indexed hypergraphs* by $\mathrm{iHG}_{N \cup T}$.

We denote indexed hypergraphs in the same manner as other hypergraphs, with the values of the functions *ind* and *ann* being written in square brackets to their respective elements. Note that a confusion between those two functions is not possible since their domains are disjoint. The annotation function is partial, i.e. not necessarily all concrete elements are annotated. If the function is completely empty, it may also be omitted.

Remark 3. One the one hand HRGs tread annotations like an extended or second label, i.e. the annotation function after applying a rule is defined exactly as the labelling function as the union of the old and the inserted graph and it is also preserved by embeddings. On the other hand, symbolic execution ignores them, i.e. the program has no access to the annotation and cannot change it, except that a deleted element is removed from its domain.

Indexed string grammars have been introduced by Aho [1], but Ullman [13] has provided a slightly different yet equivalent formulation which can be more easily transferred to HRGs. We first define indexed HRGs strictly according to their rules and then show that more flexible rules can be added without changing expressiveness. (Note that the example in Figure 15 makes use of these additional rules). For this section we use the following notation:

$X[\sigma]$ is a combination of the non-terminal symbol X and the stack σ, i.e. it refers to an edge e where $lab(e) = X$ and $ind(e) = \sigma$;
$K[\sigma]$ represents a hypergraph where σ is assigned to all non-terminal edges by the index function. Formally, $\forall e \in E_K^N : ind_K(e) = \sigma$.

Definition 25 ((Strict) Indexed Grammar). An *indexed grammar* is a tuple $G = (N, T, P, F)$ where F is a stack alphabet. Rules in P are of one of three forms:

i) $X[\sigma] \to K[\sigma]$
ii) $X[\sigma] \to K[f\sigma], f \in F$
iii) $X[f\sigma] \to K[\sigma], f \in F$

The first rule type is most similar to 'usual' hyperedge replacement rules. An edge e labelled by the non-terminal X is replaced by the hypergraph K. The index function assigns the stack originally assigned to e to all non-terminal edges in K. The second rule additionally adds one element to the stacks and the third rule pops one stack symbol. Note that rules of this last type are only applicable if the top stack symbol is the one specified by the rule (Thus, this type can be used for case distinctions depending on the top stack symbol).

In principle, edge replacement with indexed HRGs is identical to edge replacement with simple HRGs. Thus, we give only the index and annotation function for all three types.

Definition 26 (Hyperedge Replacement with Index). Let $H \in \mathrm{iHG}_{N \cup T}$ be the original graph and $H' \in \mathrm{iHG}_{N \cup T}$ the graph after applying the rule. Let e be an edge in H that is labelled with X and has the stack σ. The application of a rule yields:

i) $H' \cong H[\ K[\sigma]/X[\sigma]\ ]$: $ind_{H'} = (ind_H \upharpoonright (E_H^N \backslash \{e\})) \cup \{e' \mapsto ind_H(e) | e' \in E_K^N\}$

ii) $H' \cong H[\ K[f\sigma]/X[\sigma]\ ]$: $ind_{H'} = (ind_H \upharpoonright (E_H^N \backslash \{e\})) \cup \{e' \mapsto f \cdot ind_H(e) | e' \in E_K^N\}$

iii) $H' \cong H[\ K[\sigma]/X[f\sigma]\ ]$: $ind_{H'} = (ind_H \upharpoonright (E_H^N \backslash \{e\})) \cup \{e' \mapsto stack \mid e' \in E_K^N\}$, so that $f \cdot stack = ind_H(e)$

Furthermore, the annotation function is $ann_{H'} = ann_H \cup ann_K$ in all three cases.

8.2 Emptiness

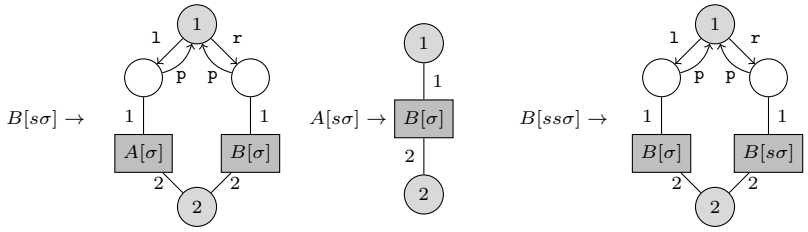
For indexed string grammars it is decidable if the corresponding language is empty. This result can be extended for indexed HRGs using Theorem 1.

Theorem 8. *For an indexed grammar $G = (N, T, P, F)$ and an indexed hypergraph $H \in iHG_{N \cup T}$ it is decidable whether the language is empty, i.e. if $\mathcal{L}(H) = \emptyset$.*

Proof. Let $G = (N, T, P, F)$ be an indexed HRG. Let us assume for a moment, that the stacks do not matter, i.e. that rules of the third type are always applicable. The resulting HRG is thus equivalent to a normal HRG without any stacks. For those, we know by Theorem 1 that a corresponding string grammar can be constructed, having isomorphic derivation trees. We can now reintroduce the stacks, i.e. add them to both the HRG and the corresponding CFG (yielding an indexed string grammar). We will notice that some of the derivation trees are no longer admissible, with respect to both the indexed HRG and the indexed string grammar, as in some cases rules of type three are not applicable due to the indices. Since the languages corresponding to the grammars are empty if and only if the set of derivation trees is empty and as emptiness for the string grammar is decidable [1], it is also decidable for indexed HRGs. □

8.3 Generalisation

While the rule types introduced so far provide sufficient expressiveness and can be conveniently handled in proofs, they have various disadvantages: See for example Figure 16a. Those rules are part of an HRG generating balanced trees and allow the left branch to be less deep by one than the right branch. Figure 16b aims at the same productions, but does not need the detour with non-terminal A. On the other hand it does need several mechanisms that are not provided by indexed HRGs as introduced in Definition 25. Firstly, it needs to see more than one stack symbol at a time and secondly, the non-terminal edges occurring



(a) The strict version of the grammar

(b) The easier version of the grammar

Fig. 16: Illustration of strict and general indexed Grammars

in the right-hand side of the rule are not all assigned the same stack. This example might highlight two disadvantages of HRGs that are only build according to the strict rules given in Definition 25. The grammar is not easy to read, as the production of a new vertex where the depth of the subtrees differ by one is distributed on two separate rules. Also it poses a problem for heap abstraction as it does not ensure increasingness, i.e. when applying the second rule to A, no new vertex is created.

We now formalize what degree of flexibility we allow and show that it does not change expressiveness. We will further see that all grammars using the new types can be transformed easily to grammars using only the three strict types. We allow the following rules:

- The graph on the right-hand side can be any indexed graph, where the stacks may end with a variable σ referring to the stack of the replaced edge.
- Rules may check for a finite sequence of stack symbols, i.e. we allow rules $p : X[f_1 \dots f_n\sigma] \to K \in \mathrm{iHG}_{N\cup T}$.

This yields the following extension of indexed HRGs:

Definition 27 (Indexed Hyperedge Replacement Grammar). An *indexed HRG* is a tuple $G = (N, T, P, F)$, where F is a finite alphabet of stack symbols and rules in P have the form $p : X[f_1 \dots f_n\sigma] \to K^\sigma$, where $f_1 \dots f_n$ is a possibly empty sequence over the alphabet F, σ is a variable and K^σ is an indexed (annotated) hypergraph in which stacks can optionally end with the variable σ.

Note that all rules according to Definition 25 are also covered by this new definition. We also need a new formalism for edge replacement.

Definition 28 (Indexed Hyperedge Replacement). Let $K^{[f_1\cdots f_n/\sigma]}$ denote the graph that results from replacing all occurrences of σ in K^σ by $f_1 \dots f_n$. Let $H, H', K' \in \mathrm{iHG}_{N\cup T}$ be indexed hypergraphs, $e \in E_H$ an edge of H and $K' \cong K^{[f_1\cdots f_n/\sigma]}$.
Then $H' \cong H[K^{[f_1\cdots f_n/\sigma]}/e]$ if and only if $V_{H'}$,$E_{H'}$,$att_{H'}$, $lab_{H'}$ and $ext_{H'}$ are as in Definition 2, $ind_{H'} = (ind_H \upharpoonright (E_H^N \backslash \{e\})) \cup ind_{K'}$ and $ann_{H'} = ann_H \cup ann_{K'}$.

Theorem 9. *For each indexed HRG G in the general form (Definition 27) an equivalent HRG G' in the strict form (Definition 25) can be constructed such that $\forall H \in iHG_{N \cup T} : \mathcal{L}(G, H) = \mathcal{L}(G', H)$.*

Proof. We proof this theorem by giving a construction of G' in the strict form for any given indexed HRG G and then show the correctness of the construction.

Construction: Without loss of generality assume that all non-terminal labels created during this process are distinct (this can always be achieved by renaming).

For rules where the right-hand side contains a non-terminal edge $X[f_n \dots f_1 \sigma]$: Replace this edge by an edge of the same rank labelled with new non-terminal X_1 and stack σ. Create additional new non-terminals $X_2 \dots X_n$ and add rules $X_1[\sigma] \to X_2^\bullet[f_1\sigma]$, …, $X_{n-1}[\sigma] \to X_n^\bullet[f_{n-1}\sigma]$, $X_n[\sigma] \to X^\bullet[f_n\sigma]$ which push the desired sequence to the stack. Repeat this for all such occurrences.

For rules where the right-hand side contains a non-terminal edge $X[f_n \dots f_1]$: Replace this edge by an edge of the same rank labelled with new non-terminal Y and stack σ. We now add rules so that the original stack σ is removed: $Y[f\sigma] \to Y^\bullet[\sigma]$ for all $f \in F$. Then we create new non-terminals $X_1, \dots, X_{n-1}$ and rules $Y[] \to X_1^\bullet[f_1]$, $X_1[\sigma] \to X_2^\bullet[f_2\sigma]$, …, $X_{n-1}[\sigma] \to X[f_n\sigma]$. These rules push the sequence $f_n \dots f_1$ on the stack.

The two transformations above ensure that non-terminal edges on the right-hand side of all rules are labelled by σ or $f\sigma$ for an $f \in F$ which satisfies the requirements of indexed HRGs.

We now give a transformation for rules that additionally check for more than one stack symbol, i.e. rules of the form $X[f_1 \dots f_n\sigma] \to K^\sigma$: First introduce new non-terminals $X_1, \dots, X_n$ and new rules checking for the sequence $f_1 \dots f_n$: $X[\sigma] \to X_1^\bullet[\sigma]$,$X_1[f_1\sigma] \to X_2^\bullet[\sigma]$, …, $X_n[f_n\sigma] \to K^\sigma$. The last rule is then further transformed using the transformation given for this rule type above.

Correctness We show the inclusion in both directions:
$\mathcal{L}(G, H) \subseteq \mathcal{L}(G', H)$: Let H' be a hypergraph in $\mathcal{L}(G, H)$. We show that it is also in $\mathcal{L}(G', H)$. Since $H' \in \mathcal{L}(G, H)$ there is a derivation sequence $p_1 \dots p_n \in P^*$ such that $H \Rightarrow_{p_1} \dots \Rightarrow_{p_n} H'$. Induction on the length of the derivation sequence:

IB: The case $H = H'$ is trivial, therefore we consider derivations that consist of only one step $\Rightarrow_{p_1}$. The rule p_1 has the form $X[f_1 \dots f_n\sigma] \to K^\sigma$. Then there are rules $X[\sigma] \to X_1^\bullet[\sigma]$, …,$X_{n-1}[f_n\sigma] \to X_n^\bullet[\sigma]$. Therefore, $H \Rightarrow^* H[X_n^{[\sigma/f_1 \dots f_n\sigma]}/X]$, the graph where the X-edge is relabelled by X' and the stack symbols $f_1 \dots f_n$ are removed, is a valid derivation in G'. The constructed grammar must further contain the rule $X_n[\sigma] \to K'[\sigma]$. Applying this rule yields a graph that is with exception of the indices and renaming of non-terminal edges, isomorphic to H'. By construction, G' consists of additional rules for each non-terminal edge in K' that set the correct index and, in the last step, also set the label of the non-terminal edges correctly. Therefore, we have for each derivation $H \Rightarrow_p H'$ a sequence $p'_1 \dots p'_m$ in P' such that $H \Rightarrow_{p'_1} \dots \Rightarrow_{p'_m} H'$.

IH: For $H \Rightarrow^n H'$ in G, i.e. H' is derived from H in less than n steps with rules from P, we have that $H \Rightarrow^* H'$ in G'.

IS: Let $H' \in \mathcal{L}(G, X)$ be derived from H in $n+1$ steps. Then there is an intermediate graph H'' such that $H \Rightarrow^n H'' \Rightarrow H'$ is a valid derivation in G. By the induction hypothesis there are valid derivations $H \Rightarrow^* H''$ and $H'' \Rightarrow^* H'$ in G' possible and thus also a derivation $H \Rightarrow^* H'$ is possible in G' and since H' is terminal (as it is in $\mathcal{L}(G, H)$) we have $H' \in \mathcal{L}(G', H)$.

Since we have shown for all $H' \in \mathcal{L}(G, H)$ derived from H in arbitrarily many steps that H' is also in $\mathcal{L}(G', H)$ we have shown that $\mathcal{L}(G, H) \subseteq \mathcal{L}(G', H)$.

$\mathcal{L}(G', H) \subseteq \mathcal{L}(G, H)$*:* Let $H' \in \mathcal{L}(G', H)$, i.e. H' is terminal and there is a derivation sequence $p_1 \dots p_m \in P'^*$ such that $H \Rightarrow_{p_1} \dots \Rightarrow_{p_m} H'$. Note that most rules in P' consist of new non-terminals that are then used only in the right-hand side of one rule and the left-hand side of another rule, i.e. they yield deterministic derivation sub-sequences (Though several such sequences can be intertwined). For any derivation $H \Rightarrow^* H'$ there is also a derivation where those sequences are executed one after another, i.e. where after applying a rule $X \to X_1^\bullet$, $X_1 \in N' \backslash N$, the next rule is the (by construction unique) rule p with $rhs(p) = X_1$. It is further possible that after applying a rule $p : X \to K$ with $lab(E_K^N) \subset N' \backslash N$ first all non-terminal edges introduced by this rule are replaced before any concretisation on other parts of the graph is done. All this reordering is possible since all rules are context-free.

After this reordering we can split the derivation sequence so that the intermediate graphs have only non-terminal edges labelled in N, i.e. $H \Rightarrow^* H'$ becomes $H \Rightarrow^* H^1 \Rightarrow^* \dots H^n \Rightarrow^* H'$ with $lab(E_{H^i}^N) \subset N$ for all $i \in \{1, \dots, n\}$. We now analyse one derivation segment $H^i \Rightarrow^* H^{i+1}$, where $lab(E_{H^i}^N), lab(E_{H^{i+1}}^N) \subset N$, and this property does not hold for further intermediate graphs. According to the ordering we applied, the derivation sequence consists of two phases:

1. First an edge e in H^i is renamed (probably several times). By construction, this process removes a sequence $f_1 \dots f_n$ from the respective stack. (To be precise e itself is not renamed but replaced by new edges - but the result is the same). This phase ends when a rule $X \to K[\sigma]$ is applied. If $lab(E_K^N) \cap N' \backslash N \neq \emptyset$, the second phase is triggered:
2. After a rule $X \to K[\sigma]$ was applied, all edges with label in $lab(E_K^N) \cap N'$ are renamed (possibly several times). By construction, stack symbols can be either pushed or the whole stack is removed and replaced by a new one. This phase ends after the graph has no non-terminals in $N' \backslash N$. By construction, after this phase the edge e is replaced by a graph K^σ.

All rules used in these phases have been added by the construction and for all intermediate non-terminal there is exactly one rule where it appears on the left and one where it appears on the right-hand side. By construction, those rules where only added if there is a rule $X[f_1 \dots f_n \sigma] \to K^\sigma$ in the original grammar G. Therefore, we can apply this rule and have $H^i \Rightarrow H^{i+1}$.

Finally, we consider the case $H^i \Rightarrow H^{i+1}$, i.e. the derivation consists of one step and no non-terminals from $N' \backslash N$ are introduced. Such a rule is never created during the transformation and thus has to originate from the original grammar G (G might contain rules that are already of one of the three types). This derivation is therefore also possible in G.

Since the above arguments apply to all segments of the derivation, we have shown that for all $H' \in \mathcal{L}(G', H)$ it holds also $H' \in \mathcal{L}(G, H)$ and therefore $\mathcal{L}(G', H) \subseteq \mathcal{L}(G, H)$.

Altogether we have thus shown that $\mathcal{L}(G, H) = \mathcal{L}(G', H)$ for each H in $\mathrm{iHG}_{N \cup T}$. □

Note that derivations in such a transformed grammar can be unproductive although the original grammar was not if the sequence is different from $f_1 \dots f_n$, i.e. not all derivations lead to a terminal graph and that the resulting grammar is probably not increasing, i.e. there are rules where the right-hand side is neither terminal nor strictly larger than the handle corresponding to the left-hand side. Both are rather bad properties in the context of heap abstraction.

Emptiness is still decidable since every grammar in this form can be transformed to the classic form before projecting to string grammars.

8.4 Abstraction

As we have already seen for automata controlled and Indian parallel grammars, a main challenge for an extension of HRGs is to provide a suitable abstraction mechanism. For indexed grammars there are two different mechanisms: One as usual for the graphs and a second one for the indices (see Definition 33).

The abstraction mechanism on graphs is similar to the one for HRGs. It embeds the right-hand side of a rule in the graph and replaces it with a handle corresponding to the left-hand side, merging the external vertices. For indexed grammars additionally the index has to agree and if necessary the variable σ is instantiated appropriately. We thus lift the definitions of embedding and the replace mapping to indexed hypergraphs:

Definition 29 (Embedding). Given K, $H \in \mathrm{iHG}_{N \cup T}$, an *embedding* emb of K in H is a pair of mappings $emb_V : V_K \to V_H$ and $emb_E : E_K \to E_H$ where all properties of Definition 6 hold and additionally

$$\forall e \in E_K^N : ind_K(e) = ind_H(emb_E(e)).$$

If the graph is annotated we further require that

$$\forall e \in dom(ann_K) \cap E_K : ann_K(e) = ann_H(emb_E(e)) \text{ and}$$
$$\forall v \in dom(ann_K) \cap V_K : ann_K(v) = ann_H(emb_V(v)).$$

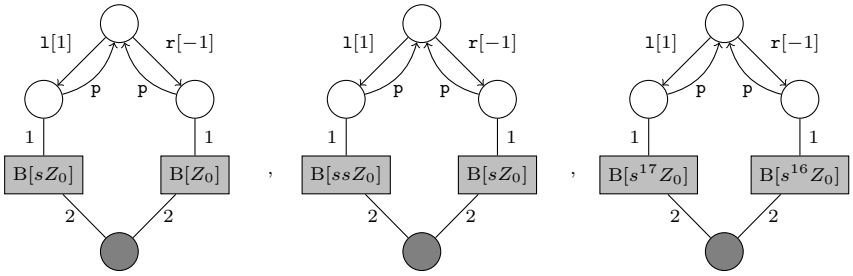
Definition 30 (Replace mapping). $H' \cong replace(K^\sigma, H, emb, X^\bullet[f_1 \dots f_n\sigma])$ if and only if there is a sequence $g_1 \dots g_m \in F^* \cdot \{N^i \cup \{\epsilon\}\}$ such that emb is

an embedding from $K^{[g_1 \dots g_m/\sigma]}$ to H, all properties from Definition 7 hold and additionally

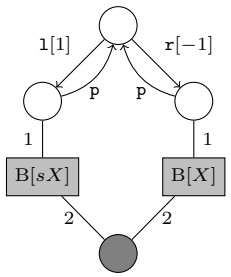
$$ind_{H'} = ind_H \restriction E_{H'} \cup \{e \mapsto f_1 \dots f_n g_1 \dots g_m\}$$

If the graph is annotated it must also hold that $ann_{H'} = ann_H \restriction (V_{H'} \cup E_{H'})$.

Abstraction of Indices The goal of using HRGs in the context of heap abstraction is to obtain a finite state space that can then be analysed using standard techniques. The abstraction mechanism on graphs alone does not fulfil this goal. Although the number of graphs may become finite when neglecting the index, the state space can still be infinite via the set of possible indices. For an example consider the grammar for balanced trees (Figure 15): The grammar enables us to abstract most parts of trees, but we still have the index to memorize the depth of the tree, i.e. for each different depth we obtain different indices and thus an infinite state space (see Figure 17a). On the other hand we see, that the essential information to guarantee balanced trees is encoded by the difference between indices not by their absolute values (Figure 17b highlights this by replacing parts of the index by an abstract symbol X).



(a) Illustration of Infinite State Space



(b) Example of Abstract Stack

Fig. 17: Motivation for Stack Abstraction

Furthermore, any rule application depends only on a finite prefix of the index. It is thus useful to provide a mechanism to abstract the postfix of an index. An abstract postfix represents a set of concrete postfixes, i.e. it is a language over the alphabet F. In the most general case a single abstract postfix representing F^* is sufficient, but it can be any language as long as the original postfix is contained. This mechanism works also if the postfix already contains an abstract part. Then the language of the new abstract postfix must contain the concatenation of the concrete part and the old language.

Concretisation of Index Allowing abstraction of the index, we should also provide an operation to reverse this, i.e. to concretise an abstract index. The result must be a subset of the given abstract index.

Both concretisation and abstraction of the index can be realized using a right regular grammar (see Appendix A). Derivations in these grammars always consist of a terminal prefix and a single non-terminal at the right. Each derivation step enlarges the prefix by one. To enable index abstraction the user therefore provides a right regular grammar. Then the rightmost part of all stacks can be replaced by the left-hand side of a rule, if it is equal to the right-hand side. The stacks can then be concretised by applying one of the production rules in the grammar to all stacks. (Note that this mechanism always has to be applied to all occurring stacks in parallel. Otherwise too much information is lost). With $G^i = (\{X\}, F, \{(X \to fX \mid f) : f \in F\})$ we give a default grammar that produces the language F^*.

Example. To illustrate how a more specific grammar can be used to preserve more information, we give an index abstraction grammar for the example of balanced trees (Figure 15): $G^i = (\{X\}, \{s, Z_0\}, P)$

$$\begin{aligned} P: \quad & X \to sX \\ & X \to Z_0 \end{aligned}$$

This grammar preserves the bottom symbol Z_0 and thus ensures that stacks can always be interpreted as valid natural numbers.

In comparison to the over variants, indexed grammars have two different types of concretisation and abstraction. We now define all these cases formally:

Definition 31 (Concretisation / Derivation). Let $G = (N, T, P, F)$ be an indexed HRG, $G^i = (N^i, F, P^i)$ a right regular grammar and $H, H' \in \mathrm{iHG}_{N \cup T}$.

a) Rule Application ($H \Rightarrow_p H'$):
   H' is derived from H by the application of $p : X \to K^\sigma \in P$ if and only if e is a non-terminal edge in E_H labelled with X and H' is isomorphic to $H[K^\sigma/e]$.
b) Index Concretisation ($H \Rightarrow_{p^i} H'$):
   H' is derived from H by application of the index abstraction rule $q : X^i \to$

$\alpha \in P^i$ ($\alpha \in \{fY^i, f\}$) if and only if h is an isomorphism from H to H', for all edges in E_H^N holds $ind(e) \in F^*X$ and $ind(h(e)) = ind(e)[\alpha/X]$, i.e. all occurrences of X in H are replaced by α in H'.

We use $\Rightarrow$ to denote $(\Rightarrow_P \cup \Rightarrow_{P^i})$. $\Rightarrow^*$ is accordingly the reflexive and transitive closure of $\Rightarrow$. Further, we define for all non-terminal indexed hypergraphs H the set $con(H) = \{H' \in \mathrm{iHG}_{N\cup T} \mid H \Rightarrow^* H'\}$ as the set of all derivable hypergraphs.

We define the language of an indexed HRG starting from an indexed hypergraph H:

Definition 32 (Language). The language generated from $H \in \mathrm{iHG}_{N\cup T}$ with respect to the indexed HRG $G = (N, T, P, F)$ is defined as the set of all concrete hypergraphs derivable from H in G, i.e. $\mathcal{L}(H) = \{K \in \mathrm{HG}_T \mid H \Rightarrow^* K\}$.

Abstraction is the reverse operation for both cases:

Definition 33 (Abstraction). Let $G = (N, T, P, F)$ be an indexed HRG, $G^i = (N^i, F, P^i)$ a right regular grammar and $H, H' \in \mathrm{iHG}_{N\cup T}$.

a) Graph abstraction ($H \Leftarrow_p H$):
   H' is the result of the backwards application of a rule $p : X[f_1..f_n\sigma] \to K^\sigma$ to H if and only if there is an embedding emb from K^σ to H and H' is isomorphic to $replace(K^\sigma, H, emb, X^\bullet[f_1 \dots f_n\sigma])$.
b) Index Abstraction ($H \Leftarrow_{p^i} H'$):
   H' is the result of an index abstraction on H if and only if there is a rule $p^i : X \to \alpha \in P^i$ ($\alpha = fX'$ or f) and there is an isomorphism h from H to H' so that for all e in E_H^N holds $lab(e) \in F^*\alpha$ and $lab(h(e)) = lab(e)[\alpha/X]$, i.e. at the end of all stacks α is replaced by the non-terminal X.

We use $\Leftarrow$ to denote $\Leftarrow_P \cup \Leftarrow_{P_i}$. $\Leftarrow^*$ is accordingly the reflexive and transitive closure of $\Leftarrow$. Further, we define for all indexed hypergraphs H the set $abs(H) = \{H' \in \mathrm{iHG}_{N\cup T} \mid H \Leftarrow^+ H'\}$ as the set of hypergraphs that can be abstracted from H in at least one step.

Remark 4. It might be necessary to further restrict index abstraction in order to not lose too much information. For example in the grammar for balanced trees (Figure 15) this abstraction must only be used if no vertex has a pointer to the *null* vertex left. Otherwise unbalanced parts might be introduced.

As indexed HRG derivations consist of two different mechanisms, which makes derivation sequences more complicated, we establish a normal form where first the index is concretised and then the hyperegde replacement is applied. The formal statement and proof can be found in Appendix B.

We are now in a position to show the correctness of our proposed abstraction mechanism: As we have done in the previous sections, we first show that pointer manipulations and derivation steps are commutable:

Lemma 5. *For an indexed HRG G, an admissible heap configuration $H \in HG_{N\cup T}$ and a pointer manipulation $\mu \in \{x = P,\ x.s = P,\ new(x),\ x.d_i = v_i$, where P is either null, x or $x.s$ and $v_i \in Dom_i\}$ it holds that the set $\{H'[\mu] : H' \in \mathcal{L}(H)\}$ is the same as $\mathcal{L}(H[\mu])$, i.e. Lemma 1 holds also for indexed HRGs.*

Proof. This property holds for context-free HRGs, so we only argue why this still holds for indexed HRGs. On the one hand the semantics for pointer manipulations is only concerned with the concrete part of the heap. We assume that the semantics is extended in a way, that it ignores additional annotations (such as the balancing information in the grammar for balanced trees). Then pointer manipulations are not influenced by the fact, that the underlying grammar is indexed and thus this direction of the lemma still holds. On the other hand concretisation (both concerning the graph and the index) depends only on the non-terminals and their stacks which are both not modified by any pointer operations. We can thus extend the result to indexed HRGs. □

Theorem 10. *Indexed hyperedge replacement grammars with the proposed concretisation and abstraction mechanisms yield an over-approximation.*

Proof. We show the prerequisites of Theorem 3:
First we show

$$\mathcal{L}(H) = \bigcup_{H' \in con(H)} \mathcal{L}(H') \tag{1}$$

We start with the left-hand side of Equation (1) and replace $con(H)$ and $\mathcal{L}(H')$ by their definitions:

$$\bigcup_{H' \in con(H)} \mathcal{L}(H') = \bigcup_{\{H' \in \mathrm{iHG}_{N\cup T} \mid H \Rightarrow^* H'\}} \{K \in \mathrm{iHG}_T \mid H' \Rightarrow^* K\}$$

Next, we explicitly join the set by formulating the set where there exists an intermediate hypergraph H':

$$= \{K \in \mathrm{iHG}_T \mid \exists H' \in \mathrm{iHG}_{N\cup T} : H \Rightarrow^* H' \Rightarrow^* K\}$$

Since both derivation sequences can potentially be empty, we can also omit this intermediate graph, yielding the definition of the language of H which is the left-hand side of the equation:

$$\{K \in \mathrm{iHG}_T \mid H \Rightarrow^* K\} = \mathcal{L}(H)$$

The second prerequisite is that for all $H' \in abs(H)$ we have $\mathcal{L}(H) \subset \mathcal{L}(H')$. Like in Section 6, we show instead that $H \Leftarrow H'$ implies $H' \Rightarrow H$, i.e. that if H' can be abstracted from H, then H can be derived from H', which is a stronger result. This follows directly from the definition:

a) Let H' be a graph abstraction of H by rule $p : X[f_1 \dots f_n\sigma] \to K^\sigma$. Then there is an embedding *emb* from K^σ to H and the resulting graph H' is isomorphic to $replace(K^\sigma, H, emb, X^\bullet[f_1 \dots f_n\sigma])$. By the definition of *replace*, H' consists of an edge e with $lab(e) = X$ and $ind(e) = f_1 \dots f_n\sigma$. Thus, p can be applied to H' and by the definitions of embeddings, replace and hyperedge replacement the resulting graph is isomorphic to H.

b) Assume now that H' is obtained from H by index abstraction. Since index concretisation is defined as the exact reverse operation to index abstraction, the claim follows immediately.

By the theorem it follows that indexed HRGs with the proposed concretisation and abstraction mechanisms are an over approximation. □

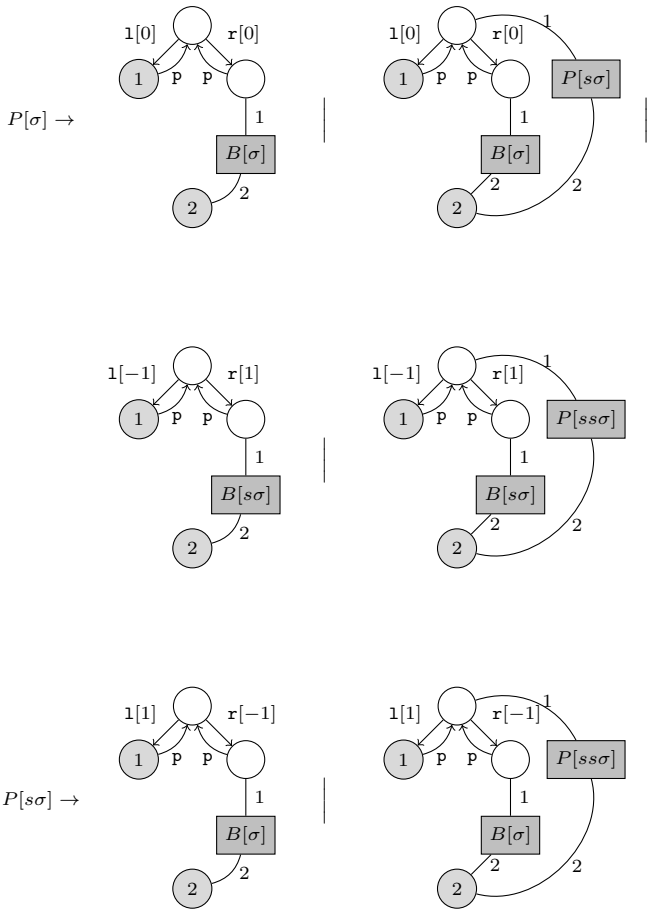


Fig. 18: Path Grammar for Balanced Trees (symmetric cases omitted)

A Path Grammar for Balanced Trees Let us return to the example of a grammar for balanced trees (see Figure 15). As explained in the introduction this grammar can produce the set of balanced binary trees. Until now we can concretise and abstract arbitrary branches, but when we fix some vertex, the path

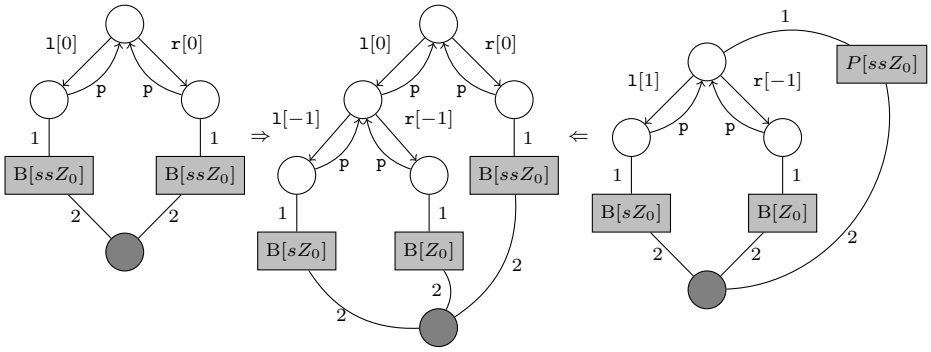


Fig. 19: An example derivation and abstraction of balanced trees (see Figures 15 and 18)

from the root to this vertex must remain concrete. Unfortunately, such a path can have arbitrary length which hinders us from obtaining a finite state space. To enable abstraction beginning at the root, we have to provide a path grammar, i.e. a grammar that enables us to abstract the root vertex together with one of its sub-trees, yielding the remaining sub-tree with a non-terminal edge attached to its root that represents this abstracted part. When re-concretising this non-terminal, the tree must be once again balanced, therefore it must remember (via its index) the depth of the concrete sibling. As our approach is context-free, it has no access to this sibling itself, but it can reconstruct it from its own height if it knows what its relative high is with respect to the sibling. This information can be generated during derivations and stored at the left/right pointers via an annotation as we have already done in the example. A path grammar for balanced trees that uses this mechanism is provided in Figure 18. Figure 19 shows first an example derivation step using the grammar in Figure 15 and then an abstraction step using the path grammar in Figure 18.

8.5 Maintaining Annotations

Annotations, like the balancing information as used in our example of balanced trees, can become outdated when pointer manipulations are applied. This does not effect correctness, since they are treated like ordinary edge labels within the grammar, and since we have shown the correctness of the approach. But it can hinder abstraction, so that a finite state space is not longer obtained. For an example see Figure 20. Assuming we want to collapse the displayed sub-graph we see by comparing with the rules in Figure 15 that this is not possible in Figure 20a, whereas the same graph with corrected annotations can be abstracted as is shown in Figure 20b.

This obstacle is overcome if we allow to change the annotations of the concrete part of the heap during the symbolic execution. Note that the verification remains correct if 'incorrect' annotations, i.e. ones that are not consistent

with the intended semantics of the annotation, are applied. Figure 21 illustrates this shortly. Although the first graph in Figure 21a is incorrectly annotated (both sides have the same depth while the annotation indicates that the left one should be deeper), we apply an abstraction. Re-concretising the P-edge then yields graphs that are also not correctly annotated. In particular Figure 21d is identical to the original graph. Therefore, no information is lost.

Thus, it would in principle be possible to try in each step all possible combinations of annotations and choose the one that enables abstraction. But intuitively it is better to compute updated annotations with respect to their intended semantics. An algorithm to do this operates rather on the hypergraph layer than on the heap layer. Thus, it is sensible that it has also access to the abstract components of the heap.

To illustrate this, we give an example algorithm that computes the annotations for a concrete segment of a balanced tree (Algorithm 1). For this algorithm we assume that the heap is still a tree. This assumption does not necessarily hold at all times during the symbolic execution. But since it is relatively easy to check whether a given graph forms a tree (by checking for cycles), we assume that the algorithm is only executed at times where this holds. Therefore, we can define that the function *sibling*(v) returns the unique sibling of a vertex v and that *edge_to*(v) returns its unique incoming edge. Note that the *edge_to* function would not be possible in a pointer manipulation program, as it returns an edge that merely represents a reference, but this algorithm operates on another level. Accordingly, we have access to all components of the graph, e.g. $att(e, 1)$ returns the first element of the sequence of vertices attached to e.

The proposed algorithm first sets all annotations to *undefined*. Then it iteratively computes the height of vertices, compares it to the height of its sibling and sets the annotations accordingly. If this difference is more than one, the annotation becomes undefined (Abstraction at this point is not possible until the balancing is fixed). The iteration is started by assigning the height -1 to the *null* vertex and then evaluating the height of those vertices that are attached to a

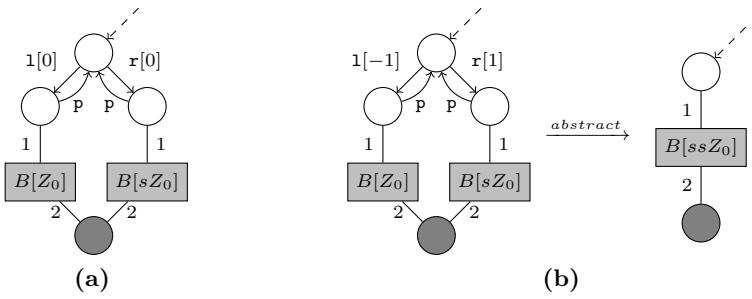


Fig. 20: Abstraction hindered

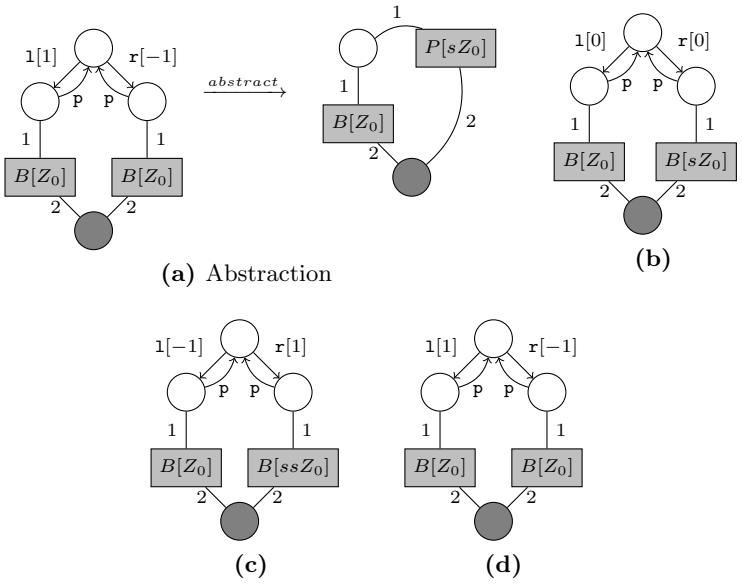


Fig. 21: Abstraction with wrong annotations

non-terminal B-edge (This is done by counting the s-symbols on the respective stack). After this initialisation the algorithm looks for pairs of siblings where both are assigned a height. It evaluates the balancing, computes the height of the parent and then sets the height of both vertices to *undefined* as they are not longer needed. The algorithm terminates if it cannot find any more such pairs. Since the height is only propagated upwards and since the graph is a finite tree, termination is ensured.

8.6 Example: Verification of AVL Insertion

We now give a detailed example that uses the concepts introduced in this section and illustrates the practical relevance of the approach. The example explains how a symbolic execution of an AVL insertion and rebalancing algorithm would work.

AVL Tree An *AVL tree* is a balanced search tree, where vertices can only be inserted at the leaves. After each insertion step the algorithm checks whether the tree is still balanced and rebalances it if necessary. This rebalancing is done by rotations that also preserve the search tree property [3].

In order to track balancing, an AVL tree stores for each vertex whether the height of its left child is greater, equal or less than the one of its right child. This information is then used to detect after an insertion or deletion where the lowest vertex that violates the balancing, is.

Algorithm 1 Computation of Balancing Annotations

Require: $G = (V, E, att, lab, ext, ind, ann)$ is a tree over the signature $T \cup N = \{left, right, parent\} \cup \{B, P\}$ with designated *null* vertex.

Ensure: The annotation *ann* represents the balancing correctly if it is in $\{-1, 0, 1\}$ and is *undefined* otherwise.

for each $e \in E^T$ **do**
  $ann(e) \leftarrow undef$
end for

$height(null) \leftarrow -1$
for each $e \in E^N$ **with** $lab(e) = B$ **do**
  $height(att(e, 1)) \leftarrow count(ind(\mathrm{e}))$
end for

for each $v \in V$ **with** $height(v) \neq undef$ **and** $height(sibling(v)) \neq undef$ **do**
  if $height(v) - height(sibling(v)) \in \{-1, 0, 1\}$ **then**
    $ann(edge\_to(v)) \leftarrow height(v) - height(sibling(v))$
    $ann(edge\_to(sibling(v))) \leftarrow height(sibling(v)) - height(v)$
  else
    $ann(edge\_to(v)) \leftarrow undef$
    $ann(edge\_to(sibling(v))) \leftarrow undef$
  end if
  $height(v.parent) \leftarrow max\{height(v), height(sibling(v))\} + 1$
  $height(v) \leftarrow undef$
  $height(sibling(v)) \leftarrow undef$
end for

For a proper implementation, we might represent this by an accordingly labelled unary edge attached to the corresponding vertex. This is possible because even during execution this information is never changed to values other than $\{-1, 0, 1\}$. One possibility is to store a 1 if the right sub-tree is deeper and 0 if they are balanced and -1 otherwise. Further, the correct labels, i.e. the labels corresponding to the respective balancing, can be created and abstracted by the grammar exactly like the annotations we created for the purpose of providing a path grammar. Annotations and program accessible balance information is somewhat redundant, but while the latter is specialised for AVL-trees with very limited operations, e.g. insertion is only possible at leaves, the first is applicable to arbitrary balanced tree programs.

Figure 22 shows how our rules for balanced trees can be adopted so that they incorporate the balancing data used by the operations on AVL trees. The same mechanism is extended to all other rules of this grammar.

In the following examples we denote this data by writing it inside the respective vertex instead of displaying the edge to increase readability. For the same reason we omit parent pointers. They can always be assumed to mirror the displayed left- and right-pointers.

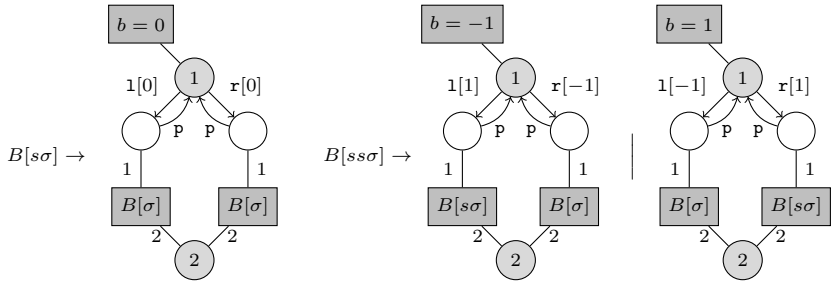


Fig. 22: An adoption of the grammar in Figure 15 to AVL trees

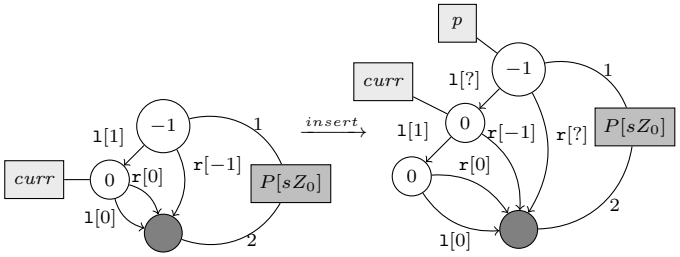


Fig. 23: AVL insertion: Step 1

Verification Objective The symbolic execution we consider does not check properties of the data, e.g. the search tree property for AVL trees, but only the structure of the heap, i.e. that it is still a balanced tree, that there are no memory leaks and so forth. Properties on the data can then be checked separately. This procedure is not uncommon, see for example [9,11,16].

Exemplary Symbolic Execution We now discuss on an example how a symbolic execution of an insertion algorithm on AVL trees works. The first graph in Figure 23 shows an initial configuration: The concrete sub-tree includes a leaf and is unbalanced to the left. Most of the tree is abstracted by the path grammar. A variable *curr* points at the leaf. In the following, an insertion procedure is executed on *curr* and we show that the program is indeed able to re-establish the AVL property of balancing.

In the first step a new vertex is inserted as a child of *curr* and assigned the balancing value zero. Also a variable p (for parent) is created and set to the parent of *curr*. Since the tree structure is not violated we also apply Algorithm 1 to get the annotations right. Some of them remain undefined (denoted by [?]) since the height difference is too large. Note that the inability to abstract at this point also indicates that the tree is not balanced.

Due to the introduction of the p-variable, we have to re-establish admissibility by concretising at the current root. This is necessary since the P non-terminal

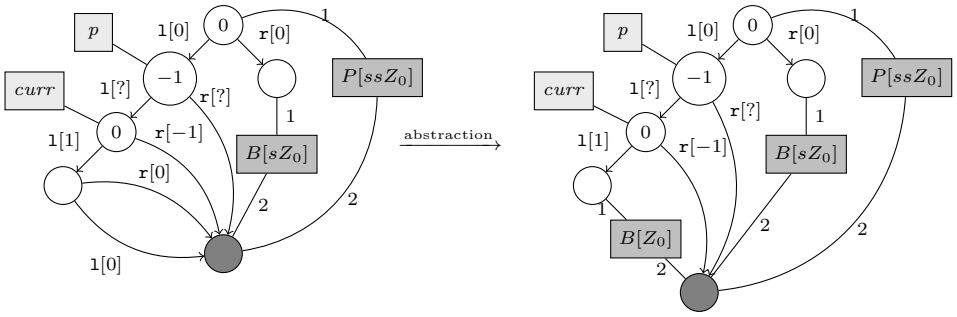


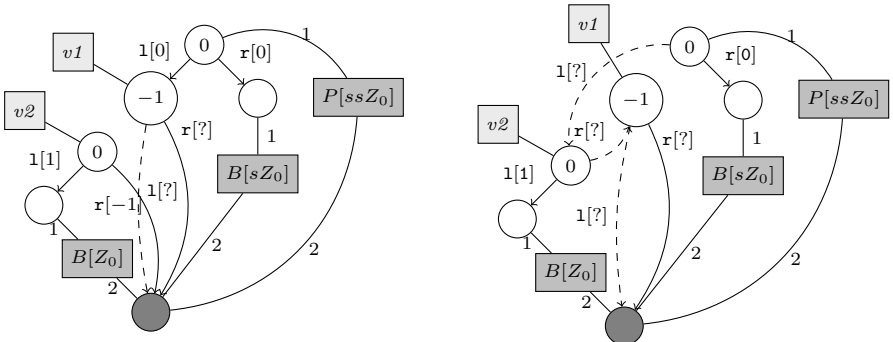
Fig. 24: AVL insertion: Concretisation and abstraction after step 1

creates an out-going p-edge to this vertex and since a variable is pointing to it. According to the path grammar we have twelve different possibilities for this concretisation (six of them shown in Figure 18, the other six being symmetric). For a valid verification one now analyses all twelve cases in parallel. In our example we just continue with the second rule in Figure 18, but provide remarks on what happens in the other cases. To complete this step we abstract the graph as much as possible without violating admissibility. The resulting graphs after concretisation and abstraction are shown in Figure 24.

```
1  void rightRotate( vertex v1 ){
2      vertex v2 = v1.left;
3      v1.left = v2.right;
4      if( v1.left != null ){
5          v1.left.parent = v1;
6      }
7      v2.parent = v1.parent;
8
9      //not at the root
10     if( v1.parent != null ){
11       if( v1 == v1.parent.right ){
12           v2.parent.right = v2;
13       }else{
14           v2.parent.left = v2;
15       }
16     }
17     v2.left = v1;
18     v1.parent = v2;
19 }
```

Listing 1: Right rotation on AVL trees

In the next phase the new balancing is propagated upwards. Since a vertex has been inserted to the left of the vertex pointed to with *curr*, its balancing shifts to the left, i.e. one is subtracted from its current value. The same is applied to the parent pointer, but since its balancing is already -1 a rebalancing takes place instead. Since the increased child is to the left (checked by *curr == p.left*) and since p is also a left child (checked by *p == p.parent.left*) a right rotation is applied to p. Listing 1 shows the code for this rotation (On the basis of Cormen et al. [3, p.313]). The function *rightRotate* is called with the vertex p as argument.



(a) After line 6

(b) After right rotate

Fig. 25: Right rotation

Figure 25a shows the abstract heap after the first 6 lines of *rightRotate(p)* have been applied. The left pointer of *v1* is now pointing at *v2.right* which is the *null* vertex, therefore, the code inside the *if*-clause is not executed (Otherwise the parent pointer would be set correctly). Note that at this point the tree structure is broken. This means that we cannot apply Algorithm 1 but this does not influence our approach. As a consequence we can analyse the rotation itself and do not consider it as an atomic operation. The other statements of the rotation procedure are executed in the same manner, resulting in a configuration that is presented in Figure 25b. At this point the heap is again a valid tree and we can therefore re-compute the annotations.

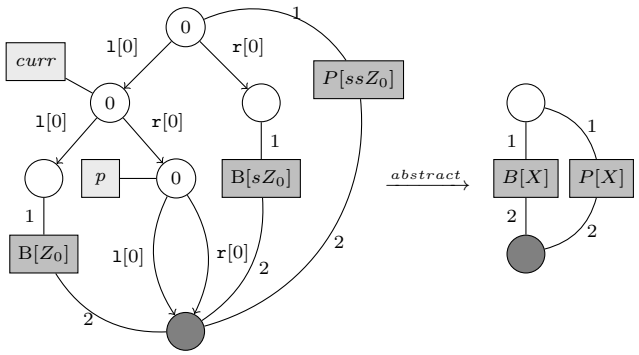


Fig. 26: AVL insertion: Last steps

As a last step the insertion procedure sets the balancing information of the vertices *curr* and p to 0 since this is the result of a right rotation. The tree is therefore once again balanced and the pointers *curr* and p are discarded.

The symbolic execution of this branch is concluded by abstracting the obtained graph as much as possible. These last two steps can be seen in Figure 26. The resulting graph consists of each a B and a P edge. By definitions of the grammars this corresponds exactly to the set of balanced trees and the program has therefore successfully re-established the balancing property.

Remember that a real symbolic execution would also consider all other rules in the concretisation step. In some of those cases other rotations are applied and some also trigger further upwards-propagations. When upwards-propagating, we always abstract parts below the vertices *curr* and p and by using index abstraction we will at some point create no new graphs any more. Therefore, also in this case the symbolic execution terminates.

8.7 Discussion

As we have seen, indexed HRGs are capable of expressing balanced trees as they are necessary to verify that AVL algorithms preserve this property. In contrast to Indian parallel HRGs, they also have an abstraction mechanism that allows us to consider an arbitrary sub-tree and abstract the rest of the tree. Still, indexed HRGs also have weaknesses: It is probably not possible to eliminate non-productive branches by eliminating non-productive rules beforehand, but one always has to perform an emptiness check before applying a rule if one wants to ensure that the graph is productive. It is further most likely not possible to hold several disconnected parts of balanced trees concrete at the same time, e.g. it is not possible to fix the root vertex and some deep sub-tree without remembering the whole path in between.

9 Conclusion and Future Work

We have introduced three variants of context-free hyperedge replacement grammars and analysed some of their basic properties, e.g. decidability of the emptiness problem as well as their potential to extend an existing heap abstraction framework.

We have one negative and two positive results. We conclude that Indian parallel grammars, although capable of expressing balanced trees, are not useful for heap abstraction, since they do not provide an abstraction mechanism that at the same time enables abstraction to a reasonable degree and on the other hand preserves the intended language.

On the other hand, we saw that automata controlled HRGs have the potential to contribute to heap abstraction by providing the means to ensure backward-confluence, i.e. to meet the objective that there is a unique fully abstracted graph for any input graph. Further research on automata controlled HRGs has to clarify whether an appropriate automaton can be constructed for any HRG.

The most promising candidate studied in this thesis are indexed HRGs. They can express balanced trees, such as AVL trees, and in contrast to Indian parallel grammars they provide a sensible abstraction mechanism. Although they do not preserve all positive properties of context-free HRGs, it seems reasonable that they can be used in practical verification. The approach is not restricted to balanced trees. An interesting question for further research is therefore whether there are other practically relevant hypergraph languages expressible by indexed HRGs.

For both automata controlled and indexed HRGs we have not analysed all properties relevant for heap abstraction. Further research is therefore necessary on this point. In case of a positive result it would be interesting to include these into Juggrnaut [11], a tool that uses heap abstraction by context-free HRGs for the verification of pointer programs.

References

1. Alfred V. Aho. Indexed grammars - an extension of context-free grammars. *J. ACM*, 15(4):647–671, 1968.
2. Y. Bar-Hillel, M. Perles, and E. Shamir. On formal properties of simple phrase structure grammars. z. phonetik, sprachwissen. komm. 15 (i961), 143-172. *Y. Bar-Hillel, Language and Information, Addison-Wesley, Reading, Mass*, pages 116–150, 1965.
3. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.
4. J. Dassow and G. Păun. *Regulated Rewriting in Formal Language Theory*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag Berlin Heidelberg, 1989.
5. Jürgen Dassow, Gheorghe Păun, and Arto Salomaa. Grammars with controlled derivations. In Grzegorz Rozenberg and Arto Salomaa, editors, *Linear Modeling: Background and Application*, volume 2 of *Handbook of Formal Languages*, pages 101–154. Springer Berlin Heidelberg, 1997.
6. Joost Engelfriet. Context-free graph grammars. In Grzegorz Rozenberg and Arto Salomaa, editors, *Beyond Words*, volume 3 of *Handbook of Formal Languages*, pages 125–214. Springer Berlin Heidelberg, 1997.
7. Jerome Feder. Plex languages. *Inf. Sci.*, 3(3):225–241, 1971.
8. Annegret Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *Lecture Notes in Computer Science*. Springer, 1992.
9. Peter Habermehl, Radu Iosif, and Tomás Vojnar. Automata-based verification of programs with tree updates. *Acta Inf.*, 47(1):1–31, 2010.
10. Michael A. Harrison. *Introduction to Formal Language Theory*, chapter 1.5 Other Families of Grammars - The Chomsly Hierarchy, pages 17–23. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1978.
11. Jonathan Heinen, Christina Jansen, Joost-Pieter Katoen, and Thomas Noll. Juggrnaut: using graph grammars for abstracting unbounded heap structures. *Formal Methods in System Design*, 47(2):159–203, 2015.
12. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - (2. ed.)*. Addison-Wesley series in computer science. Addison-Wesley-Longman, 2001.

13. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*, chapter 14.3 Indexed Languages, pages 389–390. Addison-Wesley, 1979.
14. Hans-Jörg Kreowski. Five facts of hyperedge replacement beyond context-freeness. In Zoltàn Ésik, editor, *Fundamentals of Computation Theory*, volume 710 of *Lecture Notes in Computer Science*, pages 69–86. Springer Berlin Heidelberg, 1993.
15. Sebastian Maneth. Cooperating distributed hyperedge replacement grammars. *Grammars*, 1(3):193–208, 1999.
16. Zohar Manna, Henny B. Sipma, and Ting Zhang. Verifying balanced trees. In Sergei N. Artëmov and Anil Nerode, editors, *Logical Foundations of Computer Science, International Symposium, LFCS 2007, New York, NY, USA, June 4-7, 2007, Proceedings*, volume 4514 of *Lecture Notes in Computer Science*, pages 363–378. Springer, 2007.
17. Radu Rugina. Quantitative shape analysis. In Roberto Giacobazzi, editor, *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*, volume 3148 of *Lecture Notes in Computer Science*, pages 228–245. Springer, 2004.

A Basic Definitions and Notations

DFA: A deterministic finite automaton (DFA) $\mathcal{A}$ is a tuple (Q, Σ, δ, F) where Q is the set of states, Σ is a finite alphabet and $F \subset Q$ is the set of accepting states. The partial function $\delta : Q \times \Sigma \to Q$ is the transition function that defines for each state and each read element of the alphabet a new state $\delta(q, a) = q'$. For words $w = a_1...a_n \in \Sigma^*$ we write $\delta^*(q, w) = q'$ for $\delta((...(\delta(q, a_1))...), a_n) = q'$, i.e. the state that is reached when starting at q and reading w.
By $\mathcal{A}_q$ with $q \in Q$ we denote the automaton $\mathcal{A}$ with initial state q. We say $\mathcal{A}_q$ *accepts* $w \in \Sigma^*$ if and only if $\delta^*(q, w) \in F$. The *language* $\mathcal{L}(\mathcal{A}_q)$ of such an automaton is $\{w \in \Sigma^* \mid \mathcal{A}_q \text{ accpets } w\}$.

Functions: For a function $f : D \to I$ and a subset $S \subset D$ $f \restriction S$ is the restriction of f to S. We abbreviate a function $f : \{x_1, \ldots, x_n\} \to \{v_1, \ldots, v_n\}$ with $f(x_i) = v_i$ for all $i \in \{1, ..., n\}$ by $\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$.

Right Regular Grammar [10]: A context-free grammar $G = (N, T, P)$ is called right regular if all rules $p \in P$ are of the form

$$\begin{aligned} & X \to aY && X, Y \in N \\ \text{or} \quad & X \to a && a \in T \end{aligned}$$

Sequences: A finite set A of symbols is called an *alphabet*. A sequence $w = a_1 \ldots a_n$ with $a_i \in A$ for all $i \in \{1, \ldots, n\}$is a word over A. n is the length of the sequence and denoted by $|w|$. The empty word is denoted by ϵ. The set of all words over A *including* the empty word is denoted by A^*. If the empty word is *excluded*, this is denoted by A^+. For a word $w \in A^*$ we denote by $[w] \subseteq A$ the set of elements from A occurring in w.

B Normal Form for Indexed Derivations

Lemma 6. *For all $H \in iHG_{N \cup T}$, $p \in P$ and $q \in P^i$ holds: If $H \Rightarrow_p H' \Rightarrow_q K$ is a valid derivation of K then $H \Rightarrow_q H'' \Rightarrow_p K$ is too.*

Proof. Let $H \Rightarrow_p H' \Rightarrow_q K$ be a valid derivation in some indexed grammar (G, G^i), where $p = X[f_1 \ldots f_n \sigma] \to K^\sigma \in P$ and $q = X^i \to \alpha \in P^i$.

As p is applicable to H there is an edge e in H with $lab(e) = X$ and $ind(e) = f_1 \ldots f_n g_1 \ldots g_m$ that is replaced by p. Then in the second step q is applicable, it must hold that all stacks in the resulting graph end in X^i, i.e. $\forall e' \in E_{H'} : ind(e) \in F^* \cdot X^i$. Since $H' \cong H[K^{g_1 \cdots g_m/\sigma}/e]$ it follows that $\forall e' \in E_H : ind(e) \in F^* \cdot X^i$ (For $E_H \backslash e$ this is obvious since their index does not change during edge replacement and as the indices in $K^{[g_1 \cdots g_m/\sigma]}$ are either fully concrete (which cannot happen since q is applicable) or end with g_m it follows that $g_m = X^i$. Therefore, q is already applicable to H.

Next, we have to argue why p is applicable to H''. As $H \Rightarrow_q H''$ we have an isomorphism h from H to H' with $\forall e' \in E_H : ind_H(e') \cdot \alpha = ind_{H''}(e')$ and in particular $ind_{H''}(h(e)) = f_1 \ldots f_n g_1 \ldots g_{m-1} \alpha$, where e is the edge that has been

replaced in the first derivation. Therefore, p is applicable to $h(e)$ in H'' and as by Definition 31 there are no side effects between derivations. Hence, the lemma is shown. □

The following corollary follows by repeated application of Lemma 6:

Corollary 1. *For all indexed HRGs $G = (N, T, P, F)$, right regular stack grammars $G^i = (N^i, F, P^i)$ and graphs $H \in iHG_{N \cup T}$ the language $\mathcal{L}(H)$ is equal to $\{K \mid \exists q_1, \ldots, q_n \in P^i \; \exists p_1, \ldots, p_m \in P : H \Rightarrow_{q_1} \ldots \Rightarrow_{p_m} K\}$.*