Software Modeling and Verification
Informatik 2: Prof. Dr. Ir. Joost-Pieter Katoen

Rheinisch-Westfälische Technische Hochschule Aachen
Lehrstuhl für Informatik 2
Software Modeling and Verification
Prof. Dr. Ir. Joost-Pieter Katoen

Automata-Based Detection of Hypergraph Embeddings

Henrik Barthels

Matrikelnummer 289773

1. Gutachter: Prof. Dr. Rudolf Mathar
2. Gutachter: Priv.-Doz. Dr. Thomas Noll
Betreuer: Dipl.-Inf. Jonathan Heinen

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht.

Aachen, den 28.09.2011 Henrik Barthels

Abstract The verification of programs using pointers and dynamic data structures requires to deal with potentially infinite state spaces. Because of that, it is reasonable to use abstraction techniques capable of dealing with those potentially infinite structures. The Juggrnaut framework applies hyperedge replacement grammars to dynamically abstract and concretize parts of a heap. Abstraction is done by the backwards application of grammars rules, which is related to subgraph isomorphism and therefore NP-complete. In this thesis, after giving a formal definition to hypergraphs, hyperedge replacement and heap representation, an automata model is introduced which is able to detect embeddings of grammar rules with certain properties efficiently. We provide an algorithm to construct an automaton that is able to detect a given set of embeddings of grammar rules. Finally, proofs of the NP-completeness of subgraph isomorphism on hypergraphs and embedding detection in general are presented.

Acknowledgments

Even though I will be held responsible for this thesis, it wouldn’t have come into existence (at all or in this state) without the following people who don’t deserve to remain unmentioned.

Thank you, my parents, for an exquisite cuisine. Thank you, Friederike, for your literarily opinion on the English language. Thank you, Lisa, for being one of those responsible for sparking my interest in computer science, and everything else. Thank you, Jonathan, for countless hours of productive, insightful and entertaining discussion. In order to avoid casting a bad light on us, I feel obliged to add that it was productive for the vast majority of the time. I owe you at least one inch of red pencil. Thank you, Sabrina and Christina, for not throwing us out of your office. Thank you, Prof. Dr. Mathar and Priv.-Doz. Dr. Noll, for allowing me to write this thesis.

Contents

1 Introduction

With increasing complexity and widespread usage of software, the correctness of software becomes increasingly important and at the same time more difficult to verify. Especially programs using dynamic memory allocation and deallocation, so called pointer programs, are prone to a variety of errors, for example memory leaks and null pointer dereferencing. Therefore, it is desirable to have tools to automatically detect such errors before the programs are used.

At the Software Modeling and Verification Group, a framework called *Juggrnaut (Just Use Graph GRammars for Nicely Abstracting Unbounded sTructures)* is developed which aims at proving the correctness of Java programs that use pointers.

The state space of those programs, while in fact finite due to the limited size of memory of computers, is in general too large to be fully explored. Thus, it is reasonable to assume it infinite and apply techniques that do not rely on the boundedness of the state space.

Juggrnaut uses hypergraphs to simulate the heap of a given program. However, it is in most cases not necessary to represent the whole heap in detail while manipulations by the program just have a local effect and conversely only local properties affect the execution of the program. Hence, we partially abstract from the concrete representation by using *hyperedge replacement grammars* (HRGs). Those grammars consist of rules which can be used to describe certain data structures, for example trees or lists. Nonterminal edges represent abstract parts of the heap, but retain the information of which structure this part is. Terminal edges symbolize specific pointers or variables. HRGs describe how nonterminal edges can be replaced by hypergraphs which are more concrete representations of certain data structures, which may contain further nonterminal edges.

The rules of HRGs can then be used to concretize and abstract the heap representation by forward and backward application, respectively.

In order to apply a rule in the reverse direction, first we need to find an embedding of the rule graph, which is then replaced by a hyperedge. While the forward application can be computed efficiently, finding such embeddings turns out to be NP-complete in general. Therefore, exploiting the special properties of heaps and commonly used data structures, it is crucial to develop algorithms which are able to complete this task as efficient as possible at least for a subset of instances.

Our aim is to develop an algorithm that solves this problem for restricted HRGs and is efficient for grammars describing common data structures. Due to the nature of heap representation as employed by *Juggrnaut*, it is not only desirable to have an algorithm to detect a single embedding, but one that is able to detect multiple embeddings at once, preferably benefiting from similarities between grammars rules. This leads to the idea of using automata, such that it is possible to construct an automaton for every single rule and then compute the union of all those automata.

This thesis is organized as follows. In the preliminaries, we will define hypergraphs and HRGs in general. Then, more restricted *heap configurations* are introduced, which represent feasible heaps, as well as *data structure grammars* (DSGs), which are defined in a way that the replacement of grammar rules does not lead to heap configurations which do not represent real heaps. Finally, we will define the process of abstraction and concretization of heap representations in terms of DSGs.

In the next part, the notion of paths on heap configurations is defined, as they allow to uniquely identify vertices of a heap configuration relative to another one. Those paths then can be used to give a different set of requirements for an embedding in a heap configuration, where it is only necessary to check properties of vertices reach by certain path from one fixed initial vertex. That is, however, only possible for grammar rules where there is one vertex from which every other one can be reached by a path. We then define an automata model, the so called *function automaton*, that is able to check properties of a heap configurations based on paths. In a next step, an algorithm for constructing *embedding detection automata* from a given set of DSG embeddings will be provided. Finally, we will give an analysis of the time complexity of constructing and using those embedding detection automata.

The fourth chapter consists of two proof. The first one shows that subgraph isomorphism is still NP-complete when extended to hypergraphs. In the second proof, we show that finding embeddings in heap configurations in general is NP-complete, too.

2 Preliminaries

We are going to start with an overview of the notations used in this thesis and definitions of the necessary objects, structures and grammars. It follows the definitions of [4].

Given a set S, S^* is the set of all finite sequences (strings) over S including the empty sequence ε. * is commonly known as the Kleene star operator. For $s \in S^*$, the length of s is denoted by $|s|$, the set of all elements of s is written as $[s]$, and by $s(i)$ we refer to the i-th element of s.

Given a tuple $t = (A, B, C, \dots)$ we write $A_t, B_t, \dots$ for the components.

Function $f \upharpoonright S$ is the restriction of f to S. Functions $f : A \to B$ are lifted to sets $f : 2^A \to 2^B$ and to sequences $f : A^* \to B^*$ by point-wise application. We denote the identity function on a set S by id_S.

2.1 Hypergraphs

Informally, a hypergraph can be described as a graph with edges connecting an arbitrary number of vertices, including being connected to only one vertex. Consequently, common graphs with edges connecting exactly two vertices are a special case of hypergraphs. The following definition is oriented towards the definition of hypergraphs in [5].

Definition 2.1.1 (Hypergraph)
Let Σ be a finite ranked alphabet where $\mathrm{rk} : \Sigma \to \mathbb{N}$ assigns to each symbol $a \in \Sigma$ its rank $\mathrm{rk}(a)$ and Λ a finite set of types. A (labeled and annotated) *hypergraph* over Σ and Λ is a tuple $H = (V, E, \mathrm{att}, \mathrm{lab}, \mathrm{type}, \mathrm{ext})$ where V is a set of vertices and E a set of hyperedges. The attachment function $\mathrm{att} : E \to V^*$ maps each hyperedge to a sequence of attached vertices, $\mathrm{lab} : E \to \Sigma$ is a hyperedge-labeling function, $\mathrm{type} : V \to \Lambda$ a vertex-annotation function, and $\mathrm{ext} \in V^*$ a (possibly empty) sequence of pairwise distinct external vertices.

For $e \in E$, we define the rank of edge e, $\mathrm{rk}(e) = |\mathrm{att}(e)|$ and require that $\mathrm{rk}(e) = \mathrm{rk}(\mathrm{lab}(e))$. Let $\Gamma = (\Sigma, \Lambda)$. The set of all hypergraphs over Σ and Λ is denoted by HG_Γ. ■

For now, the distinction between external and internal vertices can be ignored, as it is only needed for hyperedge replacement grammar. Note that this definition allows edges to be connected more than once to the same vertex and multiple hyperedges which are attached to the same sequence of vertices and have the same labeling.

Example 1 (Hypergraph)
Let $H \in \mathrm{HG}_\Gamma$, with $H = (V_H, E_H, \mathrm{att}_H, \mathrm{lab}_H, \mathrm{type}_H, \mathrm{ext}_H)$ as shown in Figure 2.1. Hypergraph H is depicted in Figure 2.2. Vertices are represented by circles, edges by rectangles. External vertices are colored red, the numbers next to them denote their position in the sequence of external vertices.

$$
\begin{aligned}
V_H &= \{v_1, v_2, v_3, v_4, v_5\} \\
E_H &= \{e_1, e_2, e_3, e_4\} \\
ext_H &= v_2 v_5 \\
\Sigma &= \{X, y, z\} \\
\Lambda &= \{a, b, c, d\}
\end{aligned}
$$

e_i	$att_H(e_i)$	$lab_H(e_i)$
e_1	$v_1v_2v_4v_4$	X
e_2	v_2v_3	z
e_3	v_2v_3	z
e_4	v_5	y

v_i	$type_H(v_i)$
v_1	a
v_2	b
v_3	c
v_4	d
v_5	d

Figure 2.1: Hypergraph H.

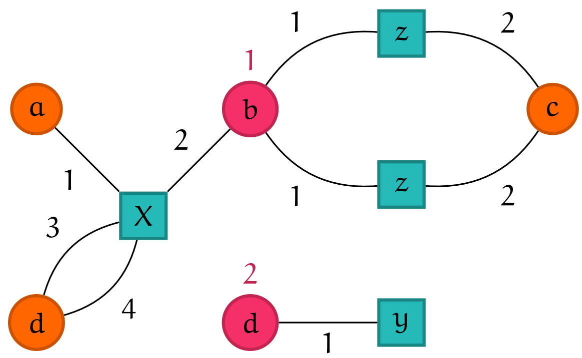

Figure 2.2: Example of a Simple Hypergraph.

We call the connections between hyperedges and vertices *tentacles* and specify them as pairs of the label of the edge they are connected to and the position of the vertex within the attachment sequence.

Definition 2.1.2 (Tentacle)
Let Σ be an alphabet. A *tentacle* is defined as a pair (X, i) with $X \in \Sigma$ and $i \in [1, rk(X)]$. The *set of all tentacles* over Σ is defined as $T_\Sigma = \{(x, n) \mid x \in \Sigma, n \in [1, rk(x)]\}$. ■

Two hypergraphs are *isomorphic* if they are equal under renaming of vertices and edges.

Definition 2.1.3 (Hypergraph Isomorphism)
Let $G, H \in HG_\Gamma$. G and H are *isomorphic* if there are bijective functions $f_V : V_G \to V_H$ and $f_E : E_G \to E_H$ such that

$$
\begin{aligned}
lab_G(e) &= lab_H(f_E(e)), \forall e \in E_G \\
type_G(v) &= type_H(f_V(v)), \forall v \in V_G \\
att_H(f_E(e)) &= f_V(att_G(e)), \forall e \in E_G \\
ext_H &= f_V(ext_G)
\end{aligned}
$$

■

2.2 Heap Representations

The heap of a program usually consists of a number of instances of classes, called objects, which are related to each other and the program by pointers. More general, they are a set of objects and a set of relations between those object. Therefore, an intuitive way to represent those heaps are graphs.

In *Juggrnaut*, hypergraphs are used. Objects are represented by vertices. For pointers, we use edges of rank two, which point from the first attached object to the second one. Those edges are called *selectors*. Pointers from program variables to objects on the heap are denoted by hyperedges of rank one. They are attached to the object on the heap they point to. We refer to them simply as *variables*.

Formally, we assume that $\Sigma = \mathrm{Var}_\Sigma \uplus \mathrm{Sel}_\Sigma$, a finite ranked alphabet, is the disjoint union of a set Var_Σ of variables and a set Sel_Σ of selectors.

In the programming language Java, every object has a *type*. To reflect that, we annotate every vertex of a hypergraph with a type from Λ.

Example 2 (Heap Representation)
A simple list could consist of list elements which have *next* pointers to their successors and program variables which point to the first and last element of the list, called *head* and *tail*.

In order to represent a heap containing such a list, we require that $\mathrm{Var}_\Sigma = \{\mathrm{head}, \mathrm{tail}\}$ and $\mathrm{Sel}_\Sigma = \{\mathrm{next}\}$. Moreover, an annotation for the list elements is needed. For the sake of simplicity, we call it $\mathfrak{l}$ and consequently obtain the set of types $\Lambda = \{\mathfrak{l}\}$.

Figure 2.3 gives an example of a representation of a simple list over the alphabet $\Sigma = \mathrm{Var}_\Sigma \uplus \mathrm{Sel}_\Sigma$. Variables are colored green, selectors blue. Tentacles are labeled with their ordinal number, vertices with their types.

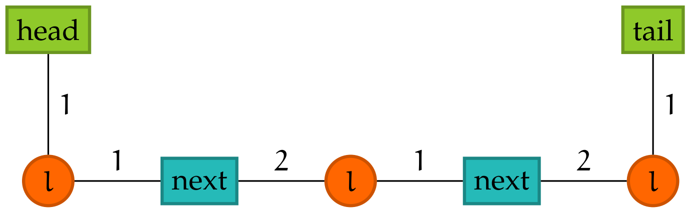

Figure 2.3: Example of a Simple Heap.

The possible selectors of a type are given by the classes of the inspected program. We will refer to them with the following function:

Definition 2.2.1 (Vertex Type Selectors)
Let $\Gamma = (\Sigma, \Lambda)$. The sequence of *vertex type selectors* is defined by the function $\mathrm{pt} : \Lambda \to \mathrm{Sel}_\Sigma^*$, mapping a type to a (possibly empty) sequence of pairwise distinct selectors. ∎

Additionally, we define the type of a label as a sequence of types, with one type for each of its tentacles. To avoid conflicts between equally named selectors of different types, we require without loss of generality that selectors of different types are uniquely named. That can easily be achieved by adopting the notation of Java, for example a selector named *next* belonging to the class *listelement* is named *listelement.next*.

2.3 Heap Configurations

As hypergraphs do not impose any restrictions on the outgoing selectors of vertices and the names of variables, they are too permissive for modeling heap structures. In common programmings languages, no two objects with the same name are allowed within one scope. Moreover, every object has exactly those pointers which were specified in the class definition, and they may be null if they do not point to another object. In Figure 2.3, the outgoing selector of the last element of the list, which would be a null pointer in a real heap, was simply omitted, ignoring that a list object always has to have a *next* pointer if it is defined like that. It requires us to limit *heap configurations* to hypergraphs with unique variables and objects which have exactly one outgoing selector s for every $s \in [\mathrm{pt}(\mathrm{type}(v))]$ and $v \in V_H$. Furthermore, in order to be able to represent null pointers, we introduce a null object to which all null pointers point.

Definition 2.3.1 (Type Sequence of Edges)
Let $G \in HG_{\Gamma_N}$, $e \in E_G$ and $\mathrm{lab}(e) = s \in \Sigma_N$. The function $\mathrm{ts} : \Sigma_N \to (\Lambda \cup \Lambda_0)^*$, with $\Lambda_0 = \{a_0 \mid \forall a \in \Lambda\}$ maps each label to a sequence of types, with $|\mathrm{ts}(s)| = \mathrm{rk}(s)$. Furthermore, we define $\forall a \in \Lambda : a =_0 a_0$. ■

With this definition of a type sequence, we also refer to $\mathrm{ts}(X)(i)$ as the type of a (X, i) tentacle. We distinguish between Λ and Λ_0 as there will be two different kinds of tentacles. Λ contains the so called *non-reduction types*. Vertices attached to *non-reduction tentacles*, which are tentacles with a non-reduction type, can be left at this tentacle. Similarly, at least in case of selectors, vertices can only be entered at *reduction tentacles*, which have *reduction types* from Λ_0. Vertices can be entered, too, via non-reduction tentacles of nonterminals. The $=_0$ relation is introduced to denote that the types of two tentacles are the same, for example *listelement*, except for the reduction property.

Definition 2.3.2 (Concrete Heap Configuration)
A *concrete heap configuration* over $\Gamma = (\Sigma, \Lambda)$ is a tuple $H = (V, E, \mathrm{att}, \mathrm{lab}, \mathrm{type}, \mathrm{ext}, v_{\mathrm{null}})$ with $(V, E, \mathrm{att}, \mathrm{lab}, \mathrm{type}, \mathrm{ext}) \in HG_\Gamma$. We require $\{v \in V \mid \mathrm{type}(v) = \bot\} = \{v_{\mathrm{null}}\}$, $\bot \in \Lambda$, and $\mathrm{pt}(\bot) = \varepsilon$. Additionally, the following conditions have to be met:

(1) $\forall a \in \mathrm{Var}_\Sigma : |\{e \in E_H \mid \mathrm{lab}(e) = a\}| = 1$

(2) $\forall v \in V_H, \forall s \in [\mathrm{pt}(\mathrm{type}(v))] : |\{e \in E_H \mid \mathrm{lab}(e) = s \wedge \mathrm{att}(e)(1) = v\}| = 1 \wedge |\mathrm{pt}(\mathrm{type}(v))| = |\{e \in E_H \mid \mathrm{att}(e)(1) = v\}|$

(3) $\forall e \in E_G, i \in [1, \mathrm{rk}(e)] : \mathrm{type}(\mathrm{att}(e)(i)) \neq_0 \mathrm{ts}(\mathrm{lab}(e))(i) \Rightarrow \mathrm{ts}(\mathrm{lab}(e))(i) \in \Lambda_0 \wedge \mathrm{att}(e)(i) = v_{\mathrm{null}}$

We denote the set of all heap configurations over Σ and Λ by HC_Γ, with $\Gamma = (\Sigma, \Lambda)$. ■

With (1), it is guaranteed that variables are defined uniquely. From (2), it follows that vertices have exactly those pointers given by their type. (3) ensures that edges are only attached to vertices of the type specified by the type sequence. If the corresponding type in the type sequence is an element of Λ_0, this tentacle is additionally allowed to be attached to v_{null}.

Over heap configurations, we can conveniently define *outgoing edges* of vertex v as those edges labeled with a selector that are attached to v with their first tentacle. Since variables represent pointers from the program to the heap, their only tentacle represents an incoming pointer to the vertex they are attached to, while their origin in the program is not specified.

Definition 2.3.3 (Outgoing Selectors)
Let $G \in HC_\Gamma$, $v \in V_H$. The set of *outgoing edges* at vertex v in G is defined as:
$\mathrm{out}(v) = \{e \in E_G \mid \mathrm{lab}(e) \in \mathrm{Sel}_\Sigma \wedge \mathrm{att}(e)(1) = v\}$ ■

The definition of heap configurations does not impose any restrictions on connectivity. However, we assume that parts of a graph not reachable from any variable, so called *garbage*, has previously been removed, since it has no influence on the behavior of the program. The null object is the only exception, as it can be accessed in any scope, even if it is not reachable by any pointers in the current state of the program.

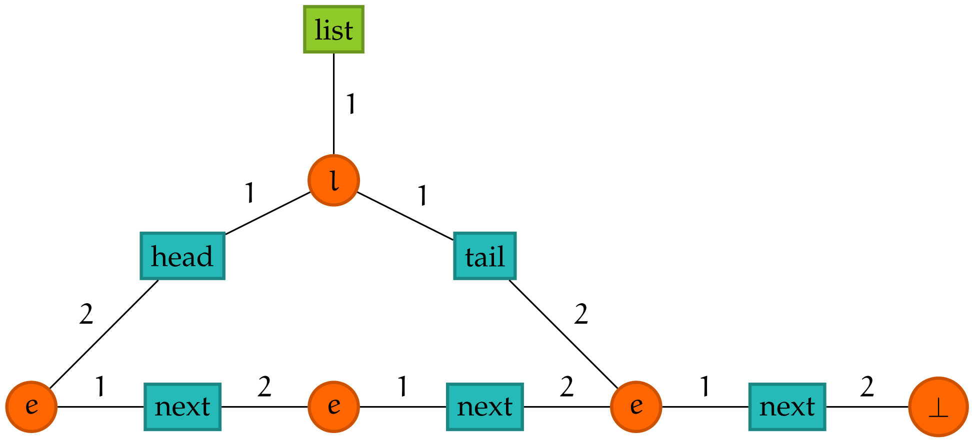

Figure 2.4: Example of a Heap Configuration.

Example 3 (Concrete Heap Configuration)
Consider Figure 2.4 for an example of a simple heap configuration. It shows a typical implementation of a list, which consist of a list object which has pointers on the first and last element of the list, labeled *head* and *tail*. The *list* program variable references the list object. Every list object has a pointer labeled *next* pointing on the next list object. The list object type is denoted by $\mathfrak{l}$, e the list element type and $\perp$ the type of the null object.

2.4 Hyperedge Replacement Grammars

As heaps of real world application programs are in general too large to be fully represented and theoretically unbounded, *Juggrnaut* introduces an abstraction mechanism by the usage of *hyperedge replacement grammars*. The idea is to represent parts of the heap which have a well known structure by a single non-terminal edge. Therefore, we extend the labeling alphabet with an additional set of *nonterminals*. Let N be a set of nonterminals of arbitrary rank. The new heap alphabet $\Sigma_N = \Sigma \cup N$ now contains variables, selectors and nonterminals.

The definitions of hyperedge replacement grammars and hyperedge replacement follow [5].

Definition 2.4.1 (Hyperedge Replacement Grammar)
A *hyperedge replacement grammar* (HRG) over alphabets $\Gamma_N = (\Sigma_N, \Lambda)$ is a set of production rules of the form $X \to H$, with $X \in N$ and $H \in HG_{\Gamma_N}$ where $|ext_H| = rk(X)$ and $ts(X)(i) =_0 type_H(ext_H(i))$ for all $i \in [1, rk(X)]$. We denote the set of hyperedge replacement grammars over Γ_N by HRG_{Γ_N}. ■

HRG rules are applied to hypergraphs by means of *hyperedge replacement*. A hyperedge labeled with a nonterminal is replaced by the hypergraph on the right hand side of one of the rules with the same nonterminal on the left hand side, analogous to context free grammars for strings. The external nodes of the replacement graph are merged with the nodes the hyperedge was previously attached to.

For hyperedge replacement, we require that the number of tentacles of the edge and the number of external vertices in the replacement graph is equal. Furthermore, the types of the external vertices and corresponding vertices in the graph have to be the same.

Definition 2.4.2 (Hyperedge Replacement)
Let $H, I \in HG_{\Gamma_N}$, $e \in E_I$ a nonterminal edge with $rk(e) = |ext_H|$. Without loss of generality, we assume that there are no vertices and edges of I and H which have the same name. If there are any, they have to be renamed. The *substitution* of e by H, $I[H/e] = J \in HG_{\Gamma_N}$, is defined by:

$$V_J = V_I \cup (V_H \setminus [ext_H]) \tag{2.1}$$
$$E_J = (E_I \setminus \{e\}) \cup E_H \tag{2.2}$$
$$lab_J = (lab_I \upharpoonright (E_I \setminus \{e\})) \cup lab_H \tag{2.3}$$
$$att_J = att_I \upharpoonright (E_I \setminus \{e\}) \cup mod \circ att_H \tag{2.4}$$
$$ext_J = ext_I \tag{2.5}$$
$$type_J = type_I \cup (type_H \upharpoonright (V_H \setminus [ext_H])) \tag{2.6}$$

with $mod = id_{V_H \setminus [ext_H]} \cup \{ext_H(i) \mapsto att_I(e)(i) \mid i \in [1, rk(e)]\}$. ■

Hyperedge replacement can now be used to replace nonterminals in a hypergraph with rule graphs of matching hyperedge replacement grammar rules, similar to string grammars. The definitions below follow the definitions of [4].

Definition 2.4.3 (HRG Derivation)
Let $G \in HRG_{\Gamma_N}$, $I, J \in HG_{\Gamma_N}$, $p = X \to H \in G$ and $e \in E_I$ with $lab(e) = X$. J *is derivable from* I *by* p if J is isomorphic to $I[H/e]$. For short, we write $I \xRightarrow{e,p} J$, $I \Rightarrow_G J$ if $I \xRightarrow{e,p} J$ for some $e \in E_I$ and $p \in G$. The reflexive-transitive closure is denoted by $\Rightarrow_G^*$. ■

Unlike string grammars, which contain a starting symbol, hyperedge replacement grammars do not contain a starting graph. Instead, a hypergraph is given as a parameter for the language of a hyperedge replacement grammar. The language of a HRG is the set of all hypergraphs without nonterminals which can be derived from the initial hypergraph.

Definition 2.4.4 (Language of a HRG)
Let $G \in HRG_{\Gamma_N}$, $I \in HG_{\Gamma_N}$. The *language of* G generated from I is defined as $L_G(I) = \{J \in HG_\Gamma \mid I \Rightarrow_G^* J\}$. ■

For the sake of simplicity, we depict selectors intuitively as directed edges from the tentacle with the lower ordinal to the tentacle with the higher ordinal, as shown in Figure 2.5, and we will omit the ordinal of tentacles of variables, as it is always one.

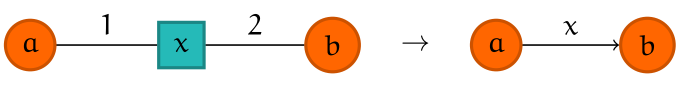

Figure 2.5: Simplified Depiction of Hyperedges.

Example 4 (Hyperedge Replacement)
Consider Figure 2.6 for a hypergraph G on the left hand side and a HRG consisting of the rule $X \to H$. Assume that the hyperedge labeled with X in G is called e. Obviously, $\mathrm{rk}(e) = |\mathrm{ext}_H|$ holds, and the types of the vertices match. Therefore, the hyperedge replacement $G[H/e]$ is possible. To obtain $J = G[H/e]$ shown in Figure 2.7, edge e in G has to be removed. Then, the first external vertex of H is merged with the vertex e was previously attached to with the first tentacle. The same is done for the remaining two external vertices. The edges between the vertices of H are then added to G.

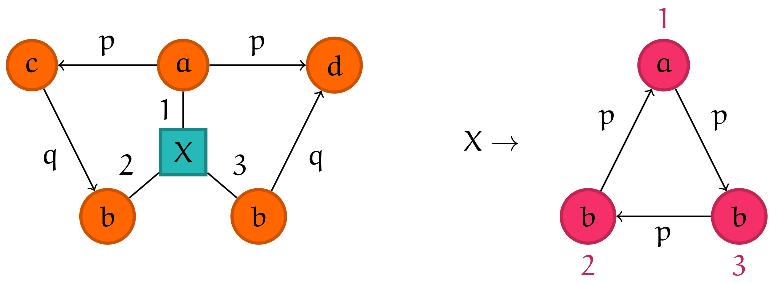

Figure 2.6: A Hypergraph G and a Hyperedge Replacement Grammar $X \to H$.

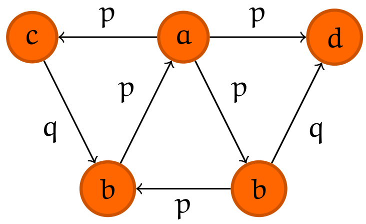

Figure 2.7: The Resulting Graph J of the Hyperedge Replacement $G[H/e]$.

2.5 Abstract Heaps

Since it is our goal to have an abstract representation of heaps, we define *abstract heap configurations* which allow this abstraction by hyperedges labeled with nonterminals of arbitrary rank.

We require that either none or all of the outgoing selectors of a vertex have to be abstracted in a single nonterminal. Thus, a vertex in a heap configuration either has all concrete pointers as required for this type, or it is attached to one non-reduction tentacle that abstracts exactly those pointers. This makes hyperedge replacement considerable easier, as it is not necessary to distinguish between nonterminals which abstract different subsets of all outgoing pointers of vertices with one type. This, however, has the disadvantage that the construction of grammar rules for some data structures becomes more complicated. Additionally, for any tentacle (X, i), we require that the attached vertices have the same type as the tentacle, unless it is the null type. Without that requirement, it would be possible to create heaps by hyperedge replacement where pointers point to objects of a wrong type, or heap configurations with outgoing pointers which do not match the type of the object.

The set of attached non-reduction edges of a vertex is the set of edges which represent outgoing selectors of a vertex, either concrete or abstracted. It is basically an extension of the notion of outgoing edges for abstract heap configurations. We need the set of non-reduction types Λ to identify the tentacles of those edges, based on their labels and the corresponding type sequences. To guarantee that this always works, the type sequence a selector $a \in Sel_\Sigma$ has to be defined as $ts(a) = bb_0$ with $b \in \Lambda$ and $b_0 \in \Lambda_0$. Since we defined selectors to represent pointers from the first to the second attached vertex, all edges representing outgoing pointers are in that set.

Definition 2.5.1 (Attached Non-Reduction Edges)
Let $G \in HC_{\Sigma_N}$. The set of *attached non-reduction edges* is defined as $\mathrm{nREdge}(v) = \{e \in E_G \mid \exists n \in [1, rk(e)] : att_G(e)(n) = v \wedge ts(lab_G(e))(n) \in \Lambda\}$. ∎

Note that for concrete heap configurations $H \in HC_\Sigma$, $\mathrm{nREdge}(v) = \mathrm{out}(v)$ for all $v \in V_H$. A vertex is called complete if it either has all concrete outgoing selectors according to its type, or if it is attached to exactly one nonterminal edge with a tentacle that abstracts all those selectors. This ensures that when replacing that nonterminal with a correct grammar rule, the resulting hypergraph is still a valid heap configuration.

Definition 2.5.2 (Complete Vertex)
Let $G \in HC_{\Gamma_N}$. $v \in V_G$ is a *complete vertex* if it either has all concrete outgoing tentacles, or one outgoing nonterminal tentacle which abstract all concrete ones.

$$\begin{aligned}\mathrm{complete}_G(v) = {} & (lab_G(\mathrm{nREdge}(v)) = [pt(type_G(e))] \wedge |\mathrm{nREdge}(v)| = |pt(type_G(e))|) \\ & \vee (lab_G(\mathrm{nREdge}(v)) \subseteq N \wedge |\mathrm{nREdge}(v)| = 1)\end{aligned}$$

∎

Using 'complete' and the extended version of the function 'out', we are able to define abstract heap configurations as a generalized version of concrete heap configurations, which allow us to fully represent abstracted heaps.

Definition 2.5.3 (Abstract Heap Configuration)
An *abstract heap configuration* over $\Gamma_N = (\Sigma_N, \Lambda)$ is a tuple $G = (V, E, att, lab, type, ext, v_{null})$ with $(V, E, att, lab, type, ext) \in HG_{\Gamma_N}$. We require $\{v \in V \mid type(v) = \perp\} = \{v_{null}\}$, $\perp \in \Lambda$, and $pt(\perp) = \varepsilon$. Additionally, the following conditions have to be met:

(1) $\forall a \in Var_{\Sigma_N} : |\{e \in E_G \mid lab(e) = a\}| = 1$

(2) $\forall v \in V_G : \mathrm{complete}_G(v)$

(3) $\forall e \in E_G, i \in [1, rk(e)] : type(att(e)(i)) \neq_0 ts(lab(e))(i) \Rightarrow ts(lab(e))(i) \in \Lambda_0 \wedge att(e)(i) = v_{null}$

We denote the set of all abstract heap configurations over Γ_N by HC_{Γ_N}, with $\Gamma_N = (\Sigma_N, \Lambda)$. ∎

The only difference to concrete heap configurations is that instead of having all outgoing pointers for that type, a vertex can be attached to one non-reduction tentacle of a nonterminal which abstracts all those outgoing pointers (2).

For abstract heap configurations, a new grammar, which we refer to as *data structure grammar*, and according replacement rules need to be introduced. This is necessary since we introduced a null vertex with a special null type and because we required in the definition of heap configurations that either all or none of the outgoing edges of a vertex have to be abstracted in one nonterminal. Therefore, external vertices are required to have either non or all of the outgoing selectors possible for that type.

Definition 2.5.4 (Data Structure Grammar)
A *data structure grammar* (DSG) over alphabets $\Gamma_N = (\Sigma_N, \Lambda)$ is a set of production rules of the form $X \to H$, with $X \in N$ and $H \in HG_{\Gamma_N}$ where $|ext_H| = rk(X)$. Additionally, the following conditions have to be satisfied:

(1) $\forall v \in V_H : \neg\, complete_H(v) \Rightarrow nREdge(v) = \emptyset \wedge \exists i \in \mathbb{N} : v = ext_H(i) \wedge ts(X)(i) \in \Lambda_0$

(2) $\forall e \in E_H, i \in [1, rk(e)] : type(att(e)(i)) \neq_0 ts(lab(e))(i) \Rightarrow ts(lab(e))(i) \in \Lambda_0 \wedge att(e)(i) = v_{null}$

(3) $\forall e \in E_H : lab_H(e) \notin Var_{\Sigma_N}$

(4) $v \in V_H, type(v) = \bot \Rightarrow v \in [ext_H]$

We denote the set of data structure grammars over Γ_N by DSG_{Γ_N}. ■

From (1), it follows that vertices either have all outgoing edges, possibly abstracted in one nonterminal, or they are external vertices and at the corresponding position in the type sequence there is a reduction type. It ensures that for every non-reduction tentacle, there are those outgoing edges, either abstracted or concrete, inside all rule graphs for that nonterminal. (2) is the same as for heap configurations. With (3), we forbid variables inside rules. Otherwise, it would be possible to create multiple variables of the same name in a heap configuration by the application of rules. We require that the null object is always an external vertex in (4) for the same reason.

Hyperedge replacement for DSGs works exactly as given by Definition 2.4.2 for HRGs.

As data structure grammars are restricted hyperedge replacement grammars, data structure grammar derivation and the language of those grammars remain the same.

Example 5 (DSG and Abstract Heap Configuration)
A doubly-linked list typically consists of a sequence of list elements connected by next and previous pointers. Figure 2.8 shows a DSG which describes a segment of such a doubly-linked list. Selectors are labeled with n or p, which stands for *next* and *previous* respectively, vertices with their type. e is the list element type. The type sequence of nonterminal L is given as $ts(L) = e_0\, e\, e\, e_0$. From the definition of DSGs, it follows that for any right-hand side of a rule $L \to H$, the first and last external node do not have any outgoing edges. Informally, the reason for that is that the *previous* pointer of the first and the *next* pointer of the last external node would point to a vertex not in this graph. The other two external nodes either have all their outgoing edges given by $pt(e) = np$, or they are both in turn abstracted by a nonterminal edge labeled with L. As the non-reduction tentacles $(L, 2)$ and $(L, 3)$ abstract all selectors of vertices with the type e, the second and third external node is required to have those selectors, either abstracted or not, which is obviously fulfilled in this example.

Figure 2.9 shows the process of abstracting a concrete segment of a doubly-linked list by the backwards application of the grammar rules given in Figure 2.8. Those parts which are replaced by a nonterminal in the next step are highlighted. In every step, we have a valid heap configuration. If a vertex does not have any of the outgoing edges of its type, then it is attached to a $(L, 2)$ or $(L, 3)$ tentacle which abstract those selectors. $(L, 1)$ and $(L, 4)$ tentacles are exclusively attached to vertices which have all of their selectors. Alternative, those vertices were allowed to be attached to a $(L, 2)$ or $(L, 3)$ tentacle.

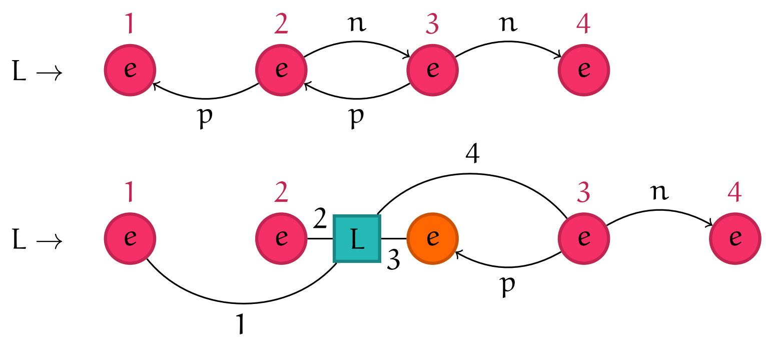

Figure 2.8: A Grammar for Doubly-Linked Lists.

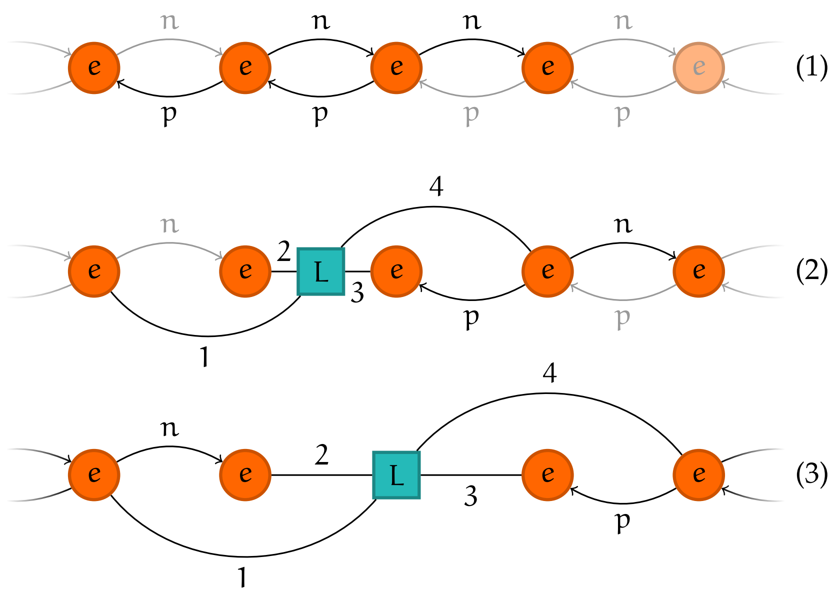

Figure 2.9: Abstraction of a Doubly-Linked Lists.

Example 6 (DSG for trees)
A binary tree typically consist of a root tree element which has two pointers on other tree elements. Every subsequent tree element either has pointers on additional tree elements or null pointers. Figure 2.10 shows a DSG which describes such a binary tree. Selectors are labeled with l or r, which stands for *left* and *right* respectively, vertices with their type. t is the tree element type, $\perp$ the type of the null vertex.

Example 7 (Data Structure Grammar Replacement)
Figure 2.11 shows an example for the application of the DSG of Figure 2.10, which we

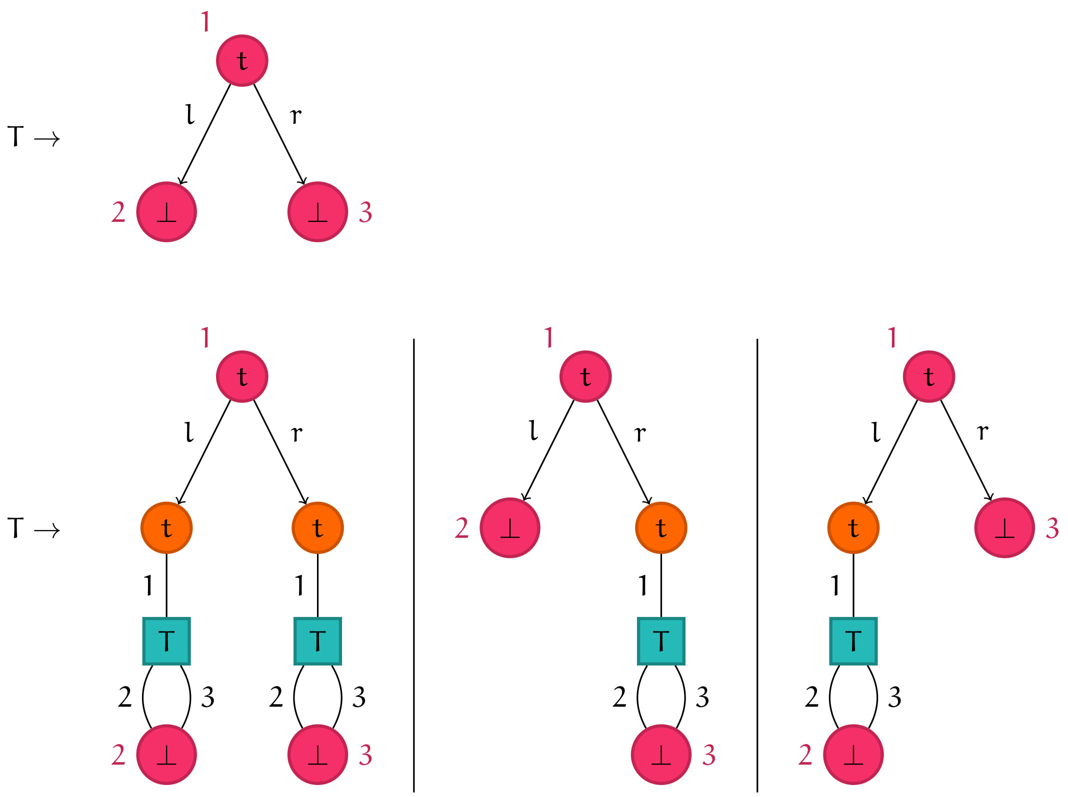

Figure 2.10: A Grammar for Binary Trees.

are going to call G, consisting of rules $T \to I$ and $T \to J \mid K \mid L$. In Figure 2.11, the small hypergraph on the left-hand side, called H, represents a heap of a program which consists of a single binary tree. The whole tree is abstracted by the nonterminal hyperedge e, labeled with T. In order to concretize the tree, we can apply one of the grammar rules given by G.

It is easy to see that $\mathrm{rk}(e) = 3$ is equal to the number of external vertices of every graph on a right hand side of a rule. Additionally, the types of every (T, i)-tentacle with $i \in [1, \mathrm{rk}(e) = 3]$ match with the types of the corresponding external vertices of every rule graph. The first external vertex is of type t, just like the node connected to the hyperedge e with the first tentacle. The other external vertices are of type $\perp$ and as required, the node connected to the second and third tentacle has the same type. Hence, a valid hyperedge replacement is possible. First, we choose to replace e_1 with J, which is shown right of H in Figure 2.11, which yields $H[J/e]$. This is done by merging external vertex $\mathrm{ext}_J(1)$ with the vertex connected to the first tentacle, $\mathrm{ext}_J(2)$ with the second and so forth, as indicated by the dashed arrows. The resulting graph $H' = H[J/e]$ is shown on the right-hand side of Figure 2.11.

We can now repeatedly apply rules of G in the resulting graph H' as long as there are nonterminals left. If we choose to replace the nonterminal on the right side with graph J, the other one with L, and all the newly introduced ones with I, there are no nonterminals left and we obtain a concrete representation of a binary tree as shown in Figure 2.12.

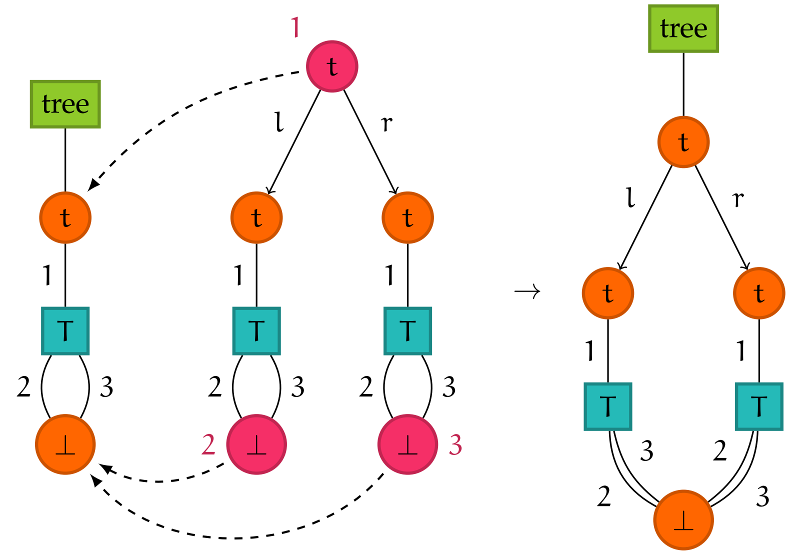

Figure 2.11: Example of Hyperedge Replacement.

2.6 Abstraction and Concretization

Formally, we define heap concretization as the process of applying grammar rules in an arbitrary manner, replacing nonterminals representing abstracted structures by more concrete representations.

Definition 2.6.1 (Heap Concretization)
Let $G \in DSG_{\Gamma_N}$, $I, J \in HC_{\Gamma_N}$. I is a *concretization* of J if $J \Rightarrow_G^* I$. ■

Consequently, a heap configuration is an abstraction of another heap configuration if the latter can be obtained by concretizing the former. That is equal to the backward application of the rules used for the concretization.

Definition 2.6.2 (Heap Abstraction)
Let $G \in DSG_{\Gamma_N}$, $I, J \in HC_{\Gamma_N}$. I is an *abstraction* of J if $I \Rightarrow_G^* J$. ■

Let $I, J \in HC_{\Gamma_N}$, $G \in DSG_{\Gamma_N}$. It holds that $(I \Rightarrow_G^* J) \Rightarrow L_G(I) \supseteq L_G(J)$. For every nonterminal in I there is a set of applicable grammar rules. The language of I is the union of all languages of I' with $I \Rightarrow_G I'$, obtained by applying one of those grammar rules. Thus, applying one specific grammar rule to I can only reduce the number of all derivable heap configurations. Therefore, this approach to heap abstraction yields an over-approximation of the state space.

An abstract heap configuration H represents a set of potential program states. This set are all heap configurations which can be derived from H, which is equal to the language of that heap configuration, $L_G(H)$. Applying abstraction or concretization to H leads to a different set of represented program states, which are reachable from the states represented by heap configurations in $L_G(H)$.

In order to by able to apply abstraction, backward DSG replacement needs to be defined as the inverse operation of DSG replacement. Prior to that, it is useful to define *embeddings* of DSG rules in heap configurations as something similar to subgraph isomorphism, which

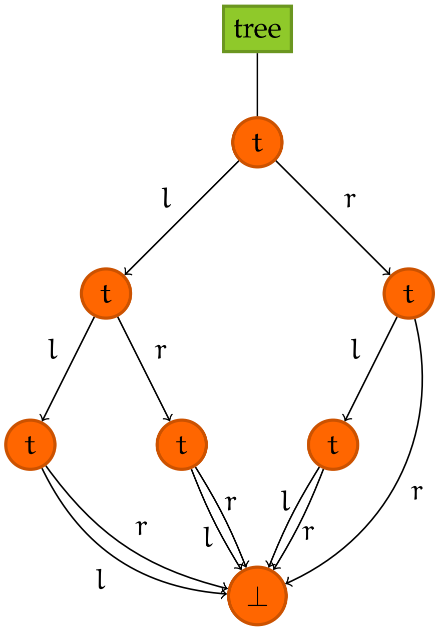

Figure 2.12: A Concrete Binary Tree.

is obtained by the replacement of a nonterminal edge with a rule graph. Embeddings for DSG rules in heap configurations are implicitly defined by the definitions of abstract heap configurations (2.5.3), data structure grammars (2.5.4) and data structure grammar replacement (2.4.2).

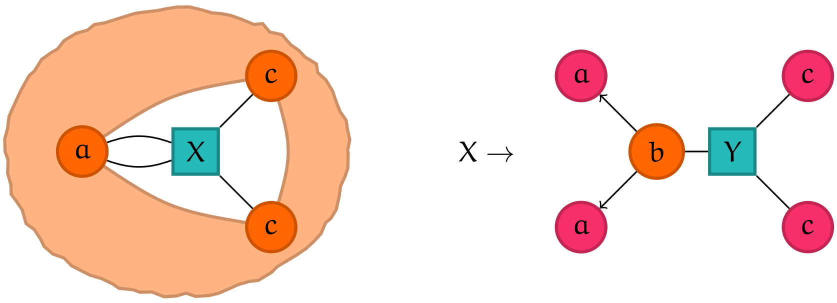

Figure 2.13: A Heap Configuration G and a Grammar Rule $X \to H$.

Consider Figure 2.13 for a simplified depiction of an arbitrary heap configuration with one nonterminal edge e labeled with X on the left side and a matching grammar rule on the right side. The heap configuration K resulting from the replacement $G[H/e]$ is shown in Figure 2.14. Obviously, there is an injective mapping of edges from H to K. Every edge of H is in K, too, and labels remain unchanged (not shown in this figure). It is not possible that there are any edges from G to internal vertices, as only external vertices are merged with vertices of the heap configuration.

The mapping of vertices is only injective for internal vertices of H. It is possible that several external vertices of H without outgoing edges are mapped to the same vertex in K, as it is possible that more than one non-reduction tentacle of one edge is attached to the same vertex. Types remain the same in K. The only excepting are external vertices of H with no

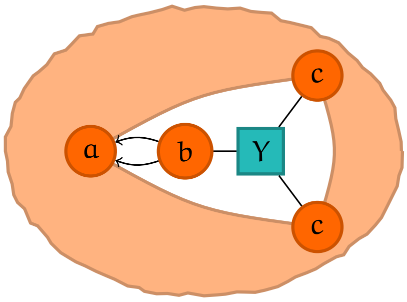

Figure 2.14: Heap Configuration $K = G[H/e]$.

outgoing edges. If the corresponding tentacle of e was attached to the null vertex, the type of that vertex would change to $\perp$.

Formally, this result in the following definition.

Definition 2.6.3 (Embedding of a DSG Rule in a HC)
Let $\{X \to H\} \in DSG_{\Gamma_N}$ and $G \in HC_{\Gamma_N}$. Without loss of generality, we assume that there are no vertices and edges of G and H which have the same name. If not, they have to be renamed. H is *embedded* in G if there are functions $f_V : V_H \to V_G$ and $f_E : E_H \to E_G$ such that

$$lab_H(e) = lab_G(f_E(e)) \forall e \in E_H \tag{2.7}$$

$$type_H(v) = type_G(f_V(v)) \forall v \in V_H \setminus [ext_H] \tag{2.8}$$

$$v = ext_H(i), type_H(v) \neq type_G(f_V(v)) \Rightarrow type_G(f_V(v)) = \perp, ts(X)(i) \in \Lambda_0 \tag{2.9}$$

$$att_G(f_E(e)) = f_V(att_H(e)) \forall e \in E_H \tag{2.10}$$

$$v_1, v_2 \in V_H, v_1 \neq v_2, v_1 \notin [ext_H] \Rightarrow f_V(v_1) \neq f_V(v_2) \tag{2.11}$$

$$ts(X)(i), ts(X)(j) \in \Lambda \Rightarrow f_V(ext_H(i)) \neq f_V(ext_H(j)), i, j \in [1, rk(X)], i \neq j \tag{2.12}$$

$$\forall e \in E_G \setminus f_E(E_H), i \in [1, rk(e)], v \in V_H \setminus [ext_H] : \neg\exists e' : f_E(e') = e \Rightarrow att_G(e)(i) \neq f_V(v) \tag{2.13}$$

The function f_E is injective, f_V is injective on all $v \in V_H \setminus [ext_H]$. An embedding of H in G with $f_V(ext_H(i)) = v \in V_G$ will be referred to as *an* i*-embedding of* $p : X \to H$ *in* G *at* v, or for short (p, i)-embedding. We denote the set of all embeddings as $Embr_{\Gamma_N}$. ■

It is now possible to define backward DSG replacement. Intuitively, a part of a heap configuration is removed and replaced by one nonterminal edge. Those vertices where the removed part was previously connected to the rest of the heap configuration stay in it. The new edge is then attached to all those vertices. Additionally to specifying the embedding that is supposed to be removed, one vertex of the heap configuration has to be given as there might be more then one embedding of that rule graph in the heap configuration.

Definition 2.6.4 (Backward DSG Replacement)
Let $J \in HC_{\Gamma_N}$, $\{X \to H\} \in DSG_{\Gamma_N}$, $e \in E_J$ with $rk(e) = rk(X)$ and $type(ext_H(i)) = type(att_J(e)(i)) \forall i \in [1, rk(e)]$. Let there be an $(X \to H, i)$-embedding at $v \in V_J$ with $f_V : V_H \to V_J$ and $f_E : E_H \to E_J$. Without loss of generality, we assume that there are no vertices and edges of J and H which have the same name. If there are any, they have to be renamed. The *backwards substitution* of the $(X \to H, i)$-embedding at v, $J[e/(X \to H, i, v)] = K \in HC_{\Gamma_N}$,

is defined by:

$$
\begin{aligned}
V_K &= V_J \setminus f_V(V_H \setminus [ext_H]) \\
E_K &= (E_J \setminus f_E(E_H)) \cup \{e\} \\
lab_K &= (lab_J \restriction (E_J \setminus f_E(E_H))) \cup \{e \mapsto X\} \\
att_K &= att_J \restriction (E_J \setminus f_E(E_H)) \cup \{e \mapsto f_E(ext_H)\} \\
ext_K &= ext_J \\
type_K &= type_J \restriction (V_J \setminus f_V(V_H \setminus [ext_H]))
\end{aligned}
$$

■

As a first step to proving the correctness of backward data structure grammar replacement, we show that there is an embedding of a rule $X \to H$ after replacing an edge with that rule.

Lemma 2.6.1
Let $I \in HC_{\Gamma_N}$, $\{X \to H\} \in DSG_{\Gamma_N}$, $e \in E_I$ with $rk(e) = rk(X)$ and $I[H/e] = J$. There is an $(X \to H, i)$-embedding in J at $att_J(e)(i)$ for all $i \in [1, rk(e)]$ with $ts(X)(i) \in \Lambda$.

Proof: If there is an embedding of H in J, there have to be functions $f_V : V_H \to V_J$ and $f_E : E_H \to E_J$ such that the requirements of Definition 2.6.3 are fulfilled. From (2.1) and (2.4), it follows that $f_V = id_{V_H \setminus [ext_H]} \cup \{ext_H(i) \mapsto att_I(e)(i) \mid i \in [1, rk(e)]\} = mod$. Following from (2.2), we get $f_E = id_{E_H}$. In combination with the functions obtained by the replacement of e with H, it is possible to derive the requirements of an embedding of H in J.

$$
\begin{aligned}
lab_J(f_E(e')) = lab_J(e') = lab_H(e') \forall e' \in E_H \\
type_J(f_V(v)) = type_J(v) = type_H(v) \forall v \in V_H \setminus [ext_H] \\
v = ext_H, type_H(v) \neq type_G(f_V(v)) \Rightarrow type_G(f_V(v)) = \bot, ts(X)(i) \in \Lambda_0
\end{aligned}
$$

The requirements for types follow from the corresponding conditions in the definitions of heap configurations (2.5.3) and data structure grammars (2.5.4).

$$att_J(f_E(e')) = att_J(e') = mod \circ att_H(e') = f_V(att_H(e')) \forall e' \in E_H$$

Let $v_1, v_2 \in V_H, v_1 \neq v_2, v_1 \notin [ext_H]$. Then, $f_V(v_1) = id_{V_H \setminus [ext_H]}(v_1) = v_1$. If $v_2 \notin [ext_H]$, too, it is $f_V(v_2) = id_{V_H \setminus [ext_H]}(v_2) = v_2$ and obviously $f_V(v_1) \neq f_V(v_2)$. In case of $v_2 = ext_H(k)$, $f_V(v_2) = att_I(e)(k) \neq v_1$, as v_1 is element of V_H. Thus, requirement (2.11) is satisfied.
With the requirement for DSGs $ts(X)(i), ts(X)(j) \in \Lambda \Rightarrow att_I(e)(i) \neq att_I(e)(j) \forall i, j \in [1, rk(X)], i \neq j$ and $att_I(e)(k) = f_V(ext_H(k))$, (2.12) directly follows.
That the last condition is satisfied can be seen as follows. Consider $\forall e' \in E_J \setminus f_E(E_H), k \in [1, rk(e')], v \in V_H \setminus [ext_H] : \neg\exists e'' : f_E(e'') = e' \Rightarrow att_J(e')(k) \neq f_V(v)$. Since f_E is only defined on E_H, $e'' = e' \in E_H$ has to hold and $f_V(v) = v$ because of $v \in V_H \setminus [ext_H]$. Then, it is $att_J(e')(k) = att_I(e')(k) \in V_I$. Now, it is not possible that $att_J(e')(k) = f_V(v)$ since $V_I \cap V_H = \emptyset$.
Since all conditions of Definition 2.6.3 are satisfied, there is an $(X \to H, i)$-embedding in J at $att_J(e)(i)$ for all $i \in [1, rk(e)]$ with $ts(X)(i) \in \Lambda$. ■

The correctness of data structure grammar replacement now can be shown by proving that first applying forward and then backward replacement of the same rule yields a heap configuration isomorphic to the initial heap configuration.

Theorem 2.6.1 (Correctness of DSG Replacement)
Let $I \in HC_{\Gamma_N}$, $\{X \to H\} \in DSG_{\Gamma_N}$, $e \in E_I$ with $rk(e) = rk(X)$ and $type(ext_H(i)) = type(att_I(e)(i)) \forall i \in [1, rk(e)]$. Let $I[H/e] = J$. Let there be an $(X \to H, i)$-embedding at $v \in V_J$ with $f_V : V_H \to V_J$ and $f_E : E_H \to E_J$ and for $K = I[H/e][e'/(X \to H, i, v)]$ with $v = att_I(e)(i)$, I is isomorphic to K.

Proof: For isomorphism of K and I, there have to be functions $g_V : V_K \to V_I$ and $g_E : E_K \to E_I$ fulfilling the requirements of Definiton 2.6.3. When replacing an edge in I with a grammar rule and then replacing the resulting embedding with an edge, the only difference between K and I is that the initially replaced edge e is not in K anymore. There is, however, a new edge e' at the same place, that is isomorphic to e. Thus, we get $g_V = id_{V_I}$ and $g_E = id_{E_I \setminus \{e\}} \cup \{e' \mapsto e\}$.
From $I[H/e] = J$ and Lemma 2.6.1, it follows that there is an $(X \to H, i)$-embedding at $v \in V_J$ with $f_V = id_{V_H \setminus [ext_H]} \cup \{ext_H(i) \mapsto att_I(e)(i) \mid i \in [1, rk(e)]\} = mod$ and $f_E = id_{E_H}$.
Now, using the definitions of forward and backward replacement, we can determine K.

$$\begin{aligned} V_K &= V_J \setminus f_V(V_H \setminus [ext_H]) \\ &= (V_I \cup (V_H \setminus [ext_H])) \setminus f_V(V_H \setminus [ext_H]) \\ &= V_I \end{aligned}$$

$$\begin{aligned} E_K &= (E_J \setminus f_E(E_H)) \cup \{e'\} \\ &= (((E_I \setminus \{e\}) \cup E_H) \setminus f_E(E_H)) \cup \{e'\} \\ &= (E_I \setminus \{e\}) \cup \{e'\} \end{aligned}$$

$$\begin{aligned} lab_K &= (lab_J \restriction (E_J \setminus f_E(E_H))) \cup \{e' \mapsto X\} \\ &= (((lab_I \restriction (E_I \setminus \{e\})) \cup lab_H) \restriction (E_J \setminus f_E(E_H))) \cup \{e' \mapsto X\} \\ &= (lab_I \restriction (E_I \setminus \{e\})) \cup \{e' \mapsto X\} \end{aligned}$$

$$\begin{aligned} att_K &= att_J \restriction (E_J \setminus f_E(E_H)) \cup \{e' \mapsto f_E(ext_H)\} \\ &= (att_I \restriction (E_I \setminus \{e\}) \cup mod \circ att_H) \restriction (E_J \setminus f_E(E_H)) \cup \{e' \mapsto f_E(ext_H)\} \\ &= (att_I \restriction (E_I \setminus \{e\})) \cup \{e' \mapsto ext_H\} \end{aligned}$$

$$\begin{aligned} ext_K &= ext_J \\ &= ext_I \end{aligned}$$

$$\begin{aligned} type_K &= type_J \restriction (V_J \setminus f_V(V_H \setminus [ext_H])) \\ &= (type_I \cup (type_H \restriction (V_H \setminus [ext_H]))) \restriction (V_I \cup (V_H \setminus [ext_H]) \setminus f_V(V_H \setminus [ext_H])) \\ &= (type_I \cup (type_H \restriction (V_H \setminus [ext_H]))) \restriction V_I \\ &= type_I \end{aligned}$$

It is now easy to show that K and I are isomorphic, using g_V and g_E.

$$\begin{aligned}
\mathrm{lab}_K(e'') &= \mathrm{lab}_I(e'') = \mathrm{lab}_I(g_E(e'')), \forall e'' \in E_K \setminus \{e'\} \\
\mathrm{lab}_K(e') &= \mathrm{lab}_I(g_E(e')) = \mathrm{lab}_I(e) = X \\
\mathrm{type}_K(v) &= \mathrm{type}_I(v) = \mathrm{type}_I(g_V(v)), \forall v \in V_K \\
\mathrm{att}_I(g_E(e'')) &= \mathrm{att}_I(e'') = g_V(\mathrm{att}_K(e'')), \forall e'' \in E_K \setminus \{e'\} \\
\mathrm{att}_I(g_E(e')) &= \mathrm{att}_I(e) = \mathrm{ext}_H = g_V(\mathrm{att}_K(e)) \\
\mathrm{ext}_I &= \mathrm{ext}_K = g_V(\mathrm{ext}_K)
\end{aligned}$$

Hereby, we have shown that DSG replacement is correct, as replacing an edge with a rule graph and replacing the resulting embedding with an edge yields a heap configuration equal under renaming to the initial heap configuration. ■

3 Automata-Based Embedding Detection

A Data Structure Grammar for more than one data structure, for example one for singly- and doubly-linked list and trees, consists of dozens of grammar rules. The rules for one certain data structure, however, are usually to some extent similar. Therefore, it is desirable to have an embedding detection mechanism that takes advantage of this fact and does not verify the same properties more often than necessary. That led to the idea of applying automata comparable to finite state machines, such that it is possible to construct an automaton for every rule and then compute the union, resulting in an automaton that detects multiple embeddings in one run, without verifying the properties of one vertex twice.

3.1 Paths on Heap Configurations

On a real heap, every object that is not garbage can be reached from the program, either directly via one pointer, or a number of pointers to other objects with pointers. We will define *paths on heap configurations* in a similar way, as a concatenation of tentacles which lead from one object to another just like pointers on a real heap. It is necessary, however, to further extend this notion to be able to deal with nonterminal edges. While it is still possible to uniquely define path, as tentacles can be distinguished by their ordinals, it is not possible to take into account the potential paths on those structures abstracted by nonterminals, as they do not preserve this information.

It turned out to be useful to evaluate properties of vertices in a path-based manner, starting from one fixed initial vertex. That allows to unambiguously refer to vertices of the graph by a path leading to it.

For hyperedges of rank two labeled with a selector, it is natural to say that a path leads from the first attached vertex to the second one, following the direction of the pointer. In case of nonterminal hyperedges, it needs to be taken into account what structures the hyperedge could possibly represent. To ensure that paths are unique, they are only allowed to leave vertices via non-reduction tentacles. Recall that there is either exactly one nonterminal non-reduction tentacle, abstracting all concrete tentacles, or there is a set of terminal tentacles as defined by the type of that vertex. Furthermore, all outgoing terminal tentacles of one vertex are uniquely labeled. We have to leave edges, however, at every tentacle, except for the one where we entered. It is not sufficient to leave edges only at reduction tentacles, as it is possible to have nonterminals with no reduction tentacles.

Formally, we define paths as a sequence of tentacles. While there are always two tentacles between two vertices directly connected by an edge, it suffices to take only that one where the path leaves the hyperedge. Assume we have a tentacle (X, i) as a path element. As every vertex is required to be complete, there is only one edge in the set of non-reduction tentacles labeled with X. The i then uniquely specifies one tentacle where that edge is left.

Definition 3.1.1 (Paths on Heap Configurations)
Let Σ_N be a heap alphabet. An element $\pi \in \Pi^*_{\Sigma_N}$ with $\Pi_{\Sigma_N} = T_{\Sigma_N}$. We say that π is

path from v_x to v_y in $G \in HC_{\Sigma_N}$, with $v_x, v_y \in V_G$, if there is a sequence of vertices $s_\pi = v_0 \dots v_n \in V_G^*$, with $n = |\pi|$ and $v_x = v_0, v_y = v_n$, such that $\forall i \in [1, |\pi|], (s,k) = \pi(i) : \exists e \in \mathrm{nREdge}(v_{i-1}), \mathrm{lab}(e) = s, \mathrm{att}_G(e)(k) = v_i$ and $v_i \neq v_{i-1}$. For simplicity, instead of a selector tentacle $(s, 2)$ we just write s. ■

Theorem 3.1.1 (Uniqueness of Paths)
Let $G \in HC_{\Sigma_N}$. For all $v, v_1, v_2 \in V_G$ and $\pi \in \Pi_{\Sigma_N}^*$, if π is a path from v to v_1 and a path from v to v_2, then $v_1 = v_2$.

Proof: Assume that π is a path from v to v_1 and a path from v to v_2 with $v_1 \neq v_2$. Thus, by the definition of paths (3.1.1), there have to be two sequences $a_1, a_2 \in V_G^*$ with $a_1(1) = a_2(1) = v$, $a_1(|\pi| + 1) = v_1$ and $a_2(|\pi| + 1) = v_2$. Following from that, it has to hold that for some $i \in [0, |\pi| - 1]$, we have $a_1(i) = a_2(i) = v_{i-1}$ and $a_1(i + 1) \neq a_2(i + 1)$. That is only possible if there are two edges $e_1, e_2 \in E_G$, with $e_1, e_2 \in \mathrm{nREdge}(v_{i-1})$, $(s, k) = \pi(i)$, $\mathrm{lab}(e_1) = \mathrm{lab}(e_2) = s$ and $\mathrm{att}_G(e_1)(k) = a_1(i + 1)$ and $\mathrm{att}_G(e_2)(k) = a_2(i + 1)$. That is a contradiction to the Definition 2.5.3 of heap configurations, which restricts a vertex v to have only one edge in $\mathrm{nREdge}(v)$ with a certain label. Hence, it is not possible that one path specifies two distinct vertices when evaluated with respect to the same initial vertex. ■

In order to evaluate a path, it has to be known on which heap configuration this path should be evaluated and at which vertex it starts. To ensure that the vertex is in the graph, we define a set of tuples of graphs and one of its vertices.

Definition 3.1.2 (Graph-Verex Tuples)
The *set of graph-vertex tuples* is defined as the set $I_{\Gamma_N} = \{(G, v) \mid G \in HC_{\Gamma_N}, v \in V_G\}$. ■

Evaluating a path on a heap configuration intuitively means following that path, from the initial vertex that was given to its endpoint.

Definition 3.1.3 (Path Evaluation)
Let $(G, v) \in I_{\Gamma_N}$. The *path evaluation function* $\mathrm{eval} : \Pi_{\Sigma_N}^* \times I_{\Gamma_N} \to V_G$ returns the vertex which is reached when the path $\pi \in \Pi_{\Sigma_N}^*$ is followed, starting at vertex v in graph G. Instead of $\mathrm{eval}(\pi, (G, v))$, we write for short $\langle \pi \rangle_{(G,v)}$. Furthermore, if (G, v) is clear from the context, we may omit it. ■

We denote the empty path with ε, and define that $\langle \varepsilon \rangle_{(G,v)} = v$.

Definition 3.1.4 (Reachability on Heap Configurations)
Let $G \in HC_{\Sigma_N}$. A vertex v' is *reachable* from v if there is a $\pi \in \Pi_{\Sigma_N}^*$ with $\langle \pi \rangle_{(G,v)} = v'$. ■

It is useful to define an ordering on path, based on the order of which labels appear in the sequence of vertex type selectors and the ordinal number of tentacles.

Definition 3.1.5 (Path Order)
Let, $t \in \Lambda$, $(s_1, 2), (s_2, 2) \in \Pi_{\Sigma_N}$, $s_1 = \mathrm{pt}(t)(i), s_2 = \mathrm{pt}(t)(j)$ and $s_1 \neq s_2$. We define that $(s_1, 2) \prec (s_2, 2)$ if and only if $i < j$. For all $(X, i), (X, j) \in \Pi_{\Sigma_N}$, we define $(X, i) \prec (X, j)$ if $i < j$. Those pairs of path elements not mentioned explicitly are ordered alphabetical. Tentacles of equal type are ordered by their ordinal number.
For two paths $\pi_1, \pi_2 \in \Pi_{\Sigma_N}^*$, we say that $\pi_1 \prec \pi_2$ if there is a k with $\pi_1(n) = \pi_2(n) \forall n : 0 \leqslant n < k$ and $\pi_1(k) \prec \pi_2(k), k \leqslant |\pi_1|, k \leqslant |\pi_2|$ or if $\pi_1(n) = \pi_2(n) \forall n : 1 \leqslant n \leqslant |\pi_1|$ and $|\pi_1(n)| < |\pi_2(n)|$. ■

Finally, it is possible to define the set of depth-first search paths on a heap configuration as the set of all paths which are evaluated during a depth-first search starting at one vertex.

Definition 3.1.6 (Set Depth-First Search Paths)
Let $G \in HC_{\Gamma_N}, \nu \in V_G$. The *set of depth-first search paths* on G starting at ν is defined as the set of all paths visiting no vertex, except for possibly the last one, twice. The sequence of all vertices a paths π visits is denoted by s_π. $\Pi_{DFS} = \{\pi \in \Pi^*_{\Sigma_N} \mid s_\pi = \nu_0 \ldots \nu_n, \forall i, j \in [1, n-1] : \nu_i \neq \nu_j\}$. It can be constructed as follows by the execution of the function DFSPath(G, ν) with the initialization $\pi := \varepsilon$ and $\Pi_{DFS} := \{\varepsilon\}$.

Function DFSPath($G \in HC_{\Gamma_N}, \nu \in V_G$)

```
mark ν as visited
forall π′ ∈ {(lab(e), i) | e ∈ nREdge(ν), i ∈ [1, rk(e)], att_G(e)(i) ≠ ν} do
    π := ππ′
    ν′ := ⟨π⟩_(G,ν)
    Π_DFS := Π_DFS ∪ {π}
    if ν′ not marked as visited then
        DFSPath(G, ν′)
    π := π(1) ... π(|π| − 1)
```

■

3.2 Path-Based Embedding

In order to be able to detect embeddings with automata working on paths on a heap configuration, we provide a set of purely path-based requirements for an embedding. The advantage of using paths is that they allow us to uniquely identify every vertex that is reachable from one initial vertex. Additionally to that, it is not necessary to check any properties of edges, since paths implicitly contain those information. As paths go through edges, they can be used to find out which vertices an edge is attached to. Furthermore, paths are sequences of tentacles, and tentacles contain labels of edges, such that labels are checked, too. This approach, however, restricts which embeddings can be detected. To guarantee that the properties of all vertices are checked, they all have to be reachable from one external vertex. It suffices to choose an external vertex since grammar rules do not contain garbage, that is, vertices which are not reachable from any external vertex. If there is an internal vertex from which every other vertex can be reached, then there is an external vertex, too. Since we have introduced an ordering on path, there is a natural order for checking the properties of all vertices, which makes it easier to construct automata. Prior to presenting those paths-based requirements, some new functions need to be defined.

The degree of a vertex is the number of all edges it is attached to. Outgoing tentacles are all non-reduction tentacles attached to a vertex. They either stand for concrete outgoing selectors or a nonterminal tentacle which abstracts all outgoing selectors.

Definition 3.2.1 (Vertex Degree)
Let $G \in HC_{\Gamma_N}$. The *degree* of a vertex is defined as the function: $\text{degree}(\nu) = |\{e \in E_G \mid \nu \in [att_G(e)]\}|$. For short, we write $\deg(\nu)$. ■

Definition 3.2.2 (Attached Tentacles)
Let $G \in HC_{\Sigma_N}$. The set of *attached tentacles* of $v \in V_G$ is defined as: $\mathrm{tentacle}(v) = \{(\mathrm{lab}(e), n) \mid e \in E_G, n \in [1, \mathrm{rk}(e)], \mathrm{att}_G(e)(n) = v\}$. ■

Definition 3.2.3 (Outgoing Tentacles)
Let $G \in HC_{\Sigma_N}$. The set of *outgoing tentacles* of $v \in V_G$ is defined as: $\mathrm{outtentacle}(v) = \{(x, n) \in \mathrm{tentacle}(v) \mid \mathrm{ts}(\mathrm{lab}(x))(n) \in \Lambda\}$. For short, we write $\mathrm{ot}(v)$. ■

Note that difference between outtentacle and nREdge is that the latter returns edges, while the former returns the tentacles of those edges which are attached to the vertex that is given as an argument.

Theorem 3.2.1 (Path-Based Embedding)
Let $\{X \to H\} \in DSG_{\Gamma_N}$ and $G \in HC_{\Gamma_N}$. Let $v_H = \mathrm{ext}_H(i), i \in [1, \mathrm{rk}(X)]$ be a vertex from which all other vertices in V_H are reachable. $\Pi_{DFS} = \mathrm{DFSPath}(H, v_H)$. There is a $(X \to H, i)$-embedding in G at $v_G \in V_G$ if and only if there is a pair (v_H, v_G) such that for all $\pi \in \Pi_{DFS}$ with $\langle\pi\rangle_{(H,v_H)} \in V_H \setminus [\mathrm{ext}_H]$

$$\mathrm{type}_H(\langle\pi\rangle_{(H,v_H)}) = \mathrm{type}_G(\langle\pi\rangle_{(G,v_G)}) \tag{3.1}$$
$$\mathrm{ot}(\langle\pi\rangle_{(H,v_H)}) = \mathrm{ot}(\langle\pi\rangle_{(G,v_G)}) \tag{3.2}$$
$$\deg(\langle\pi\rangle_{(H,v_H)}) = \deg(\langle\pi\rangle_{(G,v_G)}) \tag{3.3}$$

The properties of internal vertices are preserved in embeddings. The degree has to remain unchanged as it is not allowed that there are any edges from outside of the embedding attached to internal vertices.
For all $\pi \in \Pi_{DFS}$ with $\langle\pi\rangle_{(H,v_H)} \in [\mathrm{ext}_H]$, the following has to hold:

$$\mathrm{ot}(\langle\pi\rangle_{(H,v_H)}) = \mathrm{ot}(\langle\pi\rangle_{(G,v_G)}) \text{ if } \mathrm{ot}(\langle\pi\rangle_{(H,v_H)}) \neq \emptyset \tag{3.4}$$
$$\deg(\langle\pi\rangle_{(H,v_H)}) \leqslant \deg(\langle\pi\rangle_{(G,v_G)}) \tag{3.5}$$

If an external vertex has all outgoing tentacles in H, it has to have the same set of tentacles in G. If it has no outgoing tentacles in H, the set of outgoing tentacles just has to match the type of the vertex, which is by definition always the case for heap configurations. The degree might be larger in G, as it is possible that there are other edges of G attached to that vertex. Furthermore, if $\langle\pi\rangle_{(H,v_H)} = \mathrm{ext}_H(i)$, then

$$\mathrm{type}_H(\langle\pi\rangle_{(H,v_H)}) \neq \mathrm{type}_G(\langle\pi\rangle_{(G,v_G)}) \Rightarrow \mathrm{type}_G(\langle\pi\rangle_{(G,v_G)}) = \perp, \mathrm{ts}(X)(i) \in \Lambda_0 \tag{3.6}$$

That ensures that if the type of a vertex in H is not equal to the type of that vertex in G, the vertex in G has to be null and the corresponding tentacle is a reduction tentacle, which is always allowed to be attached to null.
For two external non-reduction vertices with $i, j \in [1, \mathrm{rk}(X)]$, $i \neq j$, $\langle\pi_1\rangle_{(H,v_H)} = \mathrm{ext}_H(i)$ and $\langle\pi_2\rangle_{(H,v_H)} = \mathrm{ext}_H(j)$, we require

$$\mathrm{ts}(X)(i), \mathrm{ts}(X)(j) \in \Lambda \Rightarrow \langle\pi_1\rangle_{(G,v_G)} \neq \langle\pi_2\rangle_{(G,v_G)} \tag{3.7}$$

That has to be the case as it is not allowed that two non-reduction tentacles are attached to the same vertex. Since those tentacles abstract outgoing selectors, the attached vertex then had all selectors twice. The final requirement states if there are two path to the same vertex in H, they have to lead to the same vertex in G, too.

$$\langle\pi_1\rangle_{(H,\nu_H)} = \langle\pi_2\rangle_{(H,\nu_H)} \Rightarrow \langle\pi_1\rangle_{(G,\nu_G))} = \langle\pi_2\rangle_{(G,\nu_G))} \forall \pi_1,\pi_2 \in \Pi_{DFS} \tag{3.8}$$

Proof: Let $\pi = (a_1, b_1)\dots(a_n, b_n) \in \Pi^*_{\Gamma_N}$, with $n = |\pi|$, $\pi_{-1} = (a_1, b_1)\dots(a_{n-1}, b_{n-1}) \in \Pi^*_{\Gamma_N}$ and $\pi(n) = (X, k)$. For all $\nu'_H \in V_H$, let $\Pi^{\nu'_H}_{DFS}$ be the set of all paths $\pi \in \Pi_{DFS}$ with $\langle\pi\rangle_{(H,\nu_H)} = \nu'_H$.
First, we show that there is a $(X \to H, i)$-embedding in G at $\nu_G \in V_G$ if there is a pair of vertices (ν_H, ν_G) as described above. We need a function $f_V : V_H \to V_G$, mapping the vertices of the rule graph to the vertices of the heap configuration. From 3.8, it follows that

$$\pi_1, \pi_2 \in \Pi^{\nu'_H}_{DFS} \Leftrightarrow \langle\pi_1\rangle_{(G,\nu_G)} = \langle\pi_2\rangle_{(G,\nu_G)} = \nu'_G$$

Hence, for all $\nu'_H \in V_H$ there is a unique mapping $\nu'_H \mapsto \nu'_G \in V_G$ defined through the paths, that can be expressed as a function $f_V : V_H \to V_G$ with $f_V(\langle\pi\rangle_{(H,\nu_H)}) = \langle\pi\rangle_{(G,\nu_G)}$.
Furthermore, a function $f_E : E_H \to E_G$ is required. Recall that grammar rules do not contain any garbage. Thus, every edge has at least one non-reduction tentacle. It holds that for all $e_H \in E_H$, there is an $i \in [1, rk(e)]$ such that $ts(lab(e))(i) \in \Lambda$. It follows that for all $e_H \in E_H$, there is a $\nu'_H \in V_H$ with $e_H \in nREdge(\nu'_H)$. Then, it is possible to define $f_E(e_H) := e_G \Leftrightarrow \exists \nu'_H \in V_H, e_H \in nREdge(\nu'_H), e_G \in nREdge(f_V(\nu'_H)), lab(e_H) = lab(e_G)$. Note that there is such a ν'_H for every non-reduction tentacle of e_H. It is guaranteed that this works for every edge in H because (3.2) and (3.4) make sure that the set of outgoing tentacles inside H is the same for ν'_H and $f_V(\nu'_H)$.

(2.10) For all $\pi \in \Pi_{DFS}$ with $ot(\langle\pi\rangle_{(H,\nu_H)}) \neq \emptyset$, the paths π_{+1} leading to the successors of $\langle\pi\rangle_{(H,\nu_H)} = \nu'_H$ are in Π_{DFS}, too. Let $\langle\pi\rangle_{(H,\nu_H)} = \nu'_H$ and $\langle\pi_{+1}\rangle_{(H,\nu_H)} = \nu''_H$. π_{+1} specifies an edge $e_H \in E_H$ such that there are $i, j \in [1, rk(e_H)]$ with $att_H(e_H)(i) = \nu'_H$ and $att_H(e_H)(j) = \nu''_H$. Evaluating those paths on G gives us an $\nu'_G = f_V(\nu'_H)$, $\nu''_G = f_V(\nu''_H)$ and $e_G = f_E(e_H)$. It then holds that $att_G(e_G)(i) = \nu'_G = f_V(\nu'_H)$ and $att_G(e_G)(j) = \nu''_G = f_V(\nu''_H)$. Since a depth-first search visits all paths and therefore all tentacles of an edge and all attached vertices in H, it follows that $att_G(f_E(e)) = f_V(att_H(e)) \forall e \in E_H$.

(2.7) That follows as a path π specifies not only edges, but also their labels, as can be seen with the definition of paths, which states that there has to be an $e_G \in nREdge(\langle\pi_{-1}\rangle_{(H,\nu_H)})$ with $lab(e_H) = X$.

(2.11) Because of $\nu_1 \notin [ext_H]$ and (3.3), $deg(\nu_1) = deg(f_V(\nu_1))$ has to hold. Assume now that $f_V(\nu_1) = f_V(\nu_2)$. That is possible as long as ν_1 and ν_2 have the same type (3.1), the same set of outgoing tentacles and the same degree. It follows that a difference in the degree of ν_1 and $f_V(\nu_1)$ is only possible due to incoming tentacles. If all paths leading to ν_1 and ν_2 are different in their last path element, $deg(f_V(\nu_1))$ would be larger than $deg(\nu_1)$ because there had to be distinct incoming tentacles for those paths. If two paths π_1 and π_2 with $\langle\pi_1\rangle_{(H,\nu_H)} = \nu_1$ and $\langle\pi_2\rangle_{(H,\nu_H)} = \nu_2$ are the same in the last path element, and $\langle\pi_{1,-1}\rangle_{(H,\nu_H)} = \langle\pi_{2,-1}\rangle_{(H,\nu_H)}$, because of the uniqueness of path it had to hold that $\nu_1 = \nu_2$, which is a contradiction to the initial assumption. If $\langle\pi_{1,-1}\rangle_{(H,\nu_H)} \neq \langle\pi_{2,-1}\rangle_{(H,\nu_H)}$ and $\langle\pi_{1,-1}\rangle_{(G,\nu_G)} = \langle\pi_{2,-1}\rangle_{(G,\nu_G)}$, there were no additional paths from $\langle\pi_{1,-1}\rangle_{(G,\nu_G)}$ to $\langle\pi_1\rangle_{(G,\nu_G)}$. That is not possible, however, because

$\langle \pi_{1,-1} \rangle_{(H,\nu_H)}$ and $\langle \pi_{2,-1} \rangle_{(H,\nu_H)}$ have to be external nodes, which cannot be merged in G because both have outgoing tentacles, as stated by (3.7).

(2.13) As internal vertices have to keep the same degree when they are mapped from H to G (3.3), as well as the same incoming (3.8) and outgoing tentacles (3.2), there can't be any other edges connected to them which are not in H.

Finally, (2.8), (2.9) and (2.12) follow from (3.1), (3.6) and (3.7) by replacing $\langle \pi \rangle_{(G,\nu_G)}$ with $f_V(\langle \pi \rangle_{(H,\nu_H)})$.

We now show the opposite direction. There is an $(X \to H, i)$-embedding at vertex $\nu = f_V(ext_H(i))$. For $f_V : V_H \to V_G$ from Definition 2.6.3, it holds that $f_V(\langle \pi \rangle_{(H,\nu_H)}) = \langle \pi \rangle_{(G,\nu_G)}$ for all $\pi \in \Pi_{DFS}$, which follows from (2.10) and (2.7). As edges as well as labels are directly mapped from H to G, all paths on H exist in G, too.
To show that (3.1) and (3.6) hold, it suffices to replace ν with $\langle \pi \rangle_{(G,\nu_G)}$ in (2.8) and (2.9) and V_H with Π_{DFS} in (2.8). Following from the construction of Π_{DFS}, still all vertices of V_H are checked.
(3.2) and (3.4) follow from (2.10) and (2.7). As for all vertices $\nu'_H \in V_H$ which are attached to an edge $e_H \in E_H$ are, the corresponding vertices $f_V(\nu'_H)$ are attached to the corresponding edges $f_E(e_H)$ and the labels of those edges stay the same, $f_V(\nu'_H)$ has the same outgoing tentacles as ν'_H if they belonged to H.

(3.3) Due to (2.13), if there is an edge $e_G \in E_G$ without any $e_H \in E_H$ such that $f_E(e_H) = e_G$, it is not allowed that this edge is attached to any vertex $f_V(\nu'_H)$ with ν'_H being an internal vertex $\nu'_H \in V_H \setminus [ext_H]$. Together with (2.10), that means that $f_V(\nu'_H)$ is attached to exactly the same set of tentacles as ν'_H. Consequently, the degrees are equal.

(3.5) For a vertex $f_V(\nu'_H)$ with $\nu'_H \in [ext_H]$, it is possible that there are, additionally to those tentacles which are also in H as required by (2.10), incoming tentacles which are not in H, resulting in a degree of vertex $f_V(\nu'_H)$ that is at least as large as the degree of ν'_H.

(3.7) That obviously follows from (2.12) using $f_V(\langle \pi \rangle_{(H,\nu_H)}) = \langle \pi \rangle_{(G,\nu_G)}$.

(3.8) It can easily be seen that this holds when considering $f_V(\langle \pi \rangle_{(H,\nu_H)}) = \langle \pi \rangle_{(G,\nu_G)}$ together with (2.7) and (2.10).

■

3.3 Function Automata

We now have a set of requirements for an embedding of a data structure grammar rule, given in Theorem 3.2.1, that is solely based on properties of vertices. The vertices are uniquely defined by paths leading to them. Our goal is now to develop an automaton that decides if there is an embedding at a certain vertex in a heap configuration based on those requirements. Each state of that automaton is supposed to verify one property of one vertex specified by a path. Thus, a path needs to be evaluated in every state. Recall that path evaluation is a function on a heap configuration and one of its vertices.

In this section, we will define an automata model, the so called *function automata*, that is suited for this purpose. It is to some extent similar to a finite state machine. There is, however, an additional function λ, that assigns to each state a function from a universe of functions $\mathfrak{F}$. Instead of labeling transitions with elements of an input alphabet, in each state, there is one transition for each element of the codomain of the corresponding λ-function. That requires that all function of $\mathfrak{F}$ have a finite codomain. Instead of an input word, the input of a function automata can be every object that might be an argument of the λ-functions. This leads us to a second requirement for λ-functions: All functions of $\mathfrak{F}$ must have the same domain.

When we use function automata to detect embeddings, the domain of all functions of $\mathfrak{F}$ will be the set of all graph-vertex tuples.

As there is no linear input word that is read from the beginning to the end, the order of states and their functions in sections of the automaton without any branching has no influence on the language, as long as the functions do not modify the input. Therefore, it is useful to require that there is an ordering relation $\prec$ on all functions of $\mathfrak{F}$, and that the function of every state precedes the function of its successive state according to that order. Because of that, additionally we have to require that function automata are cycle-free.

The absence of an input word with an explicit end demands a different mechanism to determine the end of a run on such an automaton. Therefore, we define a function $\mathrm{end} \in \mathfrak{F}$, that takes no argument and has an empty codomain. Then, a run is defined to end once it reaches a state with $\lambda(q) = \mathrm{end}$.

Definition 3.3.1 (Deterministic Function Automaton)
A *deterministic function automaton* is a cycle-free deterministic finite state machine, defined as the tuple $\mathcal{A} = (Q, \mathfrak{F}, \lambda, \delta, q_0, F)$. Q is the set of states, $\mathfrak{F}$ is a universe of functions with a common domain and a finite codomain. Furthermore, $\mathrm{end} \in \mathfrak{F}$. $\lambda : Q \to \mathfrak{F}$ assigns a function for each state. Let $\Omega = \bigcup_{f \in \mathfrak{F}} \Omega_f$ with $f : D \to \Omega_f \in \mathfrak{F}$. The transition function $\delta : Q \times \Omega \to Q$ maps each state q and element of the codomain of $\lambda(q)$ to the successive state. $q_0 \in Q$ is the starting state and $F \subseteq \{q \in Q \mid \lambda(q) = \mathrm{end}\}$ a set of accepting states. Additionally, $\lambda(q) \preccurlyeq \lambda(\delta(q, \omega)) \forall \omega \in \Omega_{\lambda(q)}$ has to hold. ∎

Note that every state q has only outgoing transitions for all $\omega \in \Omega_{\lambda(q)}$, as it is not possible that $\lambda(q)$ returns something that is not element of $\Omega_{\lambda(q)}$. Note that from $\lambda(q) \preccurlyeq \lambda(\delta(q, \omega)) \forall \omega \in \Omega_{\lambda(q)}$, it follows that $f \preccurlyeq \mathrm{end}$ for all $f \in \mathfrak{F}$.

During a run of a function automaton, at every state the corresponding λ-function is evaluated to determine the next transition. The argument of those functions is the input of the automaton.

Definition 3.3.2 (Run on Function Automaton)
Let $\mathcal{A}$ be a function automaton and $d \in D$. A *run on $\mathcal{A}$ with input* d is a sequence of states $p_0 p_1 \ldots p_{n-1} p_n$ with $p_i = \delta(p_{i-1}, \lambda(p_{i-1})(d)) \forall i \in [1, n]$, $\lambda(p_n) = \mathrm{end}$ and $\lambda(p_i) \neq \mathrm{end}$ for all $i < n$. ∎

Just like for finite state machines, and accepting run is a run that ends in a final state $q \in F$.

Definition 3.3.3 (Accepting Run on Function Automaton)
Let $\mathcal{A}$ be a function automaton. An *accepting run on $\mathcal{A}$* is a run with $p_0 = q_0$ and $p_n \in F$. ∎

Similarly, the language of a function automaton is the set of all input objects which yields an accepting run.

Definition 3.3.4 (Language of a Function Automaton)
Let $\mathcal{A}$ be a function automaton with a λ-function domain D. The *language of* $\mathcal{A}$ is defined as

$$L(\mathcal{A}) = \{d \in D \mid \text{there is an accepting run in } \mathcal{A} \text{ with input } d\}$$

■

The union of two function automata has to be computed differently than the union of finite state machines. In case of finite state machines, the idea is to construct an automaton that simulates both automata. As there is a function assigned to each state, a state of the union automaton is only able to simulate two states of the initial automata if they both have the same λ-function. If they are different, the idea is to pause the execution of one automaton and resume it later, while the other one proceeds to the next state. To decide which automaton is paused, we use the $\prec$ order. We ensure that $\lambda(q) \preccurlyeq \lambda(\delta(q,\omega)) \forall \omega \in \Omega_{\lambda(q)}$ still holds for the resulting automaton by choosing that state as the next one to simulate which has the preceding λ-function.

Definition 3.3.5 (Union of Function Automata)
Let $\mathcal{A}_1 = (Q_1, \mathfrak{F}_1, \lambda_1, \delta_1, q_0^1, F_1)$, $\mathcal{A}_2 = (Q_2, \mathfrak{F}_2, \lambda_2, \delta_2, q_0^2, F_2)$ be function automata.
The *union* $\mathcal{A} = \mathcal{A}_1 \times \mathcal{A}_2$ of $\mathcal{A}_1$ and $\mathcal{A}_2$ is defined as $\mathcal{A} = (Q, \mathfrak{F}, \lambda, \delta, q_0, F)$ with

$$\begin{aligned}
Q &\subseteq Q_1 \times Q_2 \\
\mathfrak{F} &= \mathfrak{F}_1 \cup \mathfrak{F}_2 \\
q_0 &= (q_0^1, q_0^2) \\
\lambda((q_1,q_2)) &= \lambda_i(q_i), \text{ such that } \forall (q_1,q_2) \in Q : \lambda_i(q_i) \preccurlyeq \lambda_j(q_j) \\
&\quad\ \text{with } q_i \in Q_i, q_j \in Q_j, i,j \in \{1,2\}, i \neq j \\
\delta((q_1,q_2),\omega) &= (\delta_1(q_1,\omega), q_2) \text{ if } \lambda_1(q_1) \prec \lambda_2(q_2) \\
\delta((q_1,q_2),\omega) &= (q_1, \delta_2(q_2,\omega)) \text{ if } \lambda_1(q_1) \succ \lambda_2(q_2) \\
\delta((q_1,q_2),\omega) &= (\delta_1(q_1,\omega), \delta_2(q_2,\omega)) \text{ if } \lambda_1(q_1) = \lambda_2(q_2) \\
F &= \{(q_1,q_2) \in Q \mid q_1 \in F_1 \vee q_2 \in F_2, \lambda((q_1,q_2)) = \text{end}\}
\end{aligned}$$

■

Theorem 3.3.1 (Correctness of Union)
Let $\mathcal{A}, \mathcal{B}$ be function automata, $d \in D$. For the union $\mathcal{C} = \mathcal{A} \times \mathcal{B}$, it holds that $L(\mathcal{C}) = L(\mathcal{A}) \cup L(\mathcal{B})$.

Proof: $L(\mathcal{C}) = L(\mathcal{A}) \cup L(\mathcal{B})$ holds if and only if

$$d \in L(\mathcal{A}) \vee d \in L(\mathcal{B}) \Leftrightarrow d \in L(\mathcal{C}) \forall d \in D$$

'$\Leftarrow$': Let $r_\mathcal{C}$ be the sequence of states of a run in $\mathcal{C}$ with $r_\mathcal{C}(i) = c_i = (a_i, b_i) \in Q_\mathcal{A} \times Q_\mathcal{B}$. Now, we project this run to the component of $\mathcal{A}$ as follows. Remove all states of $r_\mathcal{C}$ with $a_i = a_{i+1}$. $r_\mathcal{A}$ contains all states of $Q_\mathcal{A}$ of the remaining tuples. $r_\mathcal{C}$ is projected to $\mathcal{B}$ analogously. An example is shown in Figure 3.1.
Since $r_\mathcal{C}$ is a run, $c_i = \delta(c_{i-1}, \lambda(c_{i-1})(d))$ holds for all $i \in [1, |r_\mathcal{C}|]$. Following from the definition of the union, $\lambda(c_i) = \lambda(a_i)$ if $a_i \neq a_{i+1}$. It further follows that $a_i = \delta(a_{i-1}, \lambda(a_{i-1})(d))$. Thus, $r_\mathcal{A}$ is a run on $\mathcal{A}$. Analogously, it follows that $r_\mathcal{B}$ is a run. Hence, $r_\mathcal{A}$, $r_\mathcal{B}$ or both are

$$
\begin{array}{lccccccccc}
r_{\mathcal{C}} = & (p_1, q_1) & (p_1, q_2) & (p_1, q_3) & (p_2, q_4) & (p_3, q_4) & (p_4, q_4) & (p_5, q_4) & (p_5, q_5) & (p_6, q_5) \\
r_{\mathcal{A}} = & & & (p_1, q_3) & (p_2, q_4) & (p_3, q_4) & (p_4, q_4) & & (p_5, q_5) & (p_6, q_5) \\
r_{\mathcal{B}} = & (p_1, q_1) & (p_1, q_2) & (p_1, q_3) & & & & (p_5, q_4) & & (p_6, q_5)
\end{array}
$$

Figure 3.1: Projection of Run $r_{\mathcal{C}}$ to $\mathcal{A}$ and $\mathcal{B}$.

accepting runs, as $(a_i, b_i) \in F_{\mathcal{C}}$ if and only if $a_i \in F_{\mathcal{A}} \vee b_i \in F_{\mathcal{B}}$ and $\lambda(a_i) = \lambda(b_i) = \lambda(c_i) =$ end.

'$\Rightarrow$': Without loss of generality, assume that $d \in L(\mathcal{A})$. Then, there is an accepting run in $\mathcal{A}$ for input d ending in $q \in F_{\mathcal{A}}$. Let $r_{\mathcal{A}}$ be the sequence of all states of this run in $\mathcal{A}$ with $r_{\mathcal{A}}(i) = a_i$. Let $r_{\mathcal{C}}$ be the run on $\mathcal{C}$ for input d.
By the definition of a run, it holds that $r_{\mathcal{A}}(1) = q_0^{\mathcal{A}}$ and $\delta_{\mathcal{A}}((a_i), \omega) = (a_{i+1})$ with $\lambda_{\mathcal{A}}(a_i)(d) = \omega$. With the construction of the union, we get $r_{\mathcal{C}}(1) = (q_0^{\mathcal{A}}, b)$ with $b \in Q_{\mathcal{B}}$.
For a state $(a_i, b) \in Q_{\mathcal{A}} \times Q_{\mathcal{B}}$, there are two possibilities. In the first case, $\delta_{\mathcal{C}}((a_i, b), \omega) = (a_{i+1}, b')$.
In the second case, $\delta_{\mathcal{C}}((a_i, b_j), \omega) = (a_i, b_{j+1})$. If a_i is not a final state yet, then at some point $\lambda_{\mathcal{A}}(a_i) \prec \lambda_{\mathcal{B}}(b_{j+1})$ has to hold, as any function automaton is finite and there are only finitely many $\lambda_{\mathcal{B}}(b)$ for which $\lambda_{\mathcal{A}}(a_i) \not\succ \lambda_{\mathcal{B}}(b)$. Then, we have again $\delta_{\mathcal{C}}((a_i, b_j), \omega) = (a_{i+1}, b_k)$, which is the first case. Hence, the first component of $r_{\mathcal{C}}$ never leaves the run $r_{\mathcal{A}}$ and eventually reaches the last state of $r_{\mathcal{A}}$, which is an accepting state. It holds that $r_{\mathcal{C}}(|r_{\mathcal{C}}|) = (r_{\mathcal{A}}(|r_{\mathcal{A}}|), r_{\mathcal{B}}(|r_{\mathcal{B}}|))$ and $\lambda_{\mathcal{C}}(r_{\mathcal{C}}(|r_{\mathcal{C}}|)) =$ end. Thus, if $r_{\mathcal{A}}(|r_{\mathcal{A}}|) \in F_{\mathcal{A}}$, then $r_{\mathcal{C}}(|r_{\mathcal{C}}|) \in F_{\mathcal{C}}$, and $r_{\mathcal{C}}$ is an accepting run, too.

$\blacksquare$

3.4 Detecting Embeddings with Function Automata

In this chapter, we will show how function automata can be used to detect embeddings of data structure grammar rules in heap configurations. As we have introduced a suitable automata model, it remains to give a method for constructing an automaton for a specific set of embeddings, which we are going to call *embedding detection automaton*. Γ_N has to be the same for all grammar rules and heap configurations. Following from Theorem 3.2.1, four functions are necessary. Types have to be checked with 'type', the set out outgoing tentacles with 'ot', the degree with 'deg' and equality of two vertices. All those functions take vertices as their argument. In order to be able to use a function automation, the argument of the functions has to be the same in every state. Therefore, we use the path evaluation function $\langle \pi \rangle_{(G,\nu)}$ with a fixed π in every state. Then, the functions of all states are functions of I_{Γ_N}.

Definition 3.4.1 (Input for Embedding Detection Automata)
The *input set for embedding detection automata* D is defined as the set I_{Γ_N} of graph-vertex tuples. $\blacksquare$

The codomain of 'type', Ω_{type}, is equal to the set of types Λ, which is finite. $\Omega_{\text{ot}} = T_{\Gamma_N}$ is finite, too. Checking equality of two vertices yields $\Omega_{=} = \{\text{TRUE}, \text{FALSE}\}$. The codomain of 'deg', however, are the natural numbers. Upon close inspection of Theorem 3.2.1, it turns out that it is only necessary to considers degrees up to a fixed n, the largest degree of all vertices in a grammar rule. If there is a larger degree in some heap configuration G, its exact value is irrelevant and it suffices to define one symbol for this case. So instead of using the 'deg' function, we consider the bounded degree defined for an $n \in \mathbb{N}$ as follows.

Definition 3.4.2 (Bounded Vertex Degree)
Let $G \in HC_{\Gamma_N}$. The *bounded degree* of a vertex is defined as

$$\mathrm{degree}_{\leqslant n}(v) = \begin{cases} \mathrm{degree}(v) & \text{if } \mathrm{degree}(v) \leqslant n \\ * & \text{otherwise} \end{cases}$$

For short, we write $\deg_{\leqslant n}(v)$. ∎

Hence, we obtain $\Omega_{\deg_{\leqslant n}} = \{1, \dots, n, *\}$, which obviously is finite, too. If we now want compute the union of two automata with different degree functions, it could happen that two states with degree functions should be merged, which is not possible because those functions are not the same anymore. It is possible, however, to increase the degree of an automaton without changing its language. Let $m, n \in \mathbb{N}$ with $n < m$ and $i \in [n+1, m]$. Suppose we have an automaton with degree $n \in \mathbb{N}$. If there is a vertex with degree i, $\deg_{\leqslant n}$ returns $*$, as i is larger than n, and the next state is $\delta(q, *)$. For the same i, $\deg_{\leqslant m}$ returns i. Hence, as the language is supposed to remain unchanged, $\delta(q, i) = \delta(q, *)$ has to hold for all $i \in [n+1, m]$. Thus, without loss of generality, we can assume in the following that n is the same for all automata when computing the union.

Finally, we need an ordering for all λ-functions needed for detecting embeddings. The actual order of 'type', '$\deg_{\leqslant n}$', 'ot' and equality is arbitrary. Technically, however, one path, or in case of equality two paths, are part the function now, and functions with different path are not equal. Therefore, it is natural to use the path order.

Definition 3.4.3 (λ-Function Order)
Let $g \in I_{\Gamma_N}$. The λ-*function order* is an order on functions symbols in combination with their argument, not on the returned value. It is defined as follows:

$$\begin{aligned} \mathrm{type}(\langle \pi_1 \rangle_g) &\prec \mathrm{type}(\langle \pi_2 \rangle_g) \text{ if } \pi_1 \prec \pi_2 \\ \deg_{\geqslant n}(\langle \pi_1 \rangle_g) &\prec \deg_{\geqslant n}(\langle \pi_2 \rangle_g) \text{ if } \pi_1 \prec \pi_2 \\ \mathrm{ot}(\langle \pi_1 \rangle_g) &\prec \mathrm{ot}(\langle \pi_2 \rangle_g) \text{ if } \pi_1 \prec \pi_2 \\ (\langle \pi_1 \rangle_g = \langle \pi_1' \rangle_g) &\prec (\langle \pi_2 \rangle_g = \langle \pi_2' \rangle_g) \text{ if } \pi_1 \prec \pi_2 \text{ or if } \pi_1 = \pi_2 \text{ and } \pi_1' \prec \pi_2' \\ \mathrm{type}(\langle \pi \rangle_g) &\prec \deg_{\geqslant n}(\langle \pi \rangle_g) \prec \mathrm{ot}(\langle \pi \rangle_g) \prec (\langle \pi \rangle_g = \langle \pi' \rangle_g) \prec \mathrm{end} \end{aligned}$$

The λ-function order is symmetric, transitive and $a \preccurlyeq b$ holds if and only if $a = b \vee a \prec b$. ∎

Now, we can proceed to the actual construction. In a first step, we construct linear embedding detection automata for each rule, which are loop-free automata consisting solely of a chain of states and an additional sink state. To make sure that a run always terminates, it has to be $\lambda(q) = \mathrm{end}$ for the sink state and the last state of the chain of states. Then, given those linear automata, we can construct one automaton that detects all embeddings at once by iteratively computing the union of two those automata.

For paths leading to internal vertices, it can be read off Theorem 3.2.1 that three states are needed which check the type (3.1), the set of outgoing tentacles (3.2) and the degree (3.3). In case of external vertices with no outgoing edges, the differences are that the degree of the vertex in G might be larger than that of the vertex in H (3.5), and the set out outgoing tentacles just has to match the type, which is always the case for heap configurations.

If a path in H leads to a vertex that is external and has outgoing edges, additionally to checking the the set of outgoing tentacles is the same (3.4), it it necessary to make sure that it is not equal to any other external vertex with outgoing edges (3.7). To verify that condition, we make sure that this vertex is not equal to any of the vertices in question previously visited. Thus, we have to add one state for every previously visited vertex. With that, in the end all pairs of those vertices are checked exactly once.

In order to keep the overall number of states as small as possible, instead of verifying all properties of a vertex for every time it has been visited in H, we just check if it is equal to the first occurrence once we visited it the second time.

All transitions which do not lead to sink are then defined to lead to the state with the successive function according to the λ-function order, such that $\lambda(q) \preccurlyeq \lambda(\delta(q,\omega)) \forall \omega \in \Omega_{\lambda(q)}$ is satisfied.

During the construction of the automata, a sequence s will be used to store all vertices of H in order of their appearance during a depth-first search. Another sequence s' contains all external vertices with outgoing edges in the same order as in s to construct those states that verify pairwise inequality. The set Dejavu stores pairs of indices of vertices of s which are the same such that it is possible to compare a subsequent appearance to the previous one.

Definition 3.4.4 (Embedding Detection Automata)
Let $\mathcal{A}$ be a function automaton. $\mathcal{A}$ is an *embedding detection automaton* for a set of embeddings $A \subseteq \mathrm{Emb}_{\Gamma_N}$ if $L(\mathcal{A}) = \{(G,v) \in I_{\Gamma_N} \mid a \in A$, there is an a-embedding in G at $v\}$.
Embedding detection automata are constructed as follows. First, for every embedding $(X \to H, i) \in \mathrm{Emb}^{\mathcal{A}}_{\Gamma_N}$, construct a *linear embedding detection automaton*.

(1) Determine the largest degree in H, $d = \max\{\mathrm{degree}(v) \mid v \in V_H\}$.

(2) Compute Π_{DFS} with DFSPath$(H, \mathrm{ext}_H(i))$. Construct a sequence of vertices $s_H = v_1 \ldots v_n$ such that $\langle \pi_k \rangle_{(H,\mathrm{ext}_H(i))} = s(k)$, $\langle \pi_{k+1} \rangle_{(H,\mathrm{ext}_H(i))} = s(k+1)$, $\pi_k, \pi_{k+1} \in \Pi_{DFS}$ and $\pi_k \prec \pi_{k+1}$ for all $k \in [1, n-1]$. Note that this sequence may contain vertices more than once as it has to contain even those which have already been visited before by the depth-first search.

Construct a set $\mathrm{Dejavu}_H = \{(a,b) \mid s(a) = s(b)$ and either $s(c) \neq s(b), a < c < b$ or $a+1 = b\}$, containing pairs of indices of vertices where a is the largest index for which $s(a)$ is equal to $s(b)$.

Finally, construct a set of paths to external non-reduction vertices $\Pi'_{DFS} = \{\pi_k \in \Pi_{DFS} \mid \langle \pi_k \rangle_{(H,\mathrm{ext}_H(i))} = \mathrm{ext}_H(l), \mathrm{ot}(\mathrm{ext}_H(l)) \neq \emptyset, l \in [1, |\mathrm{ext}_H|], \neg\exists(a,k) \in \mathrm{Dejavu}_H\}$ and a sequence s' such that $\langle \pi'_k \rangle_{(H,\mathrm{ext}_H(i))} = s'(k)$, $\langle \pi'_{k+1} \rangle_{(H,\mathrm{ext}_H(i))} = s'(k+1)$, $\pi'_k, \pi'_{k+1} \in \Pi_{DFS}$ and $\pi'_k \prec \pi'_{k+1}$ for all $k \in [1, |\mathrm{ext}_H| - 1]$.

(3) Add a sink state $q_{sink} \in Q$ with $\lambda(q)(G,v) = \mathrm{end}$.

(4) For all $\nu_k = s(k)$ with $k \in [1, n]$, check if $\exists a : (a, k) \in$ Dejavu. If that is the case, construct one state $q_k^1 \in Q$ with

$$\lambda(q_k^1)(G, \nu) = (\langle \pi_k \rangle_{(G,\nu)} = \langle \pi_a \rangle_{(G,\nu)})$$
$$\delta(q_k^1, \text{TRUE}) = q_{k+1}^1$$

Note that all transitions which are not explicitly mentioned are assumed to lead to q_{sink}. If $\neg\exists(a, k) \in$ Dejavu, do the following. Construct three states $q_k^1, q_k^2, q_k^3 \in Q$ with

$$\lambda(q_k^1)(G, \nu) = \text{type}(\langle \pi_k \rangle_{(G,\nu)})$$
$$\lambda(q_k^2)(G, \nu) = \deg_{\geqslant d}(\langle \pi_k \rangle_{(G,\nu)})$$
$$\lambda(q_k^3)(G, \nu) = \text{ot}(\langle \pi_k \rangle_{(G,\nu)})$$

(4.1) If $\nu_k \notin [\text{ext}_H]$, define

$$\delta(q_k^1, \text{type}(\nu_k)) = q_k^2$$
$$\delta(q_k^2, \deg(\nu_k)) = q_k^3$$
$$\delta(q_k^3, \text{ot}(\nu_k)) = q_{k+1}^1$$

(4.2) If $\nu_k \in [\text{ext}_H]$ and $\text{ot}(\nu_k) = \emptyset$, define

$$\delta(q_k^1, \text{type}(\nu_k)) = q_k^2$$
$$\delta(q_k^1, \perp) = q_k^2$$
$$\delta(q_k^2, \deg(\nu_k)) = q_k^3$$
$$\delta(q_k^2, *) = q_k^3$$
$$\delta(q_k^3, \omega) = q_{k+1}^1 \forall \omega \in \Omega_{\text{ot}}^{\text{type}_H(\nu_k)} \cup \{\emptyset\}$$

(4.3) If $\nu_k \in [\text{ext}_H]$ and $\text{ot}(\nu_k) \neq \emptyset$, then there is some j with $\nu_k = s'(j)$. Construct $j - 1$ states $q_k^4, \ldots, q_k^{j+2} \in Q$ with

$$\lambda(q_k^m)(G, \nu) = (\langle \pi_k \rangle_{(G,\nu)} = \langle \pi'_{m-2} \rangle_{(G,\nu)}) \forall m \in [4, j+1]$$
$$\delta(q_k^1, \text{type}(\nu_k)) = q_k^2$$
$$\delta(q_k^2, \deg(\nu_k)) = q_k^3$$
$$\delta(q_k^2, *) = q_k^3$$
$$\delta(q_k^3, \text{ot}(\nu_k)) = q_k^4 \text{ if } j \neq 1$$
$$\delta(q_k^3, \text{ot}(\nu_k)) = q_{k+1}^1 \text{ if } j = 1$$
$$\delta(q_k^m, \text{FALSE}) = q_k^{m+1} \forall m \in [4, j]$$
$$\delta(q_k^{j+1}, \text{FALSE}) = q_{k+1}^1$$

(5) Define for the initial state $q_0 = q_1^1$.

If there is an accepting run for (G, ν) in $\mathcal{A}$, there is a $(X \to H, i)$-embedding in G at ν since all requirements of Theorem 3.2.1 are fulfilled.

To obtain an automaton $\mathcal{B}$ for a set of embeddings $A = \{a_1, \ldots, a_b\} \subseteq \mathrm{Emb}_{\Gamma_N}$, construct a linear automaton $\mathcal{A}_c$ for each $a_c \in A$ and iteratively compute the union $\mathcal{B} = ((\cdots(\mathcal{A}_1 \times \mathcal{A}_2) \times \ldots) \times \mathcal{A}_b)$. In order to identify which embedding or embeddings were found, take the final state $q = (q_1, \ldots, q_b)$ and check for all $k \in [1, b]$ if $q_k \in F_k$. If this is true, then there is an a_k-embedding in G at ν. ■

For short, we will refer to embedding detection automata as EDAs. Now, in order to find all embeddings in a heap configuration G, we have to execute this automata once for every $\nu \in V_G$ with input (G, ν).

3.5 Algorithm and Time Complexity

In this section, we will provide algorithms for constructing embedding detection automata and using them to find embeddings. Furthermore, we will analyze the time complexity of those algorithms.

Theorem 3.5.1 (Time Complexity of the Union)
The time complexity of the union of two embedding detection automata $\mathcal{A}_1$ and $\mathcal{A}_2$ for sets of embeddings A_1 and A_2, respectively, with $(X \to H, i) \in (A_1 \cup A_2)$ and H being the largest rule graph, is in $\mathcal{O}((|Q_1| + |Q_2|) \cdot |V_H| \cdot |\Omega|)$.

Proof: Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be embedding detection automata for sets of embeddings A_1 and A_2, respectively, with $(X \to H, i) \in (A_1 \cup A_2)$. $\max(\Omega_{deg})$ is in $\mathcal{O}(|\Omega_{deg}|)$, when doing a linear search on Ω_{deg}. The first outer loop is executed $|Q|$ times, the loop inside is in $\mathcal{O}(|\Omega_{deg}|)$, as in the worst case, n_k is 1. The 'if' block is in $\mathcal{O}(1)$. It follows that over all, this loop is in $\mathcal{O}(|\Omega_{deg}| \cdot |Q|)$.
The 'while' loop performs something similar to a depth-first search on $\mathcal{A}$. The number of reachable states of $\mathcal{A}$ is bounded by $|Q_1| + |Q_2|$, as in every state of $\mathcal{A}$, at least one automaton proceeds to the next state. In the worst case, this loop is executed $|Q_1| + |Q_2|$ times as only those states are put one the stack which were not reached yet. Using an array, storing states q_i at index i, testing set membership can be computed in $\mathcal{O}(1)$. Computing the union $F := F \cup \{(q_1, q_2)\}$ is in $\mathcal{O}(1)$, too, when using a hash table. In order to compare $\lambda_1(q_1)$ and $\lambda_2(q_2)$, it may be necessary to compare the paths of that function, which were obtained by a depth-first search on a heap configuration. Hence, the length of those paths is bounded by $|V_H|$. Only one of these 'if' blocks is executed at a time and nextState is in $\mathcal{O}(|\Omega|)$, as all operations inside are in $\mathcal{O}(1)$ and the loop is executed $|\Omega|$ times.
That results in a time complexity of $\mathcal{O}(|\Omega_{deg}| + |\Omega_{deg}| \cdot |Q| + (|Q_1| + |Q_2|) \cdot |V_H| \cdot |\Omega|) = \mathcal{O}((|Q_1| + |Q_2|) \cdot |V_H| \cdot |\Omega|)$. ■

It should be mentioned, however, that this result is merely a rough upper bound. Usually, only a fraction of all states of $\mathcal{A}$ is reachable.

In the following, we quite often need an upper bound for the maximum number of successors a vertex might have. On a heap configuration, a vertex ν has has either all outgoing selectors as specified by the sequence of type selectors, or they are all abstracted inside one nonterminal. In case of the nonterminal edge, the maximum number of successors of ν is the rank of that edge minus one, as every tentacle can be attached to a different vertex. Subtracting one, we take into account that one tentacle is always attached to the initial vertex

Function union($\mathcal{A}_1$, $\mathcal{A}_2$) Construction of Union

```
Input: A_1 = (Q_1, 𝔉_1, λ_1, δ_1, q_0^1, F_1), A_2 = (Q_2, 𝔉_2, λ_2, δ_2, q_0^2, F_2)
Output: A = (Q, 𝔉, λ, δ, q_0, F)

n_1 := max(Ω_{1 deg})
n_2 := max(Ω_{2 deg})
if n_1 > n_2 then
    k := 2
    l := 1
else
    k := 1
    l := 2
forall q ∈ Q_k do
    forall m ∈ [n_k + 1, n_l] do
        δ_k(q, m) := δ_k(q, ∗)
    if λ_k(q)(G, v) = deg_{≥n_k}(⟨π⟩_{(G,v)}) then
        λ_k(q)(G, v) := deg_{≥n_l}(⟨π⟩_{(G,v)})
Q := ∅
F := ∅
push((q_0^1, q_0^2))
while stack not empty do
    (q_1, q_2) := pop()
    if (q_1 ∈ Q_1 ∨ q_2 ∈ Q_2) ∧ λ(q) = end then
        F := F ∪ {(q_1, q_2)}
    Q := Q ∪ {(q_1, q_2)}
    if λ_1(q_1) ≺ λ_2(q_2)) then
        nextState(q_1, q_2, δ_1(q_1, ω), q_2, λ_1(q_1))
    else if λ_1(q_1) ≻ λ_2(q_2) then
        nextState(q_1, q_2, q_1, δ_2(q_2, ω), λ_2(q_2))
    else if λ_1(q_1) = λ_2(q_2) then
        nextState(q_1, q_2, δ_1(q_1, ω), δ_2(q_2, ω), λ_2(q_2))
```

Function nextState($q_1 \in Q_1, q_2 \in Q_2, q'_1 \in Q_1, q'_2 \in Q_2, \lambda(q)$)

```
forall ω ∈ Ω do
    δ((q1, q2), ω) := (q'1, q'2)
    if (q'1, q'2) ∉ Q then
        push((q'1, q'2))
    λ(q'1, q'2) := λ(q)
    𝔉 := 𝔉 ∪ {λ(q)}
```

v. Therefore, it is useful to define the branching factor of a heap configuration as either the largest rank of any edge minus one, or the length of the longest sequence of type selectors, depending on which number is larger.

Definition 3.5.1 (Branching Factor)
Let $G \in HC_{\Gamma_N}$. The *branching factor* of G is defined as $B_G = \max(\max_{e \in E_G}(rk(e)) - 1, \max_{t \in \Lambda}(|pt(t)|))$. ■

Theorem 3.5.2 (Time Complexity of DFSPath(G, v))
Let $(G, v) \in I_{\Gamma_N}$. The time complexity of DFSPath(G, v) is $\mathcal{O}(|V_G|^2 \cdot B_G)$.

Proof: There are two cases to distinguish for determining the size of $|\{(lab(e), i) \mid e \in nREdge(v), i \in [1, rk(e)], att_G(e)(i) \neq v\}|$. The first case is that all outgoing edges are abstracted in one nonterminal. In this case, the size is bounded by the highest rank of any edge in G. In the second case, v has concrete outgoing edges. Then, there are $|pt(type(v))|$ many, as v has to be complete. Hence, the size of this set is bounded by the branching factor B_G. Evaluating a path π on G is in $|V_G|$. DFSPath recursively calls itself as long as there are vertices left which are not marked as visited. Thus, it calls itself $|V_G|$ times. The overall complexity of DFSPath hence is $\mathcal{O}(|V_G|^2 \cdot B_G)$. The resulting Π_{DFS} contains $\mathcal{O}(|V_G| \cdot B_G)$ elements, as in each execution of the loop body, one path is added to Π_{DFS}. ■

Theorem 3.5.3 (Time Complexity of linearEDA$((X \to H, i))$)
Let $(X \to H, i) \in Embr_{\Gamma_N}$. The time complexity of linearEDA$((X \to H, i))$ is $\mathcal{O}(|V_H|^3 \cdot B_H^2 + |ext_H|^2 \cdot |V_H| + |V_H|^2 \cdot B_H^2 \cdot |\Omega| \cdot |ext_H|))$.

Proof: As stated in Theorem 3.5.2, DFSPath is in $\mathcal{O}(|V_H|^2 \cdot B_H)$. Sorting Π_{DFS} needs in the worst case $\mathcal{O}(|\Pi_{DFS}| \log(|\Pi_{DFS}|))$ comparisons, when using for example heapsort [2]. As Π_{DFS} contains paths on H, each comparison takes $\mathcal{O}(|V_H|)$ operations, resulting in $\mathcal{O}(|\Pi_{DFS}| \cdot \log(|\Pi_{DFS}|) \cdot |V_H|) = \mathcal{O}(|V_H| \cdot B_H \cdot \log(|V_H| \cdot B_H) \cdot |V_H|) = \mathcal{O}(|V_H|^2 \cdot B_H \cdot \log(|V_H| \cdot B_H))$. Computing s is in $\mathcal{O}(|V_H|^2 \cdot B_H)$, as there are $|\Pi_{DFS}|$ paths to evaluate on H.
$Dejavu_H$ can be computed in $\mathcal{O}(|s|^2) = \mathcal{O}(|V_H|^2 \cdot B_H^2)$ by looking at every $s(k)$ and then going back towards the beginning of s, comparing each $s(l)$ with $s(k)$ for all $l < k$ until either a pair (l, k) with $s(l) = s(k)$ is found that has to be added to $Dejavu_H$ or until the beginning of the sequence is reached.
In order to construct Π'_{DFS}, it is necessary to again evaluate all paths of Π_{DFS} and search $Dejavu_H$. The number of elements of $Dejavu_H$ is $\mathcal{O}(|s|) = \mathcal{O}(|V_H| \cdot B_H)$, as in the worst case, all $s(k)$ are the same. Then, $Dejavu_H$ contains all pairs $(k, k+1)$ for $k \in [1, |s| - 1]$. Hence, constructing Π'_{DFS} is in $\mathcal{O}(|V_H| \cdot B_H \cdot |V_H| \cdot B_H) = \mathcal{O}(|V_H|^2 \cdot B_H^2)$.

Function linearEDA($(X \to H, i) \in \mathrm{Emb}_{\Gamma_N}$)

Output: $\mathcal{A} = (Q, \mathfrak{F}, \lambda, \delta, q_0, F)$

$\pi := \varepsilon$

$\Pi_{DFS} := \{\varepsilon\}$

DFSPath(H, v)

$\Pi^s_{DFS} :=$ Sort(Π_{DFS})

$s := \langle \Pi^s_{DFS} \rangle_{(H, ext_H(i))}$

$\mathrm{Dejavu}_H := \{(a, b) \mid s(a) = s(b)$ and either $s(c) \neq s(b), a < c < b$ or $a + 1 = b\}$

$\Pi'_{DFS} = \{\pi_k \in \Pi_{DFS} \mid \langle \pi_k \rangle_{(H, ext_H(i))} = ext_H(l), ot(ext_H(l)) \neq \emptyset, l \in [1, |ext_H|], \neg\exists (a, k) \in \mathrm{Dejavu}_H\}$

$\Pi'^s_{DFS} :=$ Sort(Π'_{DFS})

$s' := \langle \Pi^{s'}_{DFS} \rangle_{(H, ext_H(i))}$

$d := \max\{\mathrm{degree}(v) \mid v \in V_H\}$

forall $k \in [1, |s|]$ **do**

- $v_k := s(k)$
- **if** $\exists a : (a, k) \in \mathrm{Dejavu}_H$ **then**
  - $Q := Q \cup \{q^1_k\}$
  - $\lambda(q^1_k)(G, v) = (\langle \pi_k \rangle_{(G,v)} = \langle \pi_a \rangle_{(G,v)})$
  - addTransition(q^1_k, q^1_{k+1}, TRUE)
- **else**
  - **if** $v_k \notin [ext_H]$ **then**
    - internal(v_k, π_k)
  - **else if** $v_k \in [ext_H] \wedge ot(v_k) = \emptyset$ **then**
    - externalR(v_k, π_k)
  - **else if** $v_k \in [ext_H] \wedge ot(v_k) \neq \emptyset$ **then**
    - externalNR(v_k, π_k)

$q_0 := q^1_1$

Function internal($v_k \in V_H, \pi_k \in \Pi_{DFS}$)

$Q := Q \cup \{q_k^1, q_k^2, q_k^3\}$
$\lambda(q_k^1)(G,v) = \text{type}(\langle \pi_k \rangle_{(G,v)})$
$\lambda(q_k^2)(G,v) = \deg_{\geqslant d}(\langle \pi_k \rangle_{(G,v)})$
$\lambda(q_k^3)(G,v) = \text{ot}(\langle \pi_k \rangle_{(G,v)})$
addTransition(q_k^1, q_k^2, type(v_k))
addTransition(q_k^2, q_k^3, deg(v_k))
addTransition(q_k^3, q_{k+1}^1, ot(v_k))

Function externalR($v_k \in V_H, \pi_k \in \Pi_{DFS}$)

$Q := Q \cup \{q_k^1, q_k^2, q_k^3\}$
$\lambda(q_k^1)(G,v) = \text{type}(\langle \pi_k \rangle_{(G,v)})$
$\lambda(q_k^2)(G,v) = \deg_{\geqslant d}(\langle \pi_k \rangle_{(G,v)})$
$\lambda(q_k^3)(G,v) = \text{ot}(\langle \pi_k \rangle_{(G,v)})$
addTransitions(q_k^1, q_k^2, $\{\text{type}(v_k), \perp\}$)
addTransitions(q_k^2, q_k^3, $\{\deg(v_k), \ldots, d, *\}$)
addTransitions(q_k^3, q_{k+1}^1, $\Omega_{ot}^{\text{type}_H(v_k)} \cup \{\emptyset\}$)

The number of elements of Π'_{DFS} is equal to $|\text{ext}_H|$, as it contains one path for every external vertex. Thus, sorting those paths takes at most $|\text{ext}_H| \cdot \log(|\text{ext}_H|) \cdot |V_H|$ operations. Computing s' is in $\mathcal{O}(|\text{ext}_H| \cdot |V_H|)$. $\max\{\text{degree}(v) \mid v \in V_H\}$ takes $|V_H|$ operations.
The body of the loop is executed $\mathcal{O}(|s|) = \mathcal{O}(|V_H| \cdot B_H)$ times. In every execution, Dejavu$_H$ is searched once ($\mathcal{O}(|V_H| \cdot B_H)$). The first two 'if' blocks are in $\mathcal{O}(|\Omega|)$, as a fixed number of states are created and transitions are added for every $\omega \in \Omega_{\lambda(q)}$ with the functions internal and externalR. In the last block, up to $|s'| = |\text{ext}_H|$ states are added by externalNR. That, however, happens only once for every external vertex.
Hence, the entire loop is in $\mathcal{O}(|V_H| \cdot B_H \cdot |V_H| \cdot B_H \cdot |\Omega| \cdot |\text{ext}_H|) = \mathcal{O}(|V_H|^2 \cdot B_H^2 \cdot |\Omega| \cdot |\text{ext}_H|)$. Omitting the Ω and the operations needed for searching Dejavu$_H$, we get an approximation for the number of states of $\mathcal{A}$, that is $\mathcal{O}(|V_H| \cdot B_H \cdot |\text{ext}_H|)$.
For the time complexity, we get $\mathcal{O}(|V_H|^2 \cdot B_H + |V_H|^2 \cdot B_H \cdot \log(|V_H| \cdot B_H) + |V_H|^2 \cdot B_H + |V_H|^2 \cdot B_H^2 + |V_H|^2 \cdot B_H^2 + |\text{ext}_H| \cdot \log(|\text{ext}_H|) \cdot |V_H| + |\text{ext}_H| \cdot |V_H| + |V_H| + |V_H|^2 \cdot B_H^2 \cdot |\Omega| \cdot |\text{ext}_H|)) = \mathcal{O}(|V_H|^2 \cdot B_H \cdot \log(|V_H| \cdot B_H) + |\text{ext}_H| \cdot \log(|\text{ext}_H|) \cdot |V_H| + |V_H|^2 \cdot B_H^2 \cdot |\Omega| \cdot |\text{ext}_H|))$. ■

Theorem 3.5.4 (Time Complexity of EDA(A))
Let $A = \{a_1, \ldots, a_b\} \subseteq \text{Embr}_{\Gamma_N}$ with $(X \to H, i) \in A$. The time complexity of EDA(A) is $\mathcal{O}(|A| \cdot |V_H|^2 \cdot B_H \cdot |\text{ext}_H| \cdot |\Omega|)$.

Proof: The first loop is in $\mathcal{O}(|A| \cdot |V_H|^3 \cdot B_H^2 + |A| \cdot |\text{ext}_H|^2 \cdot |V_H| + |A| \cdot |V_H|^2 \cdot B_H^2 \cdot |\Omega| \cdot |\text{ext}_H|))$. The first iteration of the second loop is in $\mathcal{O}((|Q_2| + |Q_1|) \cdot |V_H| \cdot |\Omega|)$. The second is in $\mathcal{O}((|Q_3| + |Q_2|) \cdot |V_H| \cdot |\Omega|)$. The number of elements in Q_2, however, is now equal to what was in the last iteration $|Q_2| + |Q_1|$, resulting in $\mathcal{O}((|Q_3| + |Q_2| + |Q_1|) \cdot |V_H| \cdot |\Omega|)$. In the final iteration, we have $\mathcal{O}((|Q_k| + \ldots + |Q_1|) \cdot |V_H| \cdot |\Omega|)$, which is equal to the overall time complexity of that loop according to the $\mathcal{O}$ notation. That can be simplified to $\mathcal{O}(|A| \cdot |Q| \cdot |V_H| \cdot |\Omega|)$. Together with the first loop, we obtain $\mathcal{O}(|A| \cdot |V_H|^3 \cdot B_H^2 + |A| \cdot |\text{ext}_H|^2 \cdot |V_H| +$

Function externalNR($v_k \in V_H, \pi_k \in \Pi_{DFS}$)

---

find j such that $v_k = s'(j)$
$Q := Q \cup \{q_k^1, \ldots, q_k^{j+2}\}$
$\lambda(q_k^1)(G, v) = \text{type}(\langle \pi_k \rangle_{(G,v)})$
$\lambda(q_k^2)(G, v) = \text{deg}_{\geqslant d}(\langle \pi_k \rangle_{(G,v)})$
$\lambda(q_k^3)(G, v) = \text{ot}(\langle \pi_k \rangle_{(G,v)})$
$\lambda(q_k^m)(G, v) = (\langle \pi_k \rangle_{(G,v)} = \langle \pi'_{m-2} \rangle_{(G,v)}) \forall m \in [3, j+1]$
addTransition(q_k^1, q_k^2, type(v_k))
addTransitions(q_k^2, q_k^3, $\{\text{deg}(v_k), \ldots, d, *\}$)
addTransition(q_k^3, q_k^4, ot(v_k)) if $j \neq 1$
addTransition(q_k^3, q_{k+1}^1, ot(v_k)) if $j = i$
addTransition(q_k^m, q_k^{m+1}, FALSE) $\forall m \in [4, j]$
addTransition(q_k^{j+1}, q_{k+1}^1, FALSE)

Function addTransition($q_1 \in Q, q_2 \in Q, \omega \in \Omega_{\lambda(q)}$)

$\delta(q_1, \omega) = q_2$
$\delta(q_1, \omega') = q_{sink} \forall \omega' \in \Omega_{\lambda(q)} \setminus \{\omega\}$

$|A| \cdot |V_H|^2 \cdot B_H^2 \cdot |\Omega| \cdot |\text{ext}_H| + |A| \cdot |Q| \cdot |V_H| \cdot |\Omega|)$. Using the approximation for the number of states of $\mathcal{A}$ from Theorem 3.5.3, we can simplify even further to $\mathcal{O}(|A| \cdot |V_H|^3 \cdot B_H^2 + |A| \cdot |\text{ext}_H|^2 \cdot |V_H| + |A| \cdot |V_H|^2 \cdot B_H^2 \cdot |\Omega| \cdot |\text{ext}_H| + |A| \cdot (|V_H| \cdot B_H \cdot |\text{ext}_H|) \cdot |V_H| \cdot |\Omega|) = \mathcal{O}(|A| \cdot |V_H|^2 \cdot B_H \cdot |\text{ext}_H| \cdot |\Omega|)$. ■

The length of the longest run on an embedding detection automata indicates the time complexity of embedding detection. Intuitively, in the best case, all grammar rules exclude each other, for example by having a different type of the initial vertex. In this case, for any number of grammar rules, the automaton branches immediately and solely the properties of the only potentially matching rule are verified. It is, however, more realistic to assume that there are rules with the same types, for example a set of rules for lists and another set for trees. In this case, the types immediately rule out a number of grammar rules. For the remaining rules, there might be vertices which have the same properties in all rules. In this case, using an automaton pays off as those properties are only checked once. The worst case is that no two rules exclude each other, which means that there could be at the same time an embedding of every rule at one vertex, while they share only a few common vertices. Consider Figure 3.2 for an example. The highlighted vertices form one possible data structure grammar rule. The dashed lines symbolize arbitrarily long list. It is possible to construct three other grammar rules, each containing one list. As those rules do not exclude each other, searching for those embeddings requires to traverse all four list. If the number of vertices of those list is much larger than the number of vertices in the tree, the number of vertices which have to be visited approximately adds up. In this case, using one automaton for a set of rules is not much more efficient than using a linear automaton for every single rule. This case, however, is very unlikely to happen for realistic data structure grammars. There are, however, other upper bounds, which are better under certain circumstances.

Function addTransitions($q_1 \in Q, q_2 \in Q, \Omega' \in \Omega_{\lambda(q)}$)

$\delta(q_1, \omega') = q_2 \forall \omega' \in \Omega'$
$\delta(q_1, \omega') = q_{\text{sink}} \forall \omega' \in \Omega_{\lambda(q)} \setminus \Omega'$

Function EDA($A \subseteq \text{Emb}_{\Gamma_N}$)

Input: $A = \{a_1, \dots, a_b\} \subseteq \text{Emb}_{\Gamma_N}$
Output: $\mathcal{A} = (Q, \mathfrak{F}, \lambda, \delta, q_0, F)$

forall $k \in [1, b]$ do
  $\mathcal{A}_k :=$ linearEDA(a_k)
forall $k \in [2, b]$ do
  $\mathcal{A}_k :=$ union($\mathcal{A}_k, \mathcal{A}_{k-1}$)
$\mathcal{A} := \mathcal{A}_b$

Theorem 3.5.5 (Length of the Longest Run on EDA)
Let $G \in \text{HC}_{\Gamma_N}$, $A = \{a_1, \dots, a_b\} \subseteq \text{Emb}_{\Gamma_N}$. The lower bound for the longest run of $\mathcal{A} = \text{EDA}(A)$ is the largest $|Q_k|$ of all $\mathcal{A}_k = \text{EDA}(\{a_k\})$ with $k \in [1, b]$. Two upper bounds are

$$\mathcal{O}\left(\sum_{k=1}^{b} |Q_k|\right) = \mathcal{O}\left(|A| \cdot |V_H| \cdot B_H \cdot |\text{ext}_H|\right) \text{ and } \mathcal{O}\left(\sqrt[e]{e}^{|V_H|}\right)$$

Proof: Let $\mathcal{A}_1, \mathcal{A}_2$ be embedding detection automata for sets of embeddings A_1, A_2, with $(X_1 \to H_1, i_1) \in A_1$ and $(X_2 \to H_2, i_2) \in A_2$. If $\text{type}(\text{ext}_{H_1}(i_1)) \neq \text{type}(\text{ext}_{H_2}(i_2))$ for all $(X_1 \to H_1, i_1) \in A_1$, $(X_2 \to H_2, i_2) \in A_2$, there are two branching path from the initial state (q_0^1, q_0^2) of $\mathcal{A} = \mathcal{A}_1 \times \mathcal{A}_2$. All transitions from the initial state with $\omega \neq \text{type}(\text{ext}_{H_1}(i_1))$ and $\omega \neq \text{type}(\text{ext}_{H_2}(i_2))$ lead to $(q_{\text{sink}}^1, q_{\text{sink}}^2)$. For $\omega = \text{type}(\text{ext}_{H_1}(i_1))$, it is $(\delta_1(q_0^1, \omega), \delta_2(q_0^2, \omega)) = (\delta_1(q_0^1, \omega), q_{\text{sink}}^2)$, and conversely $(\delta_1(q_0^1, \omega), \delta_2(q_0^2, \omega)) = (q_{\text{sink}}^1, \delta_2(q_0^2, \omega))$ for $\omega = \text{type}(\text{ext}_{H_2}(i_2))$. Thus, it is not possible that those two paths merge in another state than $(q_{\text{sink}}^1, q_{\text{sink}}^2)$. It follows that in this case, the longest run of $\mathcal{A}$ is as long as the longest run from $\mathcal{A}_1$ and $\mathcal{A}_2$, which are in $\mathcal{O}(|Q_1|)$ and $\mathcal{O}(|Q_2|)$, respectively, as they are loop-free.
If it is possible that there is at the same time an $(X_1 \to H_1, i_1)$- and $(X_2 \to H_2, i_2)$-embedding

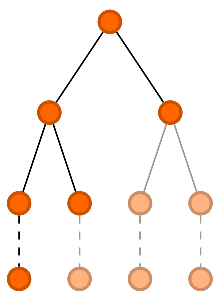

Figure 3.2: Example of a Worst Case Grammar Rule.

at some $\nu \in V_G$, all requirements of those embeddings have to be checked by $\mathcal{A}$ (following from Theorem 3.3.1). In the worst case, there are $|Q_1| + |Q_2|$ different $\lambda(q)$ to check. Thus, and accepting run of $\mathcal{A}$ is at most as long as the accepting runs of $\mathcal{A}_1$ and $\mathcal{A}_2$ together. That is in $\mathcal{O}(|Q_1| + |Q_2|)$.

For a fixed number of vertices of grammar rules $|V_H|$, there is only a finite number of different possible grammar rules. The largest number of vertices that have to be checked in order to detect embeddings of such grammar rules is the size of the largest heap configuration that can be obtained by merging grammar rules of size $|V_H|$ which do not exclude each other.

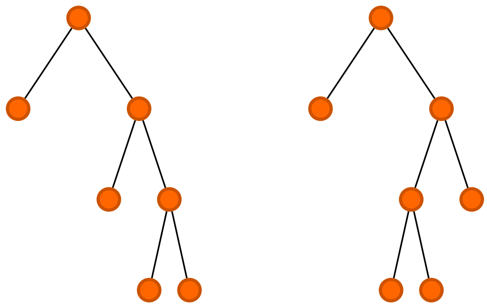

Figure 3.3: Shape of Grammar Rules with Fixed Size $|V_H|$.

The largest combined graph is obtained with rules of a shape as shown in Figure 3.3. There, we have $|V_H| = 7$ and a branching factor $B = 2$. Constructing all possible partial binary trees with seven vertices and merging them then leads to a complete binary tree of height four. Recall that vertices of data structure grammars are required to have either all or none or their possible outgoing edges. Therefore, it is not possible to remove the leaves on the upper levels and append them at the bottom to obtain a combined tree of height seven. The number of vertices of the combined tree can be calculated by adding up the number of vertices on ever level i. This results in the geometric series, which can be written as follows [1].

$$\sum_{i=0}^{n-1} B^i = \frac{B^n - 1}{B - 1} \tag{3.9}$$

The variable n is the height the combined tree, B is the branching factor. The height of the tree is in $\mathcal{O}(|V_H|/B)$, as there are B vertices on every level below the root. Plugging that into (3.9), we get

$$\frac{B^{\frac{|V_H|}{B}} - 1}{B - 1} \tag{3.10}$$

as an approximation for the number of vertices in the combined tree. Clearly, the following holds for all $B > 1$.

$$\frac{B^{\frac{|V_H|}{B}} - 1}{B - 1} \leqslant B^{\frac{|V_H|}{B}} - 1 \tag{3.11}$$

For better legibility, we write now x instead of B and n instead of $|V_H|$. The function $f(x) = x^{\frac{n}{x}} - 1$ is defined for $x \in (0, \infty)$. Calculating the first derivative with respect to x yields the following.

$$
\begin{aligned}
\frac{d}{dx}\left(x^{\frac{n}{x}}-1\right) &= \frac{d}{dx}\left(e^{\ln(x)\frac{n}{x}}-1\right)\\
&= \left(\frac{d}{dx}\ln(x)\frac{n}{x}\right)e^{\ln(x)\frac{n}{x}}\\
&= \left(\left(\frac{d}{dx}\ln(x)\right)\frac{n}{x}+\ln(x)\left(\frac{d}{dx}\frac{n}{x}\right)\right)x^{\frac{n}{x}}\\
&= \left(\frac{n}{x^2}-\ln(x)\frac{n}{x^2}\right)x^{\frac{n}{x}}\\
&= (1-\ln(x))\,nx^{\frac{n}{x}-2}
\end{aligned}
$$

Now, we set the result to zero and solve to x.

$$
\begin{aligned}
& & \frac{d}{dx}\left(x^{\frac{n}{x}}-1\right) &= 0\\
&\Leftrightarrow & (1-\ln(x))\,nx^{\frac{n}{x}-2} &= 0\\
&\Leftrightarrow & \ln(x)nx^{\frac{n}{x}-2} &= nx^{\frac{n}{x}-2}\\
&\Leftrightarrow & \ln(x) &= 1\\
&\Leftrightarrow & x &= e
\end{aligned}
$$

Since $e^{\frac{n}{e}}-1$ is larger than zero for all $n>0$ and equal to zero for $n=0$, and it holds that

$$\lim_{x\to\infty} x^{\frac{n}{x}}-1=0 \qquad \lim_{x\to 0} x^{\frac{n}{x}}-1=-1$$

$x^{\frac{n}{x}}-1$ has a global maximum at $x=e$. Considering the initial meaning of x and n, that means that, using the approximation of (3.11), the average branching factor yielding the largest combined tree is $B=e$. Thus, the size of that tree is in

$$\mathcal{O}\left(e^{\frac{|V_H|}{e}}\right)=\mathcal{O}\left(\left(e^{\frac{1}{e}}\right)^{|V_H|}\right)=\mathcal{O}\left(\sqrt[e]{e}^{|V_H|}\right)\approx\mathcal{O}\left(1.44466786^{|V_H|}\right)$$

which is independent of the number of rules as well as the branching factor. It follows that this is another upper bound for the longest run of $\mathcal{A}$. ∎

Theorem 3.5.6 (Time Complexity of detectEmbeddings(G, A))
Let $G\in HC_{\Gamma_N}$, $A=\{a_1,\ldots,a_b\}\subseteq Emb_{\Gamma_N}$, $(X\to H,i)\in A$. The time complexity of detectEmbeddings(G, A) is $\mathcal{O}(|V_G|\cdot|A|\cdot|V_H|^2\cdot B_H\cdot|ext_H|)$.

Proof: The body of the outermost loop is executed $|V_G|$ times. For each step of the automaton, $\lambda(q_{curr})(G,v)$ has to be evaluated, which is in $\mathcal{O}(|V_H|)$ as every time, one or two paths are evaluated. Those paths were obtained from one H and therefore have a maximum length of $|V_H|$. According to Theorem 3.5.5, the longest run of $\mathcal{A}$ is in $\mathcal{O}(|A|\cdot|Q'|)$ where Q' is the number of states of some $\mathcal{A}'=EDA(a)$ with $a\in A$. Using the approximation for the number of states, we get $\mathcal{O}(|A|\cdot|V_H|\cdot B_H\cdot|ext_H|)$. The second inner loop is in $\mathcal{O}(|A|)$. This results in $\mathcal{O}(|V_G|\cdot(|A|\cdot|V_H|\cdot B_H\cdot|ext_H|\cdot|V_H|+|A|))=\mathcal{O}(|V_G|\cdot|A|\cdot|V_H|^2\cdot B_H\cdot|ext_H|)$. ∎

With $\mathcal{O}(|A|\cdot|V_H|^2\cdot B_H\cdot|ext_H|\cdot|\Omega|)$, the construction of embedding detection automata is computationally about as expensive as the execution with $\mathcal{O}(|V_G|\cdot|A|\cdot|V_H|^2\cdot B_H\cdot|ext_H|)$.

Function detectEmbeddings($G \in HC_{\Gamma_N}, A \subseteq Emb_{\Gamma_N}$)

Input: $G \in HC_{\Gamma_N}$, embedding detection automaton $\mathcal{A}$ for a set of embeddings $A = \{a_1, \ldots, a_b\} \subseteq Emb_{\Gamma_N}$
Output: $B_v \subseteq A$ for all $v \in V_G$

```
forall v ∈ V_G do
    q_curr := q_0
    while λ(q_curr) ≠ end do
        q_curr := δ(q_curr, λ(q_curr)(G, v))

    q_curr = (q_1, ..., q_b)
    B_v := ∅
    forall k ∈ [1, b] do
        if q_k ∈ F_k then
            B_v := B_v ∪ {q_k}
```

Using *Juggrnaut* to analyze a program, the automaton has to be constructed just once, and is then used hundreds or thousands of times, in each state of the program where abstraction is applied. Grammar rules usually have not more than ten vertices. The size of a heap graph typically is between 50 and 100 vertices, which is achieved by constantly applying abstraction. The number of rules heavily depends on the complexity of the data structures. For trees with linked leafs, there are about 40 rules. Note that it is possible to reduce the complexity of the execution to $\mathcal{O}(|V_G| \cdot |A| \cdot |V_H| \cdot B_H \cdot |ext_H|)$ by using a stack to evaluate paths on H, such that it is not necessary to evaluate every path from the beginning in every state.

Even though $\mathcal{O}\left(\sqrt[e]{e}^{|V_H|}\right)$ is exponential, it is quite often a more precise upper bound than $\mathcal{O}\left(|A| \cdot |V_H| \cdot B_H \cdot |ext_H|\right)$, as $|A|$, the number of rules, is typically large compared to the size of rules $|V_H|$.

3.6 Example

To give an impression how detecting embeddings works when using an embedding detection automaton, we provide a small example in this section. Consider Figure 3.4 for two grammar rules $L \to H_1$ and $L \to H_2$ for a doubly-linked list. Additionally to the usual notation, the name of every vertex is given below. Let $\Sigma_N = \{n, p, L, \text{list}, \text{head}, \text{tail}\}$, $\Lambda = \{\perp, e, l\}$, $pt(e) = np$, $ts(L) = e_0eee_0$. In H_1, every other vertex is reachable from $v_2 = ext_{H_1}(2)$ and $v_3 = ext_{H_1}(3)$. Thus, is would be possible to construct an automaton either for a $(L \to H_1, 2)$-or $(L \to H_1, 3)$-embedding. In H_2, $(L \to H_2, 2)$ and $(L \to H_2, 3)$ are valid. In this example, we choose $(L \to H_1, 3)$ and $(L \to H_2, 3)$ to show how the resulting EDA looks like when both embeddings are to some extent similar.

Note that the majority of intermediate steps of the construction will be left out, as they do not contribute to the understanding of the resulting automata.

In a first step, a linear EDA $\mathcal{A}_1$ for $(L \to H_1, 3)$ will be constructed. The largest degree in H_1

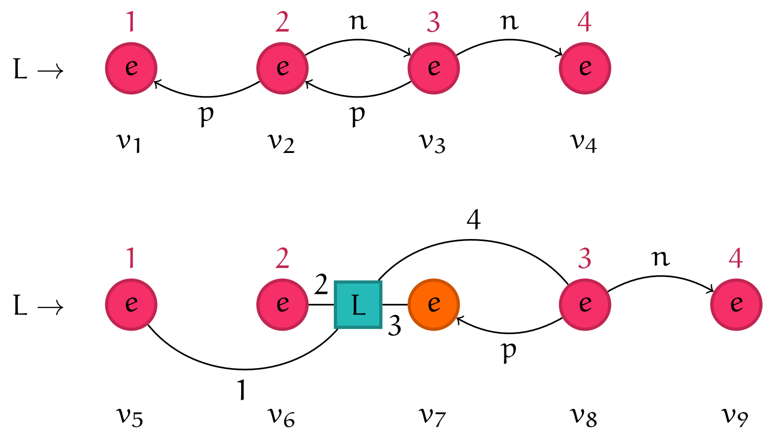

Figure 3.4: Grammar Rules $L \to H_1$ (top) and $L \to H_2$ (bottom).

(and H_2) is 3. Applying the algorithm, we get $\Pi_{DFS} = \{\varepsilon, n, p, pn, pp\}$, which is already sorted according to the path order for the sake of simplicity. We further obtain $s_{H_1} = v_3v_4v_2v_3v_1$ and $Dejavu_{H_1} = \{(1,4)\}$. The vertex that is visited twice is v_3, as it is the starting point of the depth-first search and reached again via the path pn. Since v_2 and v_3 are the only external vertices with outgoing tentacles, we get $s'_{H_1} = v_3v_2$ and the corresponding set of paths $\Pi'_{DFS} = \{\varepsilon, p\}$. After the construction of all states, the automaton as shown on the left side of Figure 3.6 is obtained. In the interest of visual clarity, the sink state and all transitions to it are not shown and we write $\langle\pi\rangle$ instead of $\langle\pi\rangle_{(G,v)}$. States are labeled with their λ-function.

The construction of the second linear EDA, $\mathcal{A}_2$, for a $(L \to H_2, 3)$-embedding, yields $\Pi_{DFS} = \{\varepsilon, n, p, p(L,1), p(L,2), p(L,4)\}$, $s_{H_2} = v_8v_9v_7v_5v_6v_8$ and $Dejavu_H = \{(1,6)\}$. As both rules have the same nonterminal on the left-hand side, external nodes with outgoing tentacles are again $ext_{H_2}(2)$ and $ext_{H_2}(3)$. This time, however, all outgoing tentacles of $ext_{H_2}(2)$ are abstracted inside a $(L,2)$-tentacle, which is possible because $ts(L)(2) = e \in \Lambda$. Thus, we get $s'_{H_2} = v_8v_6$ and $\Pi'_{DFS} = \{\varepsilon, p(L,2)\}$. The resulting automaton is depicted on the right side of Figure 3.6. The result of computing the union $\mathcal{A} = \mathcal{A}_1 \times \mathcal{A}_2$ is shown in Figure 3.7.

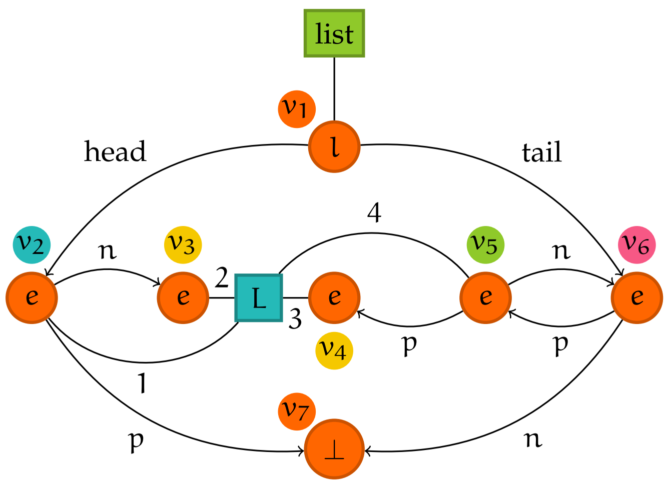

Figure 3.5: Heap Configuration G.

It is now possible to use $\mathcal{A}$ to find all $(L \to H_1, 3)$- and $(L \to H_2, 3)$-embeddings in G, which is shown in Figure 3.5. Next to each vertex, its name is denoted inside a circle. The colors of these circles correspond to the colors of the respective runs on $\mathcal{A}$ in Figure 3.7, which are indicated by lines. With input (G, v_1) and (G, v_7), a run ends in the sink immediately after the first state, as $\text{type}(\langle\varepsilon\rangle) \neq e$. In case of (G, v_3) and (G, v_4), it is $\deg_{\leqslant 3}(\langle\varepsilon\rangle) = 2$, such that there can not be an embedding of either $(L \to H_1, 3)$ or $(L \to H_2, 3)$. For input (G, v_2), we get $\text{ot}(\langle n\rangle) = (L, 2) \in \Omega_{\text{ot}}^{e}$, but $\text{type}(\langle p\rangle) = \bot \neq e$. (G, v_5) and (G, v_6) lead to accepting runs. In order to find out which embedding was found in each case, one needs to examine the last states of those runs. With input (G, v_5), that is $(q_{\text{sink}}^1, q_2) \in Q_1 \times Q_2$ and furthermore $q_2 \in F_2$. Thus, as $\mathcal{A}_2$ was constructed to detect $(L \to H_2, 3)$-embeddings, there is such an embedding at v_5. Analogously, we can infer that there is an $(L \to H_1, 3)$-embedding at v_6.

$$
\begin{array}{c}
\downarrow\\
\mathrm{type}(\langle\varepsilon\rangle)\\
\Big\downarrow e\\
\mathrm{deg}_{\leqslant 3}(\langle\varepsilon\rangle)\\
\Big\downarrow 3,*\\
\mathrm{ot}(\langle\varepsilon\rangle)\\
\Big\downarrow \{(n,1),(p,1)\}\\
\mathrm{type}(\langle n\rangle)\\
\Big\downarrow e,\bot\\
\mathrm{deg}_{\leqslant 3}(\langle n\rangle)\\
\Big\downarrow 1,2,3,*\\
\mathrm{ot}(\langle n\rangle)\\
\Big\downarrow \Omega^{e}_{\mathrm{ot}}\cup\{\emptyset\}\\
\mathrm{type}(\langle p\rangle)\\
\Big\downarrow e\\
\mathrm{deg}_{\leqslant 3}(\langle p\rangle)\\
\Big\downarrow 3,*\\
\mathrm{ot}(\langle p\rangle)\\
\Big\downarrow \{(n,1),(p,1)\}\\
\langle p\rangle=\langle\varepsilon\rangle\\
\Big\downarrow \mathrm{FALSE}\\
\langle pn\rangle=\langle\varepsilon\rangle\\
\Big\downarrow \mathrm{TRUE}\\
\mathrm{type}(\langle pp\rangle)\\
\Big\downarrow e,\bot\\
\mathrm{deg}_{\leqslant 3}(\langle pp\rangle\\
\Big\downarrow 1,2,3,*\\
\mathrm{ot}(\langle pp\rangle)\\
\Big\downarrow \Omega^{e}_{\mathrm{ot}}\cup\{\emptyset\}\\
\boxed{\mathrm{end}}
\end{array}
\qquad\qquad
\begin{array}{c}
\downarrow\\
\mathrm{type}(\langle\varepsilon\rangle)\\
\Big\downarrow e\\
\mathrm{deg}_{\leqslant 3}(\langle\varepsilon\rangle)\\
\Big\downarrow 3,*\\
\mathrm{ot}(\langle\varepsilon\rangle)\\
\Big\downarrow \{(n,1),(p,1)\}\\
\mathrm{type}(\langle n\rangle)\\
\Big\downarrow e,\bot\\
\mathrm{deg}_{\leqslant 3}(\langle n\rangle)\\
\Big\downarrow 1,2,3,*\\
\mathrm{ot}(\langle n\rangle)\\
\Big\downarrow \Omega^{e}_{\mathrm{ot}}\cup\{\emptyset\}\\
\mathrm{type}(\langle p\rangle)\\
\Big\downarrow e\\
\mathrm{deg}_{\leqslant 3}(\langle p\rangle)\\
\Big\downarrow 2\\
\mathrm{ot}(\langle p\rangle)\\
\Big\downarrow \{(L,3)\}\\
\mathrm{type}(\langle p(L,1)\rangle\\
\Big\downarrow e,\bot\\
\mathrm{deg}_{\leqslant 3}(\langle p(L,1)\rangle)\\
\Big\downarrow 1,2,3,*\\
\mathrm{ot}(\langle p(L,1)\rangle)\\
\Big\downarrow \Omega^{e}_{\mathrm{ot}}\cup\{\emptyset\}\\
\mathrm{type}(\langle p(L,2)\rangle\\
\Big\downarrow e\\
\mathrm{deg}_{\leqslant 3}(\langle p(L,2)\rangle)\\
\Big\downarrow 1,2,3,*\\
\mathrm{ot}(\langle p(L,2)\rangle)\\
\Big\downarrow \{(L,2)\}\\
\langle p(L,2)\rangle=\langle\varepsilon\rangle\\
\Big\downarrow \mathrm{FALSE}\\
\langle p(L,4)\rangle=\langle\varepsilon\rangle\\
\Big\downarrow \mathrm{TRUE}\\
\boxed{\mathrm{end}}
\end{array}
$$

Figure 3.6: Linear EDA $\mathcal{A}_1$ (left) and $\mathcal{A}_2$ (right).

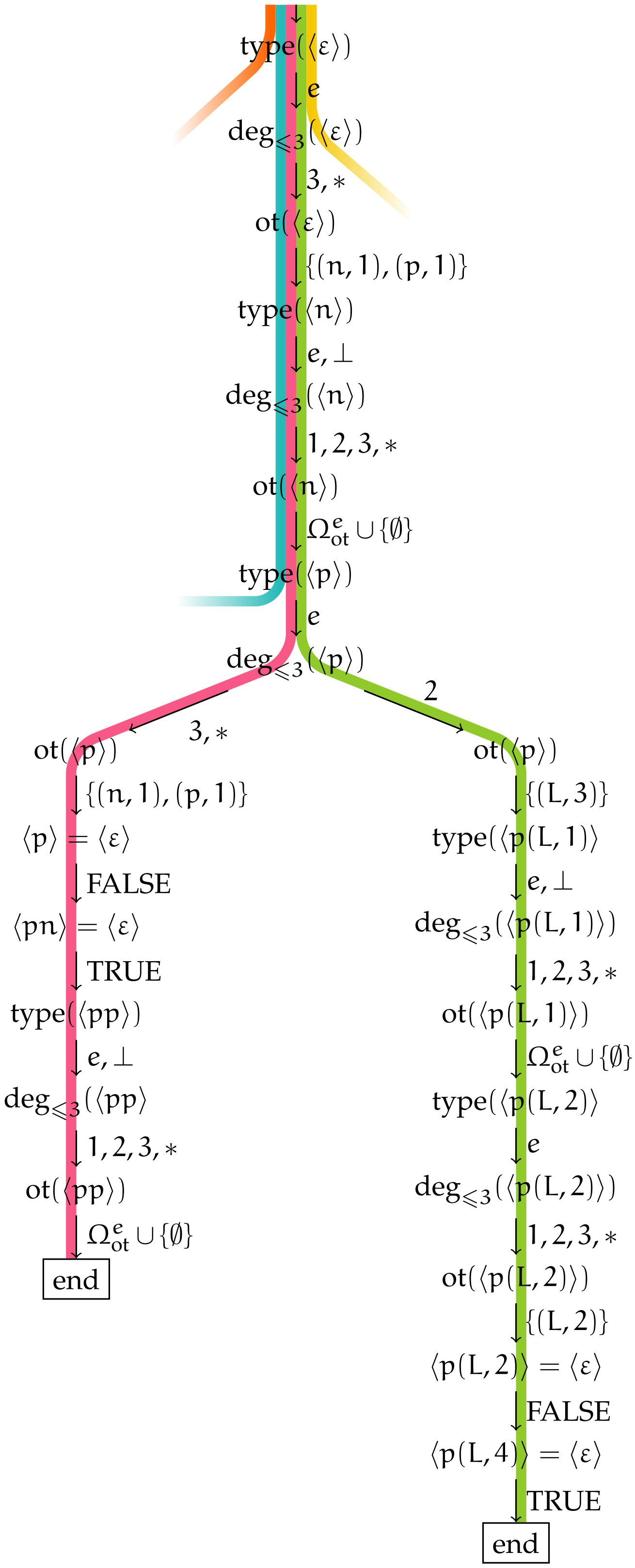

Figure 3.7: EDA $\mathcal{A}$ with All Runs on G.

4 NP-Completeness of Related Problems

Finding an embedding of a rulegraph in a hypergraph can be described as finding a subgraph isomorphism in a hypergraph. For common graphs, where edges connect exactly to vertices, this problem is known to be NP-complete, as proven by [3]. In this section, we are going to prove that this also holds for hypergraphs in the most general case and even for the more restricted heap configurations.

4.1 Subgraph Isomorphism on Hypergraphs

The proof for hypergraphs consists of two parts. It has to be shown that the problem is in NP and therefore decidable, whereafter we are going to show that it is furthermore NP-complete by reducing it to CLIQUE.

To begin, subgraph isomorphism needs to be defined for hypergraphs.

Definition 4.1.1 (Subgraph Isomorphism on Hypergraphs)
Let $G, H \in HG_{\Gamma_N}$. H is a *subgraph* of G if and only if G' is isomorphic to H with $V_{G'} \subseteq V_G$ and $E_{G'} \subseteq E_G$. ■

Lemma 4.1.1
Subgraph isomorphism on hypergraphs is in NP.

Proof: Let $H, G, G' \in HG_{\Gamma_N}$, $V_{G'} \subseteq V_G$ and $E_{G'} \subseteq E_G$. Consider a non-deterministic algorithm which is able to guess two bijective functions $f_V : V_H \to V_{G'}$ and $f_E : E_H \to E_{G'}$. It can be verified in polynomial time if H is isomorphic to G' by checking for all $v \in V_H$ and $e \in E_H$ if the requirements of Definition 2.1.3 are satisfied.
If $\text{type}_H(v) = \text{type}_{G'}(f_V(v))$ and $\text{ext}_H(i) = f_V(\text{ext}_{G'})$ hold can be computed in $\mathcal{O}(|V_H|)$, $\text{lab}_H(e) = \text{lab}_{G'}(f_E(e))$ in $\mathcal{O}(|E_H|)$. $\text{att}_G(f_E(e)) = f_V(\text{att}_{G'}(e))\, \forall e \in E_{G'}$ takes $\mathcal{O}(|V| \cdot |E_H|)$. Thus, subgraph isomorphism can be verified in $\mathcal{O}(|V_H| + |E_H| + |V_H| \cdot |E_H|)$. ■

Definition 4.1.2 (CLIQUE [3])
CLIQUE is the question whether there is for some graph $G = (V, E)$ and a positive integer $k \leqslant |V|$ a subset $V' \subseteq V$ with $|V'| \geqslant k$, such that every pair of vertices in V' is connected by an edge. In other words, it asks for a subgraph of G which is a complete graph with at least k vertices. ■

Theorem 4.1.1
Subgraph isomorphism on hypergraphs is NP-complete.

Proof: We will reduce subgraph isomorphism to CLIQUE, which is well known to be NP-complete [3]. In a first step, we show how an equivalent hypergraph can be constructed for every common undirected graph $G = (V_G, E_G)$. As CLIQUE is defined for graphs without

labels and annotations, we assume $|\Sigma| = |\Lambda| = 1$. An equivalent hypergraph $H \in HG_\Gamma$ can be constructed as follows:
Let $V_G = V_H$ and $ext_H = \varepsilon$. For all $(v_x, v_y) \in E_G$, add e_{xy} and e_{yx} to E_H, with $att_H(e_{xy}) = v_x v_y$ and $att_H(e_{yx}) = v_y v_x$. Since we assumed $|\Sigma| = |\Lambda| = 1$, lab and type map every vertex or edge to the same label or annotation, respectively.
As this construction method works for every graph, it works in particular for graphs G which are complete, that is for all $v_1, v_2 \in V$, there is an edge $(v_1, v_2) \in E$.
Thus, as hypergraphs contain common graphs as a special case, for every instance of CLIQUE it is possible to construct an equivalent instance of subgraph isomorphism on hypergraphs. Since CLIQUE is NP-complete, subgraph isomorphism on hypergraphs is NP-complete, too. ∎

4.2 Embeddings in Heap Configurations

First of all, it is useful to formulate a new NP-complete problem.

Definition 4.2.1 (EMBED)
EMBED is the question whether there is for some $X \to H \in DSG_{\Gamma_N}$ and $G \in HC_{\Gamma_N}$ an embedding of H in G. ∎

Proving that EMBED is NP-complete is very similar to proving CLIQUE on hypergraphs. Solely a more elaborate construction of an equivalent graph is necessary to circumvent the restriction that no two outgoing edges of one vertex are allowed to have the same label. In order to better illustrate the limitations of embedding detection automata, we provide an alternative definition for EMBED and prove that one.

Definition 4.2.2 (EMBED*)
Let $X \to P \in DSG_{\Gamma_N}$. As DSGs are not allowed to contain vertices which cannot be reached from any external vertex, there is a set $S \subseteq [ext_P]$, such that starting at some $s_i = ext_P(i) \in S$ and traversing edges entering at all tentacles (X, i) with $ts(X)(i) \in \Lambda_0$ and leaving at all other tentacles, every $v \in V_P$ can be reached. Let $P_i \in HG_{\Gamma_N}$, with $i \in [1, rk(X)]$, be the graph which consists of all vertices which can be reached from $ext_P(i)$ and all edges connecting them.
Let $H \in HC_{\Gamma_N}$ be a heap configuration. Searching for all P_i then yields for each P_i a (possibly empty) set $M_i \subseteq V_P$ of matching embeddings, which are unambiguously identified by that vertex which, inside the embedding, is the external vertex from which the search started.
EMBED* is the question whether for all $s_i \in S$, there is one $m_i \in M_i$, such that all m_i and all vertices which are reachable from those and all edges connecting them together make up an embedding of P in H. ∎

Lemma 4.2.1
EMBED* is in NP.

Proof: A non-deterministic algorithm guesses subsets $V_{G'} \subseteq V_G$, $E_{G'} \subseteq E_G$ and two bijective functions $f_V : V_P \to V_{G'}$ and $f_E : E_P \to E_{G'}$, such that for each $s_i \in S$, there is one $m_i \in M_i$, such that $f_V(m_i) \in V_G$.
It can be verified in polynomial time if those functions describe an embedding of H in G by checking for all $v \in V_H$ and $e \in E_H$ if the requirements of Definition 2.6.3 are satisfied.

(2.7) takes $\mathcal{O}(|E_H|)$, (2.8) and (2.9) takes $\mathcal{O}(|V_H|)$. (2.10) can be verified in $\mathcal{O}(|E_H| \cdot \max\{rk(e_H) \mid e_H \in E_H\}) = \mathcal{O}(|E_H|)$. (2.11) is quadratic in the number of vertices of H, that means $\mathcal{O}(|V_H|^2)$. (2.12) is quadratic in the rank of X, which is bounded by the number of vertices of H, so we have again $\mathcal{O}(|V_H|^2)$. Finally, (2.13) takes $\mathcal{O}(|E_G| \cdot \max\{rk(e_G) \mid e_G \in E_G\}) = \mathcal{O}(|E_G|)$. Thus, the overall complexity is $\mathcal{O}(|E_H| + |V_H| + |V_H|^2 + |E_G|)$.

■

Definition 4.2.3 (HAMILTONIAN PATH [3])
HAMILTONIAN PATH is the question whether some directed graph $G = (V, E)$ contains a hamiltonian path, that is, an ordering $v_1 \dots v_n$ of the vertices of G, where $n = |V|$, such that $(v_i, v_{i+1}) \in E$ for all $i \in [1, n-1]$. ■

Theorem 4.2.1
EMBED* is NP-complete.

Proof: We will reduce EMBED* to HAMILTONIAN PATH on directed graphs, which is NP-complete as proven by [3]. This is stated as the question whether it is possible, for a directed graph $G = (V_G, A_G)$, to find and ordering $v_1 v_2 \dots v_n$ of the vertices of G, with $n = |V_G|$, such that $(v_i, v_{i+1}) \in A_G$ for all $i \in [1, n-1]$. In other words, we are looking for a path in G which visits every vertex exactly once.
For every instance of HAMILTONIAN PATH, it is possible to construct a heap configuration $H \in HC_{\Gamma_N}$ as follows: We assume that $Sel_{\Sigma_N} = \{a, b\}$ and $\Lambda = \{u, w\}$ with $pt(u) = ab$, $pt(w) = \varepsilon$ and $ts(a) = ts(b) = uw_0$. Let $V_G = V_H$ and $\forall v \in V_H : type(v) = w$. For all $(v_x, v_y) \in A_G$, add $v_{x,y}$ to V_H, with $type(v_{x,y}) = u$. Then, add two edges $e_{x,x}$ and $e'_{y,y}$ to E_H with $lab_H(e_{x,x}) = a$, $lab_H(e'_{y,y}) = b$ and $att_H(e_{x,x}) = v_{x,y} v_x$, $att_H(e'_{y,y}) = v_{x,y} v_y$.
Additionally, we need a DSG_{Γ_N} containing one rule $X \to P$. This can easily be done by taking a directed graph $Q = (V_Q, A_Q)$ with $|V_Q| = n$ and $(v_i, v_{i+1}) \in A_G$ for all $i \in [1, n-1]$ and transforming it to a heap configuration $P \in HC_{\Gamma_N}$ as described above. Furthermore, we require $|ext_P| = |V_P|$ and $ts(X)(k) = w$ for all $ext_P(k) = v_i$ and $ts(X)(k) = u$ for all $ext_P(k) = v_{i,i+1}$ with $k \in [1, 2n-1], i \in [1, n]$. P is depicted in Figure 4.1.

Figure 4.1: Graph P, Obtained by Transforming a Directed Graph in an Equivalent HC.

Obviously, there is no external vertex in P from which every other vertex can be reached. Hence, we have to decompose P into graphs with that property and search for those, which is possible in polynomial time with an embedding detection automaton. It is sufficient to choose S such that it only contains vertices with type u, as all other vertices can be reached from those. All P_i are then isomorphic. One is shown in Figure 4.2.

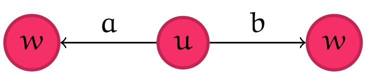

Figure 4.2: A Subgraph P_i of P, Containing All Vertices Reachable from one Vertex with Type u.

Those P_i correspond to one directed edge of Q or G and searching for those in H yields a M_i which contains every vertex $v \in V_H$ with $\text{type}(v) = u$, which corresponds to every edge of G. As all P_i are isomorphic, all M_i are equal, too.
Since we required $\text{ts}(X)(k) = w$ and not $\text{ts}(X)(k) = w_0$ for all vertices representing vertices in the original graph, it is not possible that two of those vertices with type w are mapped to the same vertex in H, as this would be a violation of Definition 2.6.3 for embeddings. From $|V_Q| = n$, it follows that there are as many vertices in P with type w as there are vertices of the same type in H. Thus, every embedding of P in H is equivalent to a hamiltonian path in G.
We now have all M_i and need to find one $m_i \in M_i$ each which together constitute an embedding of P. As all M_i are equal, this is the same as choosing $|S|$ elements of one M_i. This in turn is equivalent to asking for a subset of edges of G which form a hamiltonian path. Thus, as HAMILTONIAN PATH is NP-complete, it follows that EMBED* is NP-complete, too. ∎

It can be seen in this proof that the difference in the computational complexity of finding embeddings depends on whether there is one external vertex from which every other one can be reached.

5 Conclusion

In this thesis, we were able to propose a method for finding embeddings of hyperedge replacement grammar rules in hypergraphs in polynomial time. We used the requirement that all vertices of a rule have to be reachable from one external vertex and proved that otherwise, the problem is NP-complete. That restricition, however, is in practice not very limiting. If parts of a heap are not reachable at all, they are garbage and there is no use in abstracting those parts. For common data structures like lists and trees, grammar rules have that property. In other cases, it is possible to decompose a grammar rule into smaller parts that are reachable from one external vertex. Finally, those structures may appear rarely anyway, such that it would not significantly increase the size of the heap graph if they are not abstracted at all.

In a next step, the proposed embedding detection mechanism should be implemented. There is still room for improving the efficiency by using a stack for evaluating paths. Currently, for an input (G, v), every path is completely evaluated, starting at v. As those path, however, are ordered in the automaton as they would appear in a depth-first search, it would be possible to push the current vertex on a stack for each new path. Then, when proceeding to the next path, either only one path element has to be evaluated, starting at the vertex on top of the stack, or a number of vertices have to be removed from the stack. Further efficiency improvements could be achieved by different orderings of the selectors in the sequence of vertex type selectors. Those improvements are, however, highly dependent on the shape of rule graphs. Regarding *Juggrnaut*, it would be useful to find mechanisms to determine which embedding to choose for abstraction if more than one is found. In the current definition of heap configurations, inheritance is not taken into account. Therefore, it would be useful to introduce a type hierarchy. For more theoretical research, it would be interesting to further explore the properties of function automata, how their languages compare to regular and context free graph languages and how changes to the automata model, like introducing loops and changing the initial vertex of paths, change the language. Furthermore, the function automaton could be used to work on words instead of graphs. Then, its expressiveness could be compared to that of finite state machines and pushdown automata. Unlike common automata, the use of functions allows random access on the input word. Alternatively, functions could be used that modify the input.

Bibliography

[1] Ilja N. Bronstein, Konstantin A. Semendjajew, Günter Grosche, and Eberhard Zeidler. *Teubner-Taschenbuch der Mathematik*. Teubner-Taschenbuch der Mathematik. Teubner, 2003.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT electrical engineering and computer science series. MIT Press, 2001.

[3] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[4] Christina Jansen, Jonathan Heinen, Joost-Pieter Katoen, and Thomas Noll. A Local Greibach Normal Form for Hyperedge Replacement Grammars. In *Proc. of 5th Int. Conf. on Language and Automata Theory and Applications (LATA 2011)*, volume 6638 of LNCS. Springer, 2011.

[5] Grzegorz Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific, 1997.