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

Inferring Heap Abstraction Grammars

Alexander Dominik Weinert

1. Gutachter: Prof. Dr. Ir. Joost-Pieter Katoen
2. Gutachter: apl. Prof. Dr. Thomas Noll
Betreuer: Dipl.-Inform. 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 30.07.2012 Alexander Dominik Weinert

Abstract Heap abstraction grammars have successfully been used in the past to extend traditional model checking techniques to pointer programs. The downside of this approach is that it requires the user to specify such a grammar for every program that is to be verified. In this work we first give a brief introduction to hypergraphs, heap abstraction grammars and how they are used for modeling heap configurations. We then present an algorithm that observes some runs of a given program and produces a heap abstraction grammar that describes the behavior of this program on the heap in the most concise way, using techniques from the field of machine learning. We also take into account simple recursive data structures. Subsequently we present the results of some experiments with an implementation of this algorithm using the Juggrnaut-framework and conclude with a brief outlook on possible improvements of grammatical inference.

Contents

1 Introduction

1.1 Motivation

In recent years, software has become more and more complex. Even though it is by no means a sufficient measure of complexity, a simple look at the growth of the major operating systems in terms of lines of code shows this development towards rapidly increasing complexity: Whereas Microsoft Windows NT 3.1, which was released in 1993, consisted of about 5 million lines of code, one of its popular successors, Windows XP, which came out in 2001, raised this measure to 40 million lines [20].

Similar growth can be observed when considering the Linux kernel, which grew from about 122 000 lines of code in 1994 [25], to nearly ten million lines in its most recent version in 2012 [26]. The Debian operating system, which is built upon this kernel, displays an increase from 25 million lines of code in its 2.0 release in 1998 to about 282 million lines in 2009.

The usual approach for ensuring the correct behavior of these critical programs is testing, in which the program is given some input and it is checked, whether or not the output complies with the expectations. However, such a technique can only prove the existence of errors, but, in non-trivial cases, fails to demonstrate their absence, due to the number of possible inputs, which can quickly become infinite.

Thus, the field of *software verification*, which deals with mathematically proving the correct behavior of programs, has significantly gained importance. The general question asked here is the following: Given a system S and a specification ϕ, does S satisfy ϕ?

The first approach, the so-called *axiomatic approach*, tries to formalize the method of proving mathematical sentences and expands it to programs. The best known example of this approach is probably the Hoare calculus [16], which expands the works of Floyd [8] and provides a logical system for proving partial correctness.

Even though there are some partly automated tools that simplify this formal verification nowadays, one major drawback of this technique remains: There are some steps when constructing formal proofs, in which human intuition has to be used, i.e., for specifying loop invariants.

In the early eighties, another approach, specifically tailored to the verification of hardware and software systems was presented by Clarke, Emerson and Sifakis [3],[21]. This technique, called *model checking*, works by exploring all possible states a system can be in during the execution and checking whether a "bad state", i.e., a state violating the specification can be reached from a starting state.

While this approach can not work for all systems and all specifications in its traditional form, due to the infinite state-space it would have to explore, its major benefit is its possibility to be completely automated in cases where it does work.

The main drawback, however, is the underlying assumption: The system in question has to define a *finite* state-space. As soon as this is not the case, traditional model checking fails. In terms of software systems this means that such model checking techniques are not applicable to programs operating on unbounded space, i.e., the heap.

There are some approaches to providing a possibility to also describe heap-states finitely, most notably separation logic, which was first presented in [22] by John Reynolds, and the use of *heap abstraction grammars* (HAGs), which was first presented in [14] and forms the basis for the Juggrnaut-framework, which is presented in the same publication and used for verification of Java bytecode.

The goal of this work is to expand upon the latter approach. Its main idea is to model the heap at a given point in the execution of a program as a simple graph, with nodes representing objects on the heap and edges representing the pointers between them. It then employs a heap abstraction grammar as a finite representation of all states of the heap that can occur during a run of the program.

In its current state, it is necessary for the user to provide this heap abstraction grammar, which can be quite unintuitive in many cases. To address this flaw of the technique, the use of a learning algorithm for HAGs was proposed in [17], which takes a set of graphs as input (in this case the heap configurations observed during the run of the program) and produces an HAG which describes *at least* the graphs provided in the set. In this work, we present an alternativ algorithm for the same task, which produces different grammars as output.

The structure of the remainder of this work is as follows: In the second chapter, we will define the basic models employed in this work, namely hypergraphs and hyperedge abstraction grammars. The third chapter presents the requirements imposed on these models when using them for finitely describing all possible heap states of a program. In the fourth chapter we present an alternative approach to inferring so-called data structure grammars, with an evaluation following in chapter 5. The sixth chapter summarizes the results and gives a brief outlook on further work.

1.2 Basic Notation and Assumptions

We base our notation on [17], which in turn is based on [10].

Tuples For any tuple $A = (B, C)$, we let B_A and C_A denote the elements of A, unless A is clear from the context.

Sequences For any given set V we denote by V^* the set of sequences of symbols from V. For a given sequence $\alpha = a_1 \dots a_n$ we write $|\alpha| := n$ for the length of the sequence and $\alpha[i] = a_i$ for the i-th character in the sequence.

We also let $[\alpha] := \{a_1, \dots, a_n\}$ denote the set of all elements in α and define the $\in$-operator for sequences, such that $a \in \alpha :\Leftrightarrow a \in [\alpha]$.

Functions Let A, B and C be sets, where B is a subset of A (which we denote by $B \subseteq A$) and let $f : A \to C$ be a function mapping values of A to values from C.

We then denote by $f\big|_B : B \to C$ the restriction of f to B, i.e., the function that returns $f(x)$ for all x in B.

For any set A the identity function is designated by $id_A : A \to A$, i.e., $id_A(x) = x$ for all $x \in A$.

We also define $f : A^* \to C^*$ and $f : 2^{|A|} \to 2^{|C|}$ implicitely for sequences and sets by pointwise application.

For two functions $f : A \to B$ and $g : C \to D$, where A and C are disjoint sets, we write $f \cup g : A \cup C \to B \cup D$ for the function that is defined as

$$(f \cup g)(x) = \begin{cases} f(x) & , \text{ if } x \in A \\ g(x) & , \text{ if } x \in C \end{cases}.$$

Relations For any relation R we denote by R^* the *transitive closure* of R, i.e.,

$$(X, Y) \in R^* :\Leftrightarrow \exists Z_1, \dots, Z_n.\ X = Z_1 \wedge \bigwedge_{i=1}^{n-1} (Z_i, Z_{i+1}) \in R \wedge Z_n = Y$$

for some $n \in \mathbb{N}_{>0}$.

2 Hypergraphs and Grammars

In this chapter we introduce *Hyperedge Replacement Grammars* as a simple method for describing a set of graphs. Since these grammars adapt the concept of context-free string grammars to graphs, we have to define alphabets for this purpose in section 2.1 first and then expand the concept of graphs in section 2.2 to account for the nonterminal and terminal objects used in grammars.

Lastly, we show how to replace such a nonterminal object of a graph in section 2.3 and join all these concepts together in the definition of *Hyperedge Replacement Grammars* in section 2.4

2.1 Alphabets

Our first step in expanding the definition of a graph later on is to introduce labels for its edges. Since these labels come from a pre-defined set, the so-called alphabet, we will have a short look at how such an alphabet is structured in this work. First, we require that every symbol from the alphabet is either *terminal* or *nonterminal*, both of which have meanings similar to their use in string grammars.

We also demand that the alphabet is ranked, meaning that there is a ranking function that assigns a rank to every symbol, both terminal and nonterminal.

Definition 2.1.1 (Alphabet)**.** *An alphabet Σ is a tuple: $\Sigma := (N, T, rk)$, where N and T are finite, disjoint sets and $rk : N \cup T \to \mathbb{N}$. Furthermore, T is nonempty.*

We call the elements of T the terminal *symbols and the elements of N the* nonterminal *symbols. We let nonterminal symbols of any alphabet be designated by uppercase letters and strings and terminal symbols by lowercase ones.*

2.2 Hypergraphs

A hypergraph is a simple extension of labeled traditional graphs. Recall that standard graphs consist of *vertices* (or *nodes*) and *edges*, where every edge connects exactly two vertices. In labeled graphs, either the vertices or the edges are labeled with symbols from some alphabet Σ. In this work, we first consider labels for edges only and will introduce labels for vertices later on, when talking about heap abstraction in section 3.2.

Analogously to the traditional case, hypergraphs consist of vertices and labeled *hyper*edges, where a hyperedge is a generalization of the concept of traditional edges: In contrast to a traditional edge, a hyperedge can connect an arbitrary number of vertices.

Definition 2.2.1 (Hyperedge)**.** *A hyperedge is a construct that connects an arbitrary number of vertices in a hypergraph and is labelled with some symbol from an alphabet Σ.*

We call a hyperedge that is labelled with $a \in \Sigma$ an a-hyperedge*. If the label of a hyperedge is terminal, we call the hyperedge terminal as well. Analogously, we call hyperedges with a nonterminal label nonterminal hyperedges.*

Since we only deal with hyperedges in this work, we will not differentiate between edges and hyperedges.

Thus, a hyperedge basically consists of two parts: its label and references to the nodes it is connected to. We call the part that connects a hyperedge to one of the attached nodes a *tentacle*.

A graph that contains (labeled) hyperedges is called a *hypergraph*.

Definition 2.2.2 (Hypergraph)**.** *A hypergraph G over an alphabet Σ is a tuple: $G := (V, E, lab, att, ext)$, where the components are as follows:*

- *V is a set of vertices.*
- *E is a set of hyperedges.*
- *$lab : E \to \Sigma$ is a labeling function that assigns a label to every edge.*
- *$att : E \to V^*$ is an attachment function that assigns a sequence of attached nodes to every edge.*
- *$ext \in V^*$ is a sequence over V, in which each element appears at most once, denoting the external nodes of the hypergraph.*

The size of a hypergraph G is denoted by $|G|$ and defined as the sum of its number of vertices and edges: $|G| = |V_G| + |E_G|$.

We call the number of vertices connected by a specific hyperedge the rank *of that hyperedge and denote it by the function $rk : E \to \mathbb{N}_0$, i.e., for $e \in E : rk(e) := |att(e)|$. We furthermore demand that the rank of a hyperedge is equal to the rank of its label, i.e., $\forall e \in E.\ rk(e) = rk(lab(e))$ has to hold.*

The rank of a hypergraph G is defined as the number of external vertices: $rk(G) = |ext_G|$. We call a hypergraph in which all edges are labeled with terminal symbols a terminal *hypergraph.*

We denote the set of all hypergraphs over an alphabet Σ by $\mathcal{HG}_\Sigma$.

Note that the labeling function does neither have to be injective, nor surjective, which means that every symbol may be assigned to several edges and not all symbols have to be used.

Also note that the sequence of nodes attached to a hyperedge as well as the sequence of external nodes define an order, which enables us to enumerate the vertices.

Since we do not want to differentiate between hypergraphs of the same structure and labeling, which only differ in the identifiers of their vertices and edges, we define what it means for two graphs to be equal. For this, we first formalize this condition and call it *isomorphism*. The following definition is adapted from [2].

Definition 2.2.3 (Isomorphism and Identity of Hypergraphs)**.** *Let G and H be two hypergraphs over Σ. We say they are* isomorphic *if and only if there exist two bijective functions $iso_V : V_G \to V_H$ and $iso_E : E_G \to E_H$, such that*

$$\begin{aligned} &\forall e \in E_G \quad lab_G(e) = lab_H(iso_E(e)) \quad \wedge \\ &\forall e \in E_G \quad iso_V(att_G(e)) = att_H(iso_E(e)) \quad \wedge \\ &\qquad\qquad iso_V(ext_G) = ext_H \end{aligned}$$

holds.

We do not differentiate between isomorphic graphs, i.e., we say that two graphs are identical iff they are isomorphic.

In graphical representations of hypergraphs, we draw hyperedges as yellow boxes with numbers next to the tentacles denoting the order of the attached nodes. For simplification, when drawing hyperedges, we make one exception for terminal edges of rank 2. We draw these as normal directed edges, with the origin of the edge being the first node attached to the edge and the target of the edge the second one (cf. figure 2.1).

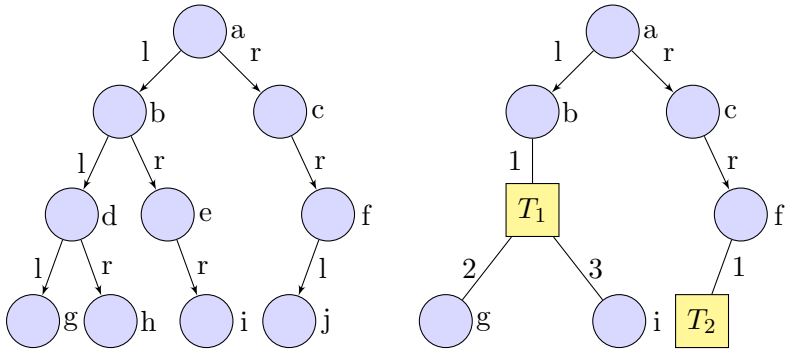
Figure 2.1: Simplified representation of hyperedges with rank 2

We draw normal nodes in a light blue shade and external nodes in a darker blue. We define their position in the sequence of external nodes by numbers in the external nodes. If we have to identify specific nodes or edges, we write their identifiers next to them.

Example 2.2.1 (Hypergraph of a Binary Tree)**.** *We start with choosing an alphabet $\Sigma = (N, T, rk)$ with $N \cap T = \emptyset$.*

In the first step we assume we know the complete tree, i.e., we have no edges which may have to be replaced later on. Thus we choose $N := \emptyset$.

The terminal symbols are those that we label the edges of our resulting graph with. Since we want to model a binary tree, we only have two kinds of edges: Those pointing to the right successor and those pointing to the left successor of a node. Thus we choose $T := \{l, r\}$. *Since both labels are used to label connections between two nodes, they are of rank 2:* $rk(l) = rk(r) = 2$.



(a) A concrete tree (b) A partly abstracted tree

Figure 2.2: A concrete and a partly abstracted tree

Figure 2.2(a) shows our representation of a tree with full knowledge of its layout.

Now assume that we do not have full knowledge of the tree. Let us say that for some reason we know of the existence of nodes a, b, c, f, g and i, but do not know about the existence of nodes d, e, h and j. However, we know that nodes b, g and i are connected in some way and there is something attached to node f as well.

We express this connection with two hyperedges. We choose to introduce new nonterminal symbols, since we want to have the possibility to include the unknown parts of the tree in the graph once we have discovered them. Thus we choose $N := \{T_1, T_2\}$ *to label these two new hyperedges and assign them the ranks 3 and 1, respectively.*

Figure 2.2(b) shows the graph that represents the situation above. The newly introduced T_1*-edge has three tentacles, which are attached to the nodes b, g and i, whereas the* T_2*-edge has only one tentacle, which is attached to node f.*

Note that we fixed an arbitrary order on the tentacles of the T_1*-edge in this case. We will see the importance of this order in section 2.3.*

Since the definition of hypergraphs makes no restriction to "useful" hypergraphs, we will constrain ourselves to only a subset of hypergraphs, i.e., graphs without isolated nodes.

Definition 2.2.4 (Freedom of Isolation)**.** *We call* G free of isolation *iff there is at least*

one edge connected to each node:

$$G \in \mathcal{HG}_\Sigma \text{ is free of isolation} :\Leftrightarrow \forall v \in V_G.\ \exists e \in E_G.\ v \in att(e)$$

We call a graph that is free of isolation wellformed.

In the remainder of this work, we will assume all hypergraphs to be wellformed.

We have introduced labeled hypergraphs and defined the alphabets used for labeling. Afterwards we have seen that we can "hide" certain, unknown parts of a graph with hyperedges. This technique will become especially useful later on, when we use hypergraphs for describing possible states of a heap.

For now, it suffices to know that we call this method of hiding certain parts of a graph *abstraction*. The part of the graph that is now hidden is called the *abstracted* part of the graph.

2.3 Hyperedge Substitution

Now, the next step is to undo this abstraction, i.e., replace a hyperedge with a hypergraph. We call this step *concretizing* the hyperedge in question.

We introduce the *replace*-operation that does exactly that. At this point we use the external nodes of a hypergraph for the first time, since they function as attachment points for the "context" of the edge that is to be replaced.

Definition 2.3.1 (replace operation)**.** *Let $G := (V_G, E_G, lab_G, att_G, ext_G)$ and $S := (V_S, E_S, lab_S, att_S, ext_S)$ be hypergraphs, $e_R \in E_G$ a hyperedge from G and let $rk(e_R) = rk(S)$.*

We then define $replace(G, e_R, S) = G'$ as the result of replacing e_R by S. The vertices, edges, labels and external nodes of $G' := (V_{G'}, E_{G'}, lab_{G'}, att_{G'}, ext_{G'})$ are defined as follows:

$$\begin{array}{rclcrcl} V_{G'} & := & V_G \uplus (V_S \setminus [ext_S]) & & E_{G'} & := & (E_G \setminus \{e_R\}) \uplus E_S \\ lab_{G'} & := & (lab_G\big|_{E_G \setminus \{e_R\}}) \cup lab_S & & ext_{G'} & := & ext_G \end{array}$$

Since the attachment function is the most complicated part of the replace operation, we treat it separately and look at the target of each tentacle in detail:

$$att_{G'}(e)[i] := \begin{cases} att_G(e)[i] & , \text{ if } e \in E_G \\ att_S(e)[i] & , \text{ if } e \in E_S \wedge att_S(e)[i] \notin [ext_S] \\ att_G(e_R)[j] & , \text{ if } e \in E_S \wedge att_S(e)[i] = ext_S(j) \end{cases}$$

For reasons of brevity we use $G\,[S/e_r]$ as a shorthand notation for $replace(G, e_R, S)$. In this notation we also allow the simultaneous replacement of multiple edges, which we denote by $G\,[S_1/e_1, \ldots, S_n/e_n]$, where $rk(S_i) = rk(e_i)$ holds for all $1 \leq i \leq n$.

Note that this pointwise definition of $att_{G'}$ really covers all edges and their tentacles. Each edge of the resulting graph is either copied from G or from S. If it was copied from G, we can simply copy the attachments of its tentacles, since all nodes of G are present in G'. If it was copied from S and is not attached to any external nodes, we can leave it untouched as well, since only the external nodes of S are not present in G'. In the last case, it was part of S and the tentacle in question was attached to the j-th external node, which has now been removed. In this case, we simply attach it to the target of the j-th tentacle of the replaced edge.

One could say that the node attached to the j-th tentacle of e_S now takes over the role of the j-th external node of S. This also explains why we demanded that $rk(e_S) = |ext_S|$. If this was not the case, then we would not be able to map each external node to a different node attached to e_S uniquely. Thus, we would have to introduce either "dangling" tentacles or a decision rule for attaching these tentacles, both of which would introduce unnecessary complexity.

Even though the simultaneous replacement of multiple edges was not defined formally, it works as one would intuitively imagine. However, as the following lemma states, allowing parallel replacement does not simplify or complicate the issue at hand, since it functions merely like a shorthand notation for the serial replacement of multiple edges.

Lemma 2.3.1 (Serializability / Parallelizability)**.** *Let G be a hypergraph over Σ, $e_1, \ldots, e_n \in E_G$ hyperedges from G and let $H_1, \ldots, H_n \in \mathcal{HG}_\Sigma$ be hypergraphs with $rk(e_i) = rk(H_i)$ for all $1 \leq i \leq n$.*

Then it holds that

$$G\left[H_1/e_1, \ldots, H_n/e_n\right] = \left(\ldots\left(G\left[H_1/e_1\right]\right)\ldots\left[H_n/e_n\right]\right).$$

Also, even though the definition imposes an order on the hyperedges to be replaced, this order is completely arbitrary and may be changed without changing the resulting hypergraph:

Lemma 2.3.2 (Confluence)**.** *Let G be a hypergraph over Σ, $e_1, e_2 \in E_G$ hyperedges from G and let $H_1, H_2 \in \mathcal{HG}_\Sigma$ be hypergraphs with $rk(e_1) = rk(H_1)$ and $rk(e_2) = rk(H_2)$.*

Then it holds that

$$\left(G\left[H_1/e_1\right]\right)\left[H_2/e_2\right] = \left(G\left[H_2/e_2\right]\right)\left[H_1/e_1\right].$$

Finally, it does not matter if we first replace some edge with a hypergraph and then replace an edge of the new subgraph, or if we do the second replacement before inserting the resulting subgraph into the original hypergraph. This property is stated in the following lemma:

Lemma 2.3.3 (Associativity)**.** *Let G be a hypergraph over Σ, $e_1 \in E_G$ a hyperedge in G, $H_1 \in \mathcal{HG}_\Sigma$ a hypergraph with $rk(e_1) = rk(H_1)$ and let $e_2 \in E_{H_1}$ and $H_2 \in \mathcal{HG}_\Sigma$ be a hyperedge from H_1 and a hypergraph with $rk(e_2) = rk(H_2)$, respectively.*

Then it holds that

$$(G\,[H_1/e_1])\,[H_2/e_2] = G\,[H_1\,[H_2/e_2]\,/e_1]\,.$$

All these lemmata are given in [17]. Since the proofs follow directly from the definition of the replace operation, we do not present them here. The interested reader can find them in [5].

We have already used the notion of subgraphs without properly defining it. Since we will later use the fact that some graph is a subgraph of another graph, we define this property formally.

Definition 2.3.2 (Subgraph)**.** *Let G and H be hypergraphs. We say that H is a* subgraph *of G if and only if there exists some hypergraph G' that is isomorphic to H and that fulfills the following conditions:*

$$\begin{array}{rcl@{\qquad}rcl} E_{G'} & \subseteq & E_G & att_{G'} & = & att_G\big|_{E_{G'}} \\ att_G(E_{G'}) & \subseteq V_{G'} \subseteq & V_G & lab_{G'} & = & lab_G\big|_{E_{G'}} \end{array}$$

$$[ext_{G'}] \supseteq ([ext_G] \cap V_{G'})$$

$$[ext_{G'}] \supseteq \{v \in V_{G'} \mid \exists e \in E_G \setminus E_{G'}.\ \exists i \in \mathbb{N}_0.\ att_G(e)[i] = v\}$$

The first four conditions simply demand that the subgraph is indeed part of the original graph and that all tentacles of the subgraph are attached to a node in the subgraph. The fifth condition states that no external nodes may be mapped to internal nodes of the subgraph, while the last condition states that each node that has at least one adjacent edge which is not in the subgraph has to be external. However, there may be more external nodes.

If there was some edge attached to a nonexternal node of the subgraph, we would lose information about the attachment of this tentacle after the abstraction.

It was shown that the decision whether or not a given hypergraph is a subgraph of another graph is NP-complete in [2].

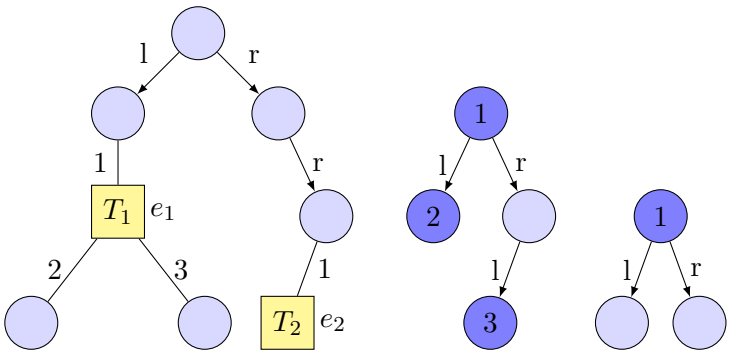
We have already stated that the process of replacing a subgraph by a single hyperedge is called *abstracting* that subgraph. Analogously we call the process of replacing a hyperedge by a hypergraph *concretizing* that hyperedge.

Let us consider an example of a concretization of a hyperedge.

Example 2.3.1 (Substitution of Hyperedges)**.** *In example 2.2.1 we have replaced subgraphs with a single hyperedge. We now want to revert that step, i.e., we want to replace the new hyperedges with some suitable hypergraphs.*

Consider figure 2.3(a), which depicts the partly abstracted tree from example 2.2.1. We replace the T_1- and T_2-hyperedges with the subtrees shown in figures 2.3(b) and 2.3(c).

We first substitute e_1 with $\mathcal{G}_1$, i.e., we compute the result of `replace`$(\mathcal{G}, e_1, \mathcal{G}_1) =: \mathcal{G}'$.

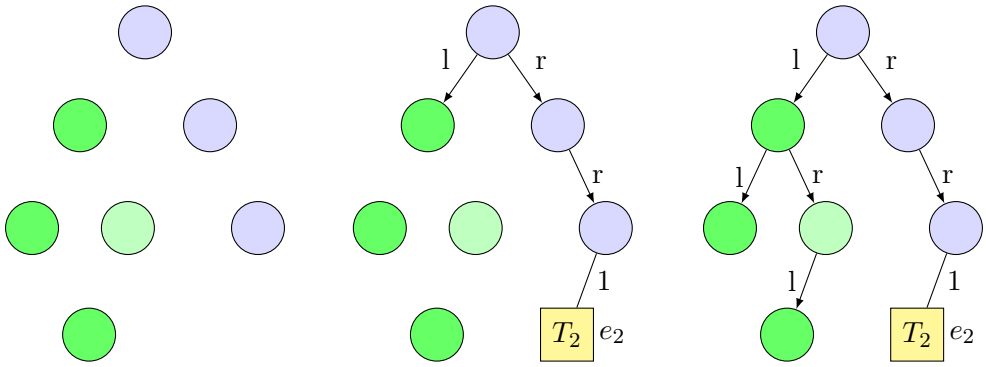


(a) $\mathcal{G}$ (b) $\mathcal{G}_1$ (c) $\mathcal{G}_2$

Figure 2.3: A partly abstracted tree and candidates for replacing the hyperedges

First we have to ensure that the preconditions hold. We see that $\mathcal{G}$ is a hypergraph that contains the edge e_1. $\mathcal{G}_1$ is a hypergraph as well and has the same rank as e_1 in $\mathcal{G}$, hence it is a fitting candidate for substitution.

The intermediate results of replacing e_1 with $\mathcal{G}_1$ are shown in figure 2.4.



(a) Nodes of $\mathcal{G}'$ (b) After copying edges from $\mathcal{G}$ (c) Final result

Figure 2.4: Intermediate steps during computation of `replace`$(\mathcal{G}, e_1, \mathcal{G}_1)$

Figure 2.4(a) shows only the nodes of $\mathcal{G}'$. As defined earlier, we unite the nodes of $\mathcal{G}$ with the internal nodes of $\mathcal{G}_1$. The newly introduced node is marked in light green; the nodes that e_1 was attached to are marked in dark green. They will be used as the connection points between $\mathcal{G}$ and $\mathcal{G}_1$.

In the next step, shown in figure 2.4(b), we copy all edges from $\mathcal{G}$.

The final figure 2.4(c) shows the complete result of `replace`$(\mathcal{G}, e_1, \mathcal{G}_1)$ after including the edges from $\mathcal{G}_1$, with those tentacles that were attached to an external node of $\mathcal{G}_1$ now

connected to the blue-colored nodes of $\mathcal{G}$.

We only show the result of $\texttt{replace}(\mathcal{G}', e_2, \mathcal{G}_2)$ *in figure 2.5 and leave the derivation of the individual steps as an exercise to the reader.*

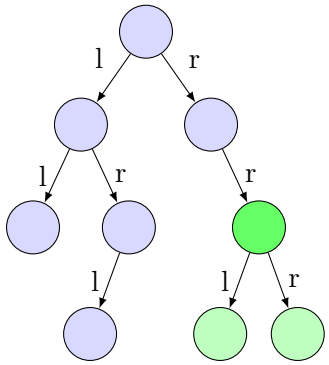


Figure 2.5: The result of $\texttt{replace}(\mathcal{G}', e_2, \mathcal{G}_2)$

2.4 Hyperedge Replacement Grammars

We have seen how one replaces a single hyperedge with a single hypergraph, both specifically chosen. A hyperedge replacement grammar systemizes these operations by providing a set of rules, specifying which edges may be replaced by which hypergraphs.

Definition 2.4.1 (Hyperedge Replacement Grammar)**.** *A Hyperedge Replacement Grammar* I *over an alphabet* Σ*, or* HRG *for short, is a tuple:* $I := (N, T, P, S)$*, where*

- N *is the set of nonterminal symbols of* Σ.
- T *is the set of terminal symbols of* Σ.
- P *is a set of* production rules *or simply* productions *of the form* $X \to H$*, where* X *is a nonterminal symbol,* H *is a hypergraph over* Σ *and* $rk(X) = rk(Ha)$ *holds for all productions.*
- $S \in HG_\Sigma$ *is the startgraph.*

Sometimes the hypergraph S *is also called the* axiom *of the grammar, e.g., in [10]. For a given production* $p := X \to H$*, we call* X *and* H *the* left *and* right *side of the production, respectively.*

We denote the set of all hyperedge replacement grammars over Σ *by* $\mathcal{HRG}_\Sigma$.

A hyperedge replacement grammar works similarly to a string grammar: We start with the axiom S and apply the productions from P, until we arrive at a hypergraph in which all edges are labeled with terminal symbols.

We introduce a *handle* for each nonterminal symbol, which allows us to use nonterminal symbols as axioms of hypergraph grammars as well as hypergraphs. The idea for this handle is to produce a graph with a single edge labeled with a nonterminal symbol and only as many vertices as the rank of the symbol demands.

Since we want to be able to replace an edge labeled with a nonterminal by the handle of another nonterminal, thus effectively exchanging the label on the edge, all the nodes of the handle are external.

Definition 2.4.2 (Handle)**.** *Let X be a nonterminal symbol, with $rk(X) = n$. An X-Handle, denoted by $X^\bullet$ is then defined as*

$$X^\bullet := (\{v_1, \ldots, v_n\}, \{e\}, (e \mapsto X), (e \mapsto v_1 \ldots v_n), v_1 \ldots v_n)$$

The application of a rule $X \to H$ to a graph G is defined as follows:

Definition 2.4.3 (Application of a Production)**.** *Let I be a hyperedge replacement grammar over Σ, let $p := X \to H$ be a production rule from this grammar, and let G be a hypergraph over Σ.*

We denote that it is possible to replace some edge labeled with X in G by H, resulting in G' by $G \overset{p}{\Rightarrow}_I G'$, or, more formally:

$$\begin{aligned} & G \overset{X \to H}{\Longrightarrow}_I G' \\ :\Leftrightarrow \quad & \exists e \in E_G. \, [lab(e) = X \wedge G' = \texttt{replace}(G, e, H)] \end{aligned}$$

We can also drop the explicit mention of the production used and simply write $G \Rightarrow_I H$:

$$G \Rightarrow_I H :\Leftrightarrow \exists p \in P_I. \left(G \overset{p}{\Rightarrow}_I H \right)$$

If the grammar is clear from the context, we may also write $G \Rightarrow H$.

If it is possible to derive H from G in some grammar, we simply write $G \Rightarrow^* H$:

$$\begin{aligned} & G \Rightarrow^* H \\ :\Leftrightarrow \quad & \exists n \in \mathbb{N}. \, \exists G_1, \ldots, G_n \in \mathcal{HG}_\Sigma. \, \left[G = G_1 \wedge \left(\bigwedge_{i=1}^{n-1} G_i \Rightarrow G_{i+1} \right) \wedge G_n = H \right] \end{aligned}$$

Note that this definition also includes the case in which G equals H, i.e., it is always possible to derive a graph from itself.

Since we pick e arbitrarily from E_G, there can be multiple hypergraphs H_1 and H_2 for a single rule p, with $G \overset{p}{\Rightarrow}_I H_1$ and $G \overset{p}{\Rightarrow}_I H_2$. Also, if there is no edge labeled with X in G, then there is no hypergraph H with $G \overset{p}{\Rightarrow}_I H$.

Note that the smallest n in the definition of $G \Rightarrow_I^* H$ is bounded by the difference between the sizes of both graphs, since productions can not remove nodes from a graph. Hence, the problem whether or not $G \Rightarrow_I^* H$ holds for a given grammar is decidable.

Repeated application of production rules yields a *derivation.*

Definition 2.4.4 (Derivation)**.** *Let G and H be hypergraphs over Σ. If $G \Rightarrow^* H$, we call the sequence of hypergraphs G_1 to G_n with $G = G_1 \Rightarrow \cdots \Rightarrow G_n = H$ the* derivation *of H from G. The derivation is said to have length $n - 1$.*

The single applications of productions $G_i \Rightarrow G_{i+1}$ are called steps of the derivation.

We call the set of all terminal hypergraphs that can be derived from the starting symbol S of a grammar the *language* of that grammar.

Definition 2.4.5 (Language of a Hyperedge Replacement Grammar)**.** *Let $I = (N, T, P, S)$ be a HRG and let G be a hypergraph. We denote the set of graphs derivable from G by $L(I, G)$:*

$$L(I, G) := \{H \in \mathcal{HG}_\Sigma \mid H \text{ is terminal } \wedge G \Rightarrow_I^* H\}$$

The language *of I is then denoted by $L(I)$ and defined as all graphs that can be derived from the axiom S_I:*

$$L(I) := L(I, S_I)$$

Example 2.4.1 (Hyperedge Replacement Grammar for Binary Trees)**.** *Our goal in this example is to construct a grammar I that describes the set of all nonempty binary trees.*

A binary tree consists of a node with two pointers that each point either to the left or right subtree of the node, or that are set to null. In our approach we model the two subtrees as a single nonterminal hyperedge labeled with S, which can be concretized to represent either only a left subtree, only a right subtree, both or neither of them. We also need the labels l and r to label the concrete pointers between nodes, just like in example 2.2.1

$$N := \{S\} \quad T := \{l,r\} \ ,$$

with

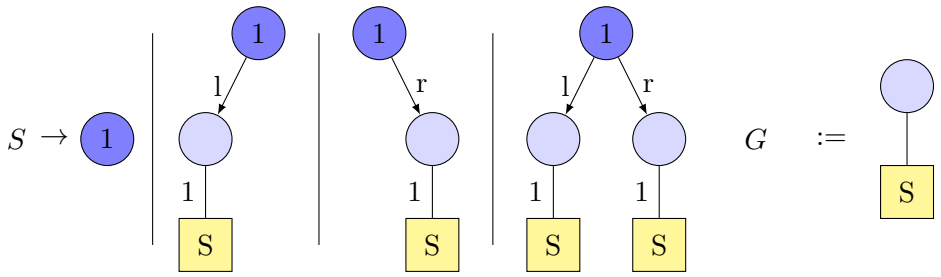
$$rk : S \mapsto 1, l \mapsto 2, r \mapsto 2.$$

According to the rationale presented above, the production rules for the only nonterminal symbol are very intuitive: We can freely choose to concretize the left and right subtree, which yields four possible productions, shown in figure 2.6(a)

We start with a single node and may then again choose how to concretize its subtrees. The starting graph resulting from this idea is shown in figure 2.6(b).

Thus, we end up with the hypergraph grammar I:

$$I := (\{S\}, \{\texttt{left}, \texttt{right}\}, P, G) \, ,$$



(a) Production rules (b) Starting graph

Figure 2.6: Parts of a grammar for binary trees

where P is the set of rules shown in figure 2.6(a) and the axiom G is shown in figure 2.6(b). We number the set of production rules from left to right, therefore $P := \{p_1, p_2, p_3, p_4\}$.

We will now derive the graph shown in figure 2.3(b) on page 12, without its external nodes. Figure 2.7 shows the individual steps of the derivation.

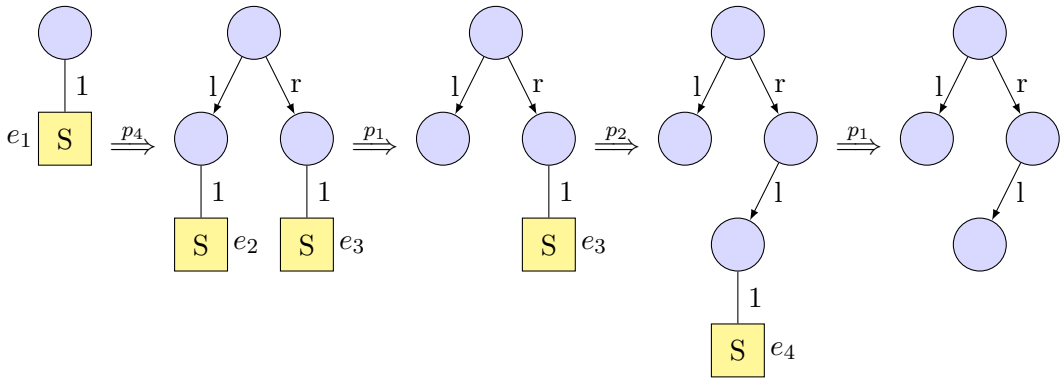


Figure 2.7: Derivation of a binary tree

As stated previously, we start the derivation from the axiom of the grammar. From there, we first apply the rightmost rule p_4, producing both a left and a right subtree. After this, we proceed to concretize the subtrees from left to right.

We apply the leftmost production e_2 to e_2, producing no subtrees for the first node on the second level. Subsequently, we produce only the left subtree for the right node on the same level by applying p_2, before we finally apply p_1 again, since the last node has no children.

This produces a terminal hypergraph, since all edges are labeled with terminal symbols.

Thus the derivation is finished and the resulting graph is in the language of grammar I.

It should be clear that derivations of hypergraphs via HRGs are realized by deriving subgraphs from nonterminal hyperedges (cf. definition 2.4.4). All these individual derivations of single hyperedges can be done locally, i.e., without regard to the other hyperedges in the graph. Thus, any derivation can be *decomposed* into the derivations of the individual hyperedges.

Lemma 2.4.1 (Decomposition Lemma)**.** *Let G and H be hypergraphs over Σ, $e_1, \ldots, e_n \in E_G$ nonterminal hyperedges in G, i.e., $\forall 1 \leq i \leq n.\ lab_G(e_i) \in N_\Sigma$. Then the following holds:*

$$(G \Rightarrow^* H)$$
$$\Leftrightarrow \quad \left(\exists K_1, \ldots, K_n \in \mathcal{HG}_\Sigma. \left(H = G\,[K_1/e_1, \ldots, K_n/e_n] \wedge \bigwedge_{i=1}^{n} ((lab_G(e_i))^\bullet \Rightarrow^* K_i)\right)\right)$$

The proof can be found in [10][p. 49f.].

Just as it is the case with string grammars, not all sets of hypergraphs can be specified by a language of a hyperedge replacement grammar. An exact characterization of the languages that can be defined by a HRG is given in [6]. It is shown there that the family of languages of HRGs is the same as the family of sets of bounded treewidth. The reader should also note that it is trivial to construct a grammar describing any finite set of graphs, by simply using a single nonterminal symbol and one production rule per graph in the set that derives the nonterminal symbol to the chosen graph.

Due to the scope of this thesis, we will not elaborate on this point any further. Interested readers can find further information on treewidth in [11], where it was first defined, and [24], in which the term "treewidth" is used for the first time.

3 Heap Abstraction using Hypergraphs

In the previous chapter we introduced hypergraphs as a general model without considering fields of application for it. In this chapter, we have a closer look at the information we can extract from the assumption that a given hypergraph is the image of the heap memory of a program during runtime. We constrain ourselves to object-oriented programs, since they still account for the major part of existing code.

In the first section we consider the program statically, i.e., without executing it. Since we do not need information about the specific instructions the program contains, but rather about the objects that can be put on the heap and their connections to one another, we distill the information about a program into its *signature*.

In section 3.2 we introduce the model of *heap configurations*, which are essentially hypergraphs with extended information on the alphabet and the vertices. This additional information can also be easily incorporated into a hypergraph when constructing it from an image of the heap-memory of a program.

Finally, we want to use hypergraph abstraction grammars for describing all possible heap configurations a program can produce during any of its runs. We discuss the implications and restrain ourselves to "meaningful" grammars in section 3.3.

3.1 Program Signatures

As stated in the introduction, we want to extract some information on the objects that are used by a given program statically. For this purpose, let P be an object-oriented program for the remainder of this section.

Definition 3.1.1 (Signature of a Program)**.** *We denote the* signature *of P by $Sig(P)$ and define it as a five-tuple as follows:*

$$Sig(P) = (class, identifier, member, type, \succ),$$

where $class$ and $identifier$ are finite sets, $member : class \to identifier^*$ is a function assigning each class its members, $type : class \times identifier \to class$ assigns a type to each member and $\succ \subseteq class \times class$ is the inheritance-relation, i.e., it is a non-reflexive, non-symmetric and "acyclic" relation on the set $class$. In this context, acyclic means that $X \succ^* Y \Rightarrow (X = Y) \vee \neg(Y \succ^* X)$ holds for all $X, Y \in N_I$.

Since each member of a class must have an identifier that is unique in the class-scope, we demand that $member(x)$ contains no identifier more than once for each $x \in class$.

The relation $\succ$ only denotes the direct inheritance relation, i.e., there is only one direct predecessor for each element. Formally speaking, the relation has to fulfill the following requirement, among the ones specified above:

$$\forall x, y \in class.\ x \succ y \Rightarrow [\forall z \in class.\ (z = x \vee \neg z \succ y)]$$

This definition takes into account that two classes may have members of the same name, but of different type. In this definition we do not consider simple datatypes, such as integer or char, since they do not contribute to the connections between objects on the heap.

Example 3.1.1 (Signature of a Tree-Implementation)**.** *Let us consider a program that only implements a binary tree and has no other classes. Since a binary tree consists of nodes and pointers to its left and right child, a signature of this implementation could look like this:*

$$(\{node\}, \{left, right\}, mem, typ, \emptyset),$$

where $mem(node) = left, right$ and $typ(node, left) = typ(node, right) = node$.

Since this implementation consists only of a single class, the inheritance relation is empty.

3.2 Heap Configurations

Since graphs can be used to describe the state of the heap memory at any given point during program execution, the same holds true for hypergraphs. However, when we construct a hypergraph from an image of the memory of a program, we can extract some more information than just the mere structure of the heap from the memory, most importantly the type information.

To keep track of this extended information, we introduce the notion of a heap configuration. This definition is based on the definition provided in [13][Definition 4].

Definition 3.2.1 (Heap Configuration)**.** *Let P be a program with $Sig(P) =: sig$. Since we know that the terminal edges of our heap configuration represent pointers, we can fix part of the alphabet for our heap configuration as*

$$\Sigma := (N, identifier_{sig}, rk)$$

for some N, with $rk(X) = 2$ for all $X \in identifier_{sig}$.

A heap configuration H of the program P is then defined as follows:

$$H := (V, E, labV, labE, att, ext, \bot),$$

where $(V, E, labE, att, ext)$ is a hypergraph, $\bot \in V$ denotes a vertex representing the null-value and $labV : V \setminus \{\bot\} \to class_{sig}$ assigns a type to every node. We furthermore

demand that each pointer is represented in the heap configuration, i.e., there is one edge for each node and each member of its type:

$$\forall v \in V, id \in [member_{sig}(labE_G(v))]\,.\ |\{e \in E \mid lab(e) = id \wedge att(e)[1] = v\}| = 1$$

Also, the pointers have to satisfy typing requirements, i.e., they have to point to vertices of a type fitting their declaration. Formally speaking, we demand that

$$\forall e \in E_G.\ \Big[type_{sig}(labV_G(att(e)[1]), labE_G(e)) \succ^* labV_G(att(e)[2])\Big] \vee \Big[att(e)[2] = \bot\Big]$$

holds.

We denote the set of all heap configurations of P by HC_P.

Since we already excluded "dangling" tentacles, i.e., tentacles that are not attached to any vertex, in the definition of hypergraphs, we model null-pointers by attaching the corresponding edges to the $\bot$-vertex.

In graphical representations of heap configurations, we display the labels of the nodes by writing them inside the vertices. Since we originally used this place for the order of external nodes, we will now display this order with numbers next to external nodes.

3.3 Heap Abstraction Grammars

Hypergraph abstraction grammars can easily be adapted to produce heap configurations, by merely changing the right sides of production rules from hypergraphs to heap configurations. We call such grammars *data structure grammars*. The previously presented properties are also intuitively applicable to data structure grammars.

However, since we will want to use these grammars algorithmically, we introduce some more constraints on them. Note that these constraints are mainly introduced to simplify algorithms later on and are not needed for describing a set of heap configurations in general.

First of all, we only want to consider grammars that describe at least a single graph and that contain no "useless" production rules and nonterminals. We call this property *productivity* and formalize it as follows:

Definition 3.3.1 (Productive Nonterminal, Hypergraph and Grammar)**.** *Let I be a data structure grammar over Σ. A nonterminal symbol X is productive if it is possible to derive a terminal hypergraph from it:*

$$X \in N_I \text{ is productive} :\Leftrightarrow L(I, X^\bullet) \neq \emptyset$$

This notion can easily be lifted to hypergraphs and grammars, analogously to string grammars:

$$G \in \mathcal{HG}_\Sigma \text{ is productive with respect to } I :\Leftrightarrow \forall X \in lab_G(E_G).\ X \text{ is productive}$$

$$I \in \mathcal{HGG}_\Sigma \text{ is productive} :\Leftrightarrow \forall X \in N_I.\ (X \text{ is productive}) \wedge (S_I \text{ is productive under } I)$$

If we restrict ourselves to consider only productive grammars, this will not result in a change of expressiveness, as is stated in the following lemma:

Lemma 3.3.1. *For each data structure grammar I with nonempty language, there is a productive grammar I' with $L(I') = L(I)$.*

The proof for this lemma can be found in [13][Appendix A]. It works similar to the corresponding proof for string grammars, in that it eliminates unproductive rules step by step. This can be done easily, since unproductive rules do not contribute to the language of the grammar.

Also, we will use productions "backwards", i.e., look for some occurrence of the right side of a production in a heap configuration and replace it with a handle of its left side. In order for this process to remain efficient and still terminate, we have to exclude productions of the form $X \to X^\bullet$ or closed circles such as $X \to Y^\bullet, Y \to X^\bullet$, where both X and Y are nonterminals.

For this we weaken the condition presented in [13] and define increasing grammars as follows:

Definition 3.3.2 (Increasing Data Structure Grammar)**.** *A data structure grammar I is increasing if there is no possibility to apply an infinite number of productions without increasing the number of nodes:*

$$I \text{ is increasing} :\Leftrightarrow \forall X \to G \in P_I.\, (|G| > |X^\bullet| \vee \neg\, (G \Rightarrow_I^* X^\bullet))$$

This definition merely formalizes the ideas presented beforehand. Either the right side of a production adds a new node ($|G| > |X^\bullet|$), or it only consists of external nodes, whereby it is a handle of some nonterminal. In the latter case we demand that the subgraph $X^\bullet$ can not be derived from the right side of the production, thus excluding the cycles mentioned.

One could also state that the production rules of I have to imply an order $\succ_I \subseteq N_I \times N_I$ on the nonterminal symbols, where $X \succ_I Y :\Leftrightarrow X^\bullet \Rightarrow_I^* Y^\bullet$

We see then that increasingness of I is equivalent to $\succ_I$ being antisymmetric.

Proof.

$$\begin{aligned}
& I \text{ is increasing} \\
\Leftrightarrow\quad & \forall X \to G \in P_I.\ (|G| > |X^\bullet| \vee \neg\, (G \Rightarrow_I^* X^\bullet)) \\
\overset{(*)}{\Leftrightarrow}\quad & \forall X, Y \in N_I, X \neq Y : \neg(X^\bullet \Rightarrow_I^* Y^\bullet) \vee \neg(Y^\bullet \Rightarrow_I^* X^\bullet) \\
\Leftrightarrow\quad & \forall X, Y \in N_I, X \neq Y : \neg(X \succ_I Y) \vee \neg(Y \succ_I X) \\
\Leftrightarrow\quad & \succ_I \text{ is antisymmetric}
\end{aligned}$$

We show the equivalence marked by $(*)$ as follows, separated into two implications.

"⇐". First, we will show the implication "from left to right" by contradiction. For this, we assume that either $|G| > |X^\bullet|$ or $\neg\,(G \Rightarrow_I^* X^\bullet)$ holds for all production rules $X \to G$ of I, but that neither $\neg(X^\bullet \Rightarrow_I^* Y^\bullet)$ nor $\neg(Y^\bullet \Rightarrow_I^* X^\bullet)$ holds for some nonterminals X and Y, and lead these assumptions to a contradiction.

Let X and Y be two nonterminals, such that $X^\bullet \Rightarrow_I^* Y^\bullet$ and $Y^\bullet \Rightarrow_I^* X^\bullet$ hold. Then there are two derivations of the form

$$X^\bullet \Rightarrow_I G_1 \Rightarrow_I \cdots \Rightarrow_I G_n \Rightarrow Y^\bullet$$

and

$$Y^\bullet \Rightarrow_I G'_1 \Rightarrow_I \cdots \Rightarrow_I G'_m \Rightarrow X^\bullet$$

for some hypergraphs $G_1, \ldots, G_n$ and $G'_1, \ldots, G'_m$. From these derivations we construct the new derivation

$$X^\bullet \Rightarrow_I G_1 \Rightarrow_I \cdots \Rightarrow_I G_n \Rightarrow Y^\bullet \Rightarrow_I G'_1 \Rightarrow_I \cdots \Rightarrow_I G'_m \Rightarrow X^\bullet.$$

The existence of this derivation however contradicts the assumption, since for the first step to be possible, there has to exist a production rule $X \to G_1 \in P_I$. The assumption of the equivalence then guarantees us that $G_1 \Rightarrow_I^* X^\bullet$ is not possible, which is obviously not the case.

Thus, we have shown the implication "from left to right" by contradiction. □

"⇒". Now we will show the implication "from left to right". Let either $\neg(X^\bullet \Rightarrow_I^* Y^\bullet)$ or $\neg(Y^\bullet \Rightarrow_I^* X^\bullet)$ hold for all nonterminal symbols in N_I and let $X \to G$ be a production rule in the grammar I.

We will show that either $|G| > |X^\bullet|$ or $\neg\,(G \Rightarrow_I^* X^\bullet)$ holds. We can distinguish two cases for the right side of the production rule. Either the rule is a "renaming" rule, i.e., its right side is only a handle, or it consists of more than a single edge and the minimum of nodes.

Case 1: There is a $Y \in N_I$, such that $Y^\bullet = G$. In this case the assumption guarantees that $Y^\bullet \Rightarrow_I^* X^\bullet$ does not hold, which is equivalent to $\neg(G \Rightarrow_I^* X^\bullet)$. Hence the implication holds in this case. □

Case 2: There is no $Y \in N_I$, such that $Y^\bullet = G$. Again, we can distinguish two cases. Either the right side of the production adds nothing to the handle of its left side, or it adds at least one node or edge.

Case 2.1: $|G| > |X^\bullet|$. In this case the left side of the implication trivially holds true, therefore the implication as a whole holds true. □

Case 2.2: $|G| = |X^\bullet|$. Since there are at least as many vertices in G as there are in $X^\bullet$, there are two possibilities for the numbers of vertices and edges in G: Either $|V_G| = rk(X) + 1$ and $|E_G| = 0$ or $|V_G| = rk(X)$ and $|E_G| = 1$.

In the former case, there is no possibility to derive any other graph from G, since there are no nonterminal edges to replace. Thus it is especially not the case that $G \Rightarrow_I^* X^\bullet$. Hence, the implication holds in this case.

In the latter case, there either is at least one node that is not connected to the single edge of G, or it holds true that $G = labV_G(e)^\bullet$, if e is the single edge of G.

Case 2.2.1: One node of G is "isolated". In the first case there is at least one node that is not connected to any edge in G. Due to the nature of replacements, this node will also not be connected to any edge in the graphs derived from G, since replacements can only "access" nodes already connected to the edge to be replaced. Thus, $X^\bullet$ can not be derived from G anymore, since all nodes of handles are attached to the single edge. □

Case 2.2.2: G is a handle of some label. In the second case, it holds that $G = labV_G(e)^\bullet$ for $E_G = \{e\}$. If $labV_G(e)$ is nonterminal, it contradicts the assumption that G is not a handle of some nonterminal. If it is terminal however, nothing can be derived from this graph anymore; especially not $X^\bullet$, hence the implication holds in both cases. □

Thus, the implication holds also if G is of the same size as the handle of its right side. □

After inspection of all possible cases, we can conclude that the implication also holds if the production rule's right side is not a handle of some nonterminal. □

Thus, the direction "from left to right" in the equivalence also holds. □

From this we can conclude that the equivalence as a whole also holds, with which we have shown that the increasingness of a hypergraph abstraction grammar is equivalent to the implied ordering $\succ_I$ being antisymmetric. □

We also need to restrain the possibilities for concretizing a given nonterminal symbol, in order to prevent creation of inadmissable heap configurations by repeated backwards- and forwards-application of productions. Consider figure 3.1, which shows a typical tree-structure with nodes of type n, which is partly abstracted and then concretized, using rules similar to those presented in figure 2.6(a). We see that this yields an invalid heap configuration, since the uppermost node has two outgoing pointers labelled with l.

The necessary condition to prevent such situations is called typedness:

Definition 3.3.3 (Typed Grammar)**.** *A data structure grammar I over the program P with $Sig(P) = sig$ is typed iff there is a function that tells us for each nonterminal's tentacles, which outgoing edges of the node it is attached to will be produced from it.*

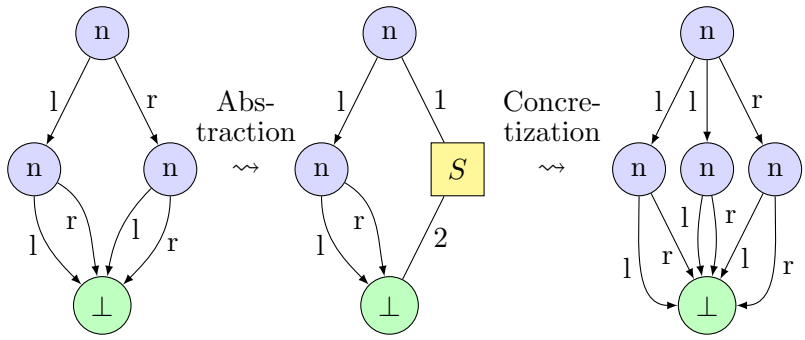


Figure 3.1: Problematic concretization using an untyped grammar

Formally speaking:

$$
\begin{aligned}
& I \text{ is typed} :\Leftrightarrow \\
& \forall X \in N_\Sigma, i \in \{1,\dots,rk(X)\}\,.\ \exists type = a_1 \dots a_n \in \Sigma^*.\ \forall G \in L(I, X^\bullet). \\
& \exists E' = \{e_1,\dots,e_n\} \subseteq E_G.\ e \in E' \Leftrightarrow att(e)[1] = ext_G[i] \\
\wedge\ & \forall 1 \le j \le n.\ labE_G(e_j) = type[j]
\end{aligned}
$$

Note that there may also be tentacles with an empty type, e.g., tentacles that are attached to the ⊥-Node. Since the null-object cannot have any outgoing pointers, tentacles attached to it always have to have an empty type. We call tentacles with an empty type reduction tentacles.

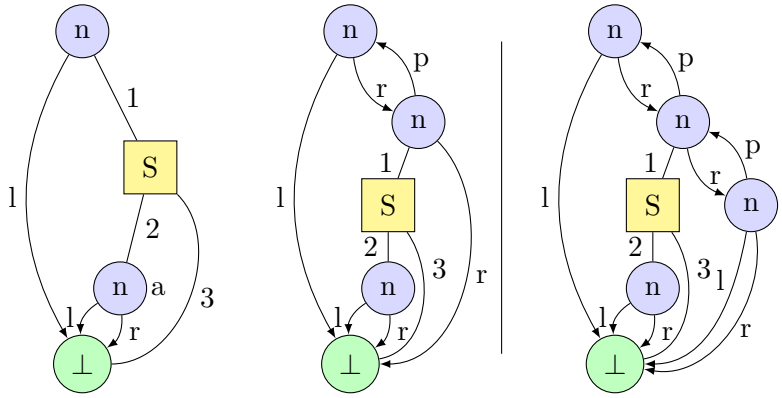
The first part of the requirement states that for each tentacle of a nonterminal edge there exists a sequence of labels. We then demand that in every terminal graph derived from the handle of the tentacle the set of labels of edges originating from the chosen node is the same as the given sequence. Informally speaking, we demand that this sequence determines the outgoing terminal edges that can be derived from this tentacle.

Note that the intuitive meaning of typedness, i.e., the fact that all pointers of terminal graphs have to point to vertices of the correct type, is already included in the definition of heap configurations.

Another reason for non-terminating repeated application of production rules is missing local concretizability.

Consider figure 3.2(a), which depicts a tree with parent-pointers and imagine we want more information about the nodes adjacent to node a. Again, using slightly adapted rules similar to those presented in figure 2.6(a), we try to concretize the environment of this node and end up with the graphs shown in figure 3.2(b). It is clear that even

the repeated application of these rules will not result in any more information on the environment of the node in question.



(a) Starting Graph (b) Possible results

Figure 3.2: Problems with non-locally-concretizable grammars

In order to solve this problem, we demand that data structure grammars are *locally concretizable*:

Definition 3.3.4 (Local Concretizability)**.** *A typed data structure grammar I is locally concretizable iff for every node attached to a nonterminal edge, there exists a set of rules that also describes the whole language as well, but in which every rule allows us to concretize all outgoing pointers of the nonterminal's type.*

Formally speaking, let X be some nonterminal from N_I, $1 \leq i \leq rk(X)$ and let $P(X,i)$ be a subset of the production rules of I.

We then define

$$I_{P(X,i)/X} := \left(N_I, T_I, \left(P_I \setminus \{X \to H \in P_I\} \cup P(X,i)\right), X^\bullet\right),$$

i.e., the grammar with the same terminal and nonterminal symbols as I, but with no productions for the nonterminal symbol X, except for those specified in $P(X,i)$. The condition for local concretizability is then formally stated as follows:

$$\begin{aligned} & I \text{ is locally concretizable} \\ :\Leftrightarrow \quad & \forall X \in N_I,\ i \in \{1,\ldots,rk(X)\}\,.\ \exists P(X,i) \subseteq P_I. \\ \Big[\quad & L\left(I_{P(X,i)/X}, X^\bullet\right) = L(I, X^\bullet) \wedge \forall X \to G \in P(X,i).\ \forall e \in E_G. \\ & [\exists j \in \mathbb{N}.\ att(e)[j] = ext_G[i]] \Rightarrow [lab(e) \in T_I \vee type(lab(e), j) = \epsilon] \quad \Big] \end{aligned}$$

Intuitively, this definition only states that all nonterminals that are still attached to the chosen vertex after application of any production rule cannot be used to produce outgoing edges anymore. However, it is still possible to concretize incoming edges to the node from this nonterminal.

Definition 3.3.5 (Heap Abstraction Grammar [13])**.** *A data structure grammar that produces only heap configurations and is productive, increasing, typed and locally concretizable is called a* heap abstraction grammar*, or* HAG *for short.*

According to [17][Theorem 4.1.2] the restriction to heap abstraction grammars does not constitute a change in expressiveness, since for each data structure grammar G a heap abstraction grammar G' can be constructed with $L(G) = L(G')$.

3.4 Example of a Heap Abstraction Grammar for Binary Trees

To summarize the conditions specified above, we give an example for a heap abstraction grammar for binary trees and demonstrate how the types for the nonterminal are chosen.

Example 3.4.1 (Heap Abstraction Grammar for Binary Trees)**.** *In this example we assume that trees consist only of vertices of a single type, called n for "node".*

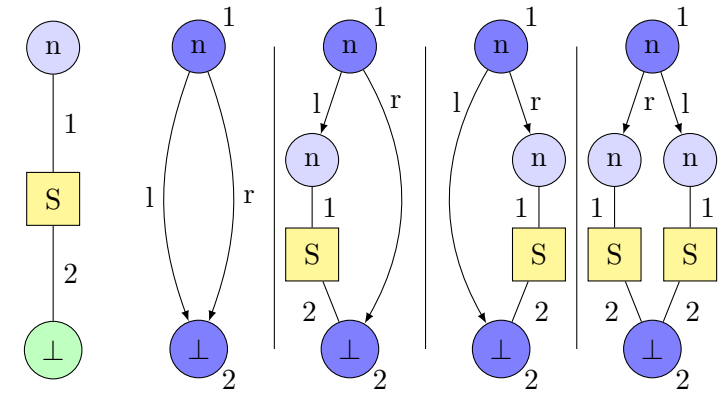
The rationale behind the grammar is quite simple and similar to the one behind the grammar presented in example 2.4.1. We start with the two components every heap configuration of a tree has to consist of: The root and the null-object. We also know that there must be some connection between the two.

The rules then only formalize the well-known rules for tree construction. Every node has a left- and a right-pointer and both point either to $\bot$, or to the root of a new subtree.

We immediately see that the grammar is productive, since we can simply apply the first rule to every nonterminal edge in every graph derived from the axiom to produce a terminal graph. It is also increasing, since every application of a rule leads either to a new vertex in the graph, as is the case with rules 2 to 4, or gives us no further possibility of deriving more graphs, in the case of rule 1.

We now have to ensure that the grammar is typed. This is obvious as well, since we can choose $Type(S, 1) = l, r$ and $Type(S, 2) = \epsilon$, since we know that we will always produce both outgoing pointers for the first external node with every rule, but that the second external node will never have an outgoing pointer. We also see that the types on the nodes satisfy the requirements, if we assume that both the left- and the right-pointer point to nodes, i.e., $type(n, l) = type(n, r) = n$.

Lastly, we have to ensure that the grammar is locally concretizable. For concretization from the only nonterminal symbol's first tentacle, we pick the subset of rules $P(S, 1) = P_I$ and see that it still describes the whole language. Also, every rule immediately concretizes all outgoing pointers, extending from the first external node. Since the only nonterminal's second tentacle is always attached to the $\bot$-vertex, it has an empty type and there can



(a) Axiom (b) Rules

Figure 3.3: Graphical Components of the Grammar

be no rule that concretizes any outgoing pointer from it. Thus, we choose $P(S,1) = P(S,2) = P_I$ *and see that the grammar as a whole is locally concretizable.*

Thus, we finally see that the provided grammar is indeed productive, increasing, typed and locally concretizable and is therefore called a heap abstraction grammar *for binary trees.*

4 Inference of Data Structure Grammars

In the previous chapter we have seen how we can use hypergraphs to describe the state of the heap at a given point during program execution. Now we define *all possible heap configurations* of the program P during any of its runs on any input as the *language of this program* $L(P)$.

The language of a program is, generally speaking, infinite and unknown, since one cannot possibly consider all of its possible inputs. Since this makes traiditional model checking approaches to verification infeasible, our goal is to find a heap abstraction grammar that describes the language of a program. Once we have such a (finite) description of the possible heap configurations, we can use it to reduce the state space of a pointer program to a finite amount of states and then apply traditional model checking. Further information on this technique can be found in [14].

However, there are two problems we have to deal with when trying to find such a grammar:

Firstly, we do not have a closed-form-description of the language yet, but rather only samples observed during some runs of the program. We call the set of these samples $\hat{L}(P)$.

Secondly, it may be possible that the actual language of the program cannot be described by any heap abstraction grammar, due to the limited expressive capabilities of hyperedge replacement and heap abstraction grammars, as described in section 2.4.

To take both these drawbacks into account, our goal in this chapter is to find an algorithm that, given the set of observed heap configurations $\hat{L}(P)$, constructs a heap abstraction grammar I that describes at least the heap configurations observed.

Note that this task is trivial if we only aim at constructing a grammar for exactly $\hat{L}(P)$, since this set is finite. We can always provide a grammar for a finite set with a single nonterminal of rank zero and one production rule for each graph in the set that derives this nonterminal to the given graph. Hence, we will have to take into consideration how to expand the language of this initial, trivial grammar *intelligently* without making the grammar itself too general.

To summarize the relations between the sets, we know that $L(P) \supseteq \hat{L}(P)$ and we are looking for a grammar I with $L(I) \supseteq \hat{L}(P)$. Note that we do not make any statement about the relation between $L(P)$ and $L(I)$.

We will describe a general principle used in grammar induction in general in the first section and then present a summary of an algorithm that produces a node-substitution grammar for simple graphs in the second section.

In the third and fourth section we formalize the ideas present in this algorithm and adapt them for the use with hypergraphs. In section five an overview over the completed inference-algorithm for heap abstraction grammars is presented.

In this chapter we omit labels on the edges where they obstruct clarity of drawings and are not necessary for understanding.

4.1 Minimum Description Length

On a very high level, the process of inferring a grammar for a given set is very easy: We start with a trivial grammar producing all graphs in $\hat{L}(P)$. We then pick some subgraph from the set of the right sides of all rules, abstract all its occurrences with a new nonterminal and add a new rule to the grammar that allows us to concretize this nonterminal again.

Pseudocode for this top-level-view of the inference-algorithm is given in listing 1, where $\text{Init}(\hat{L}(P), \Sigma)$ produces the trivial grammar for $\hat{L}(P)$.

Algorithm 1 $\text{Learn}(\hat{L}(P), \Sigma)$

Input: $L(P) \subseteq HC_\Sigma, \Sigma$
  $grammar := \text{Init}(\hat{L}(P), \Sigma)$
  $(sub, embedding) := \text{FindAbstractableSubgraph}(grammar)$
  while $sub \neq null$ **do**
    $nonterminal := \text{GetNewNonterminal}(\Sigma)$
    $grammar := \text{AbstractAllOccurrences}(grammar, sub, embedding, nonterminal)$
    $grammar := \text{AddRule}(grammar, nonterminal \to sub)$
    $(sub, embedding) := \text{FindAbstractableSubgraph}(grammar)$
  end while
Output: *grammar*

The construction of the initial grammar, as well as the addition of a nonterminal to an alphabet and the addition of a rule to a grammar are easy and deterministic.

Also, the grammar inferred by this simple algorithm only produces exactly $\hat{L}(P)$.

Therefore, the interesting questions are the following ones:

1. Which subgraph shall be abstracted?
2. How do we find all embeddings of this subgraph in all graphs in $\hat{L}(P)$?
3. When shall we stop the abstraction, i.e., return *null* when asked for a subgraph to abstract?
4. How to expand the language of the grammar, such that $L(I) \supseteq \hat{L}(P)$ holds?

We will answer the first and third question in this section, whereas answers to questions two and four can be found in section 4.3 and 4.4, respectively.

The first question is also often encountered in the field of machine learning, where it is formulated in a more general form: "When given several possible explanations of a structure, which one is to be chosen?". If we understand a grammar for a given set of heap configurations as an explanation for the common structure of these graphs, it is clear that both questions are equivalent.

A general answer for these questions is the principle of succinctness, or *Ockham's Razor*, as it is often called. The most famous formulation of this principle is given as follows in [7]:

> Pluralitas non est ponenda sine necessitate.
> (Plurality should not be posited without necessity.)

When applied to our question, this principle can also be formulated as "When given several possible explanations for a structure, the simplest one is to be chosen".

This answer has been formalized in machine learning by introducing the notion of a *description length* for structures and rules and the decision rule of *Minimum Description Length*, or MDL for short.

Definition 4.1.1 (Principle of Minimum Description Length)**.** *When given multiple contradicting rules $X \to H$ describing certain parts of a structure G, the principle of Minimum Description Length picks the rule that minimizes*

$$DL(X \to H) + DL(G \mid X \to H),$$

where $DL(X \to H)$ is the description length of the rule $X \to H$ and $DL(G \mid X \to H)$ is the description length of the structure G under the assumption that the rule $X \to H$ is known.

This principle was first introduced in [23]. Further information can be found in [12] and [9], among others.

In order to use this principle for the task at hand, we have to define the description length for sets of heap configurations and for production rules. We define these lengths as follows:

Definition 4.1.2 (Description Length)**.** *Let $H = (V, E, labV, labE, att, ext, \bot)$ be a heap configuration, $\hat{L}(P)$ a set of heap configurations and let $X \to H$ be a production rule.*

We define the description length of a heap configuration as

$$DL(H) = |V| + \sum_{e \in E} (rk(e) + 1)$$

The description length of a set of heap configurations is defined as the sum of the description lengths of all its elements:

$$DL(\hat{L}(P)) = \sum_{H \in \hat{L}(P)} DL(H)$$

Furthermore, let $X \to H$ be a production rule. We define the description length of this rule as

$$DL(X \to H) = 1 + DL(H)$$

The description length of a set of hypergraphs $\hat{L}(P)$, given a rule $X \to H$ is defined as the description length of the set L' in which all occurrences of the subgraph H have been abstracted with the nonterminal X.

We choose these definitions based on a natural understanding of hypergraphs: we assume that we need one unit of information for each node, as well as for each label and each edges tentacles. If we want to store a production rule, we have to keep the nonterminal in mind as well as the graph it can be derived to.

Note that this definition of the description length is a heuristic for the choice of the subgraph that is to be abstracted. One may simply redefine the description length to completely change the choice of abstracted subgraphs. For example, another definition could identify the description length of a graph with the number of its external nodes, which would lead to the abstraction of subgraphs with the lowest number of attachment points to the rest of the graph.

If we know the number of non-overlapping occurrences of H in $\hat{L}(P)$, we can simplify the definition of $DL(\hat{L}(P) \mid X \to H)$, such that we do not actually need to conduct the replacement only to compute the description length.

Since a replacement of a subgraph leaves the external nodes of the subgraph in the original graph, removes the internal ones as well as all edges and adds a single edge, we can compute the description length after a single replacement of H in $\hat{L}(P)$ with $DL(\hat{L}(P)) - DL(H) + 2 \cdot |ext_H| + 1$. By including $2 \cdot |ext_H|$, we account for the external nodes and the single tentacle that is attached to each node. The final offset of 1 accounts for the label of the newly added edge.

Using this direct formulation, we can rewrite the description length of a whole set under the assumption of a rule as

$$DL(\hat{L}(P) \mid X \to H) = DL(\hat{L}(P)) - \#Occ_H(\hat{L}(P)) \cdot (DL(H) + 2 \cdot |ext_H| + 1)\,,$$

where $\#Occ_H(\hat{L}(P))$ denotes the number of occurrences of H in $\hat{L}(P)$.

Using these definitions, we can fill in the first gap in the algorithm shown in listing 1. The subroutine FindAbstractableSubgraph(I) is shown in listing 2.

Note that we sanify the embeddings found by GetSubgraphs, since they may be overlapping, i.e., contain two embeddings of a subgraph that map the internal nodes of the structure to non-disjoint sets.

This problem is illustrated in figure 4.1. The original graph is shown in the middle, the structure is shown in two copies above and below the graph with its embeddings denoted by the dashed lines. It is clearly visible that the node marked with c is an internal node in both embeddings.

Such embeddings cannot be separately abstracted, therefore they have to be removed first. The SanifyEmbeddings-operation returns the maximal set of non-overlapping embeddings.

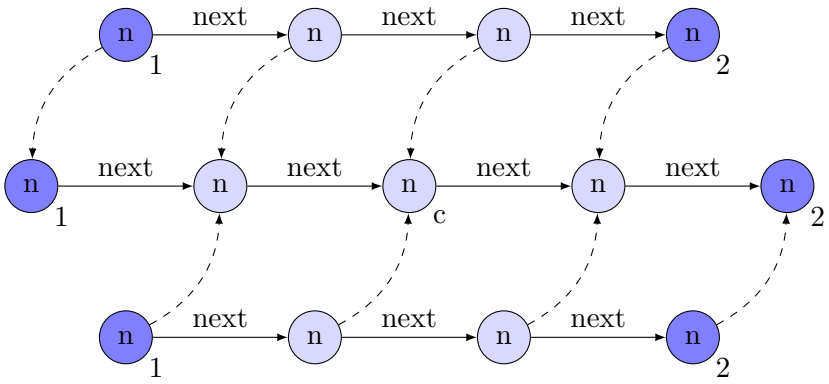


Figure 4.1: Overlapping embeddings

Also note that FindAbstractableSubgraph does not return any subgraph if none offers a gain larger than 0.

Algorithm 2 FindAbstractableSubgraph(I)

Input: $I \in DSG_\Sigma$
  $(subgraphs, embeddings) :=$ GetSubgraphs(I)
  $embeddings :=$ SanifyEmbeddings($embeddings$)
  $(optSub, embedding, optGain) = (null, 0)$
  for each $(sub, embedding) \in (subgraphs, embeddings)$ **do**
    $gain = |embeddings| \cdot (DL(sub) - 2 \cdot |ext_H| + 1)$
    $gain = gain - 1 + DL(sub)$
    if $(gain > optGain)$ **then**
      $(optSub, optEmbedding, optGain) = (sub, embedding, gain)$
    end if
  end for
Output: $(optSub, optEmbedding)$

The subprocedure GetSubgraphs(I) can, of course, be implemented naively by simply

iterating over all subsets of the input graphs' nodes. We present a more efficient way of enumerating all subgraphs in section 4.3.

4.2 SubdueGL

The principle of Minimal Description Length has already been applied to the inference of graph structures in [19] and [18], where an algorithm for graph grammar induction called SubdueGL is developed. Since we base our approach on this paper, we summarize its workings briefly and look at the non-trivial components in depth later on in sections 4.3 and 4.4.

SubdueGL works on traditional graphs and yields a node replacement grammar producing a superset of the input set. In principle, it works exactly as the algorithm described in algorithms 1 and 2, but improves the method of iterating over all subgraphs.

Instead of simply considering all subsets of the given graphs' sets of nodes, it starts with subgraphs of single nodes and extends them node by node. This way, only connected subgraphs are enumerated. Furthermore, it is not necessary to find all embeddings of a subgraph seperately, but these embeddings can be found during the actual enumeration of the subgraphs. The downside of this method is obviously that the same subgraph will be enumerated multiple times, as shown in figure 4.2. We will deal with this downside in section 4.3.

Another aspect is SubdueGL's handling of recursive data structures. After every "growth" of a substructure, it checks if a structure of the same form is reachable from it. If this is the case, it is assumed that this pattern can be repeated ad infinitum, hence this pattern is formulated as two rules for node replacement allowing for recursive concretization of this pattern. The whole chain of substructures is subsequently abstracted.

This approach only allows for simple recursions, which consist of a single pattern that is reachable from a copy of itself. More complex recursions, in which a list grows from both sides simultaneously for example, are not abstracted.

We will now discuss how to formalize the "growing" of subgraphs and how to construct their embeddings efficiently.

4.3 Subgraph Construction

The general principle of enumerating all meaningful subgraphs has been presented in the previous section. However, the question remains how to adapt this idea for hypergraphs and how to extract embeddings from the grown subgraphs.

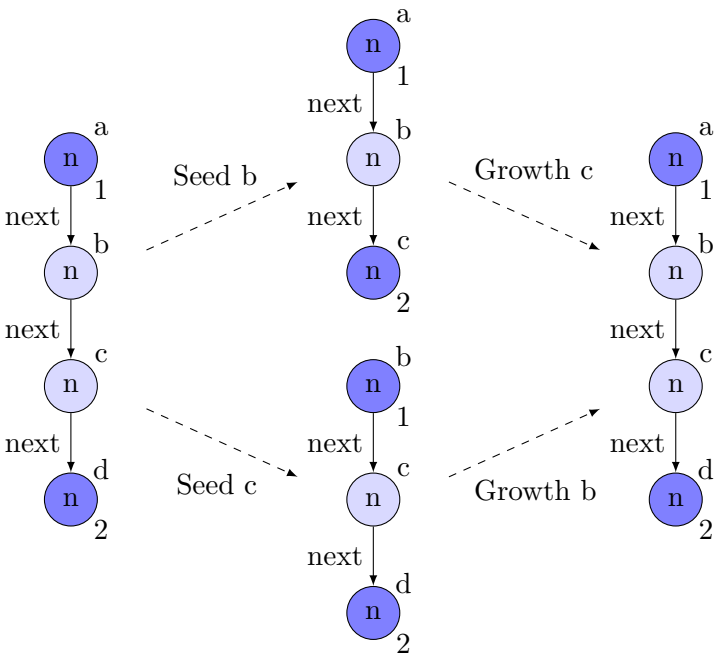


Figure 4.2: Growing the same subgraph twice

4.3.1 Subgraph Cache

For this we first introduce a subgraph cache that allows us to keep track of the generated subgraphs together with their embeddings. This cache offers the following operations:

- containsExactly($G \in HC_\Sigma$): Returns true if there is a subgraph that consists exactly of the same nodes as G.
- addGraph($G \in HC_\Sigma$): Checks if there is a graph in the cache that is isomorphic to G. If there is one, the embedding of G is saved with that structure, if there is none the subgraph G is saved directly.
- toList(): Returns a list of all saved structures and their embeddings.

Note that addGraph has an exponential runtime, since a check for graph-isomorphism is required for each pairing of the new graph and an already existing structure. We can alleviate the impact of this operation on the overall runtime by two optimizations.

Firstly, we do not keep the structures in a simple list, but rather store them in an array of lists, where the i-th list in the array contains all substructures of i internal nodes. This way, we can directly skip comparisons of structures with unequal numbers of internal nodes. This also improves the runtime of the containsExactly operation, since this operation also only has to take potentially fitting subgraphs into account now.

The second improvement consists of intelligently guessing the isomorphism itself by taking labelings of nodes into account. We start by constructing an initial partial isomorphism by mapping those node onto each other that are of a type that only appears once in each graph and call a recursive subroutine with this partial isomorphism.

This subroutine first propagates the assignments of this partial isomorphism by mapping each successor of an already mapped node to the successor of its mapped node. If we encounter a contradiction during propagation, we return an error-code. If we finish the propagation without contradictions and without unmapped nodes, we return the completed mapping.

Otherwise we guess a single mapping between two unmapped nodes of the same type in both graphs, add this mapping to the existing one and call the subroutine recursively. If this call returns successfully, we pass the returned mapping to the caller. If the call returns an error, we either guess another pair of nodes to be mapped, or we return an error, if there is no pair of nodes that has not been tested.

This method combines the information we can gather from the heap configuration with the propagation of guessed assignments. This principle is often found in state-of-the-art SAT solvers. For more information on this topic, refer to [15], where this method was first introduced, or [4], where a complete SAT solver using this technique is described.

4.3.2 Actual Enumeration

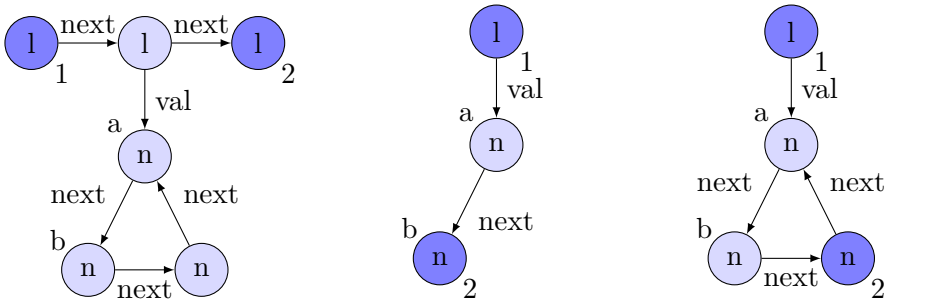
The actual enumeration of subgraphs consists mainly of two subroutines. For every internal node of a given graph the first subroutine, GenerateSeeds($L \subseteq HC_\Sigma$), generates a subgraph that only contains the chosen node and the external nodes that are necessary for replacing it in the graph. We omit the concrete description of the procedure in pseudocode, but rather present an example of the generation of a single seed in figure 4.3(b).

The second subroutine, generateExtensions($sub \in HC_\Sigma$) takes a single graph as input and checks if the graph can be extended at all. A simple criterion for determining if a graph can be extended would be to check if it has the same number of internal nodes as the graph it is a subgraph of. If this is the case, we cannot extend it anymore.

However, if we keep our original goal of finding subgraphs whose abstraction leads to a decrease in description length in mind, we can already stop the creation of extensions when we reach a graph that has half as many internal nodes as the original graph.

Any graph that has more internal nodes cannot have another non-overlapping embedding in the original graph, since this would require the original graph to have at least two more internal nodes. Thus, we stop the creation of extensions when we reach a graph that has half as many internal nodes as the graph it was created from.

If the given graph is not maximal, the subroutine creates an extension for each of its external nodes by making it internal and adding the external nodes that are necessary for replacing the newly generated subgraph. We again omit the listing of the procedure in pseudocode, but rather present an example of the extension via a single node in figure 4.3(c).



(a) Original Graph (b) Seed from node a (c) Extension via node b

Figure 4.3: Creation of seeds and extension of subgraphs

The complete enumeration procedure only has very few responsibilities: It has to generate the seeds, let them grow and check for every grown subgraph whether it already exists. If it does exist, then it is discarded, since it was already considered earlier. If it

does not exist, then its structure and embedding is stored in the cache and it is again considered for extension. The complete procedure is presented in listing 3.

Algorithm 3 GetSubgraphs(I)

Input: $I \in DSG_\Sigma$
  $L := \{G \mid X \to G \in P_I\}$
  $cache = new\ SubGraphCache()$
  $queue = new\ SubGraphQueue()$
  $queue.addAll(GenerateSeeds(L))$
  $sub = queue.poll()$
  while $sub \neq null$ **do**
    if $\neg cache.containsExactly(sub)$ **then**
      $queue.addAll(generateExtensions(sub))$
      $cache.addGraph(sub)$
    end if
    $sub = queue.poll()$
  end while
Output: $cache.toList()$

Using this enumeration procedure, we can efficiently iterate over all subgraphs and pick the one that offers the largest decrease in total description length, as shown in algorithm 2.

However, this approach would only generate the given set of hypergraphs exactly. In order to generate a superset of it, we will have to account for recursive substructures. In the next section we formalize the ideas of SubdueGL for handling recursion presented in section 4.2 and adapt them to work on heap configurations.

4.4 Recursive Structures

The strategy for recognizing recursive structures employed in SubdueGL was simply to check for each structure, whether or not there is a single edge that connects one embedding to another one. This approach does not work on hypergraphs in its original form, since we have to keep the external nodes in the original graph to serve as attachment points. We therefore choose to formalize the fact that a structure is recursive as follows:

Definition 4.4.1 (Recursive Structure)**.** *Let S be a subgraph of G and let $int_S := V_S \setminus [ext_S]$ be the internal nodes of S.*

S is recursive if and only if there exist two embeddings $iso_1, iso_2 : V_S \to V_G$ of this

subgraph in G, such that the following holds:

$$\begin{aligned}
&\forall n \in [ext_S].\ \nexists e \in E.\ labE(e) \in N \wedge n \in att(e) \\
\wedge\ & iso_1(int_S) \cap iso_2(int_S) = \emptyset \wedge \exists ext_{entry}, ext_{exit} \subseteq [ext_S]\,. \\
\Big[\ & ext_{entry} \cup ext_{exit} = [ext_S] \wedge ext_{entry} \cap ext_{exit} = \emptyset \\
\wedge\ & \forall en \in ext_{enter}.\ \nexists e \in E.\ labE(e) \in T \wedge att(e)[2] = en \\
\wedge\ & \forall ex \in ext_{exit}.\ \nexists e \in E.\ labE(e) \in T \wedge att(e)[1] = ex \\
\wedge\ & \forall ex \in ext_{exit}.\ [pred_S(iso_2^{-1}(iso_1(ex))] \subseteq ext_{entry} \qquad \Big],
\end{aligned}$$

where $pred_S(v)$ denotes the predecessors of a given node in S. The predecessors are those nodes that have a pointer to the given node.

This definition states that the two embeddings do not share internal nodes, i.e., they must be non-overlapping. Furthermore, no nonterminal hyperedge may be attached to any external node. Also, there has to exist a partitioning of the external nodes into entry- and exit-nodes, such that no entry-node is the target of a terminal edge and such that no exit-node is the source of a terminal edge.

Also, all predecessors of each exit-node in the first embedding are entry-nodes of the second embedding.

Using this definition, if a recursive structure is found, it can be attached to itself using the exit- and entry-nodes as attachment points, with the result still being a valid subgraph of G.

It is clear that this definition restricts us to simple cases of recursion. We will discuss this later on together with experimental results in chapter 5.

Since this definition is not trivially intuitive, we provide an example of a recursive structure.

Example 4.4.1 (Recursive Structure)**.** *The middle part of figure 4.4 shows the original graph in which recursion is to be found. Above and below it, copies of the same structure are displayed. The embeddings are denoted with arrows labeled with iso_1 and iso_2, respectively.*

It can be seen that these embeddings share no internal nodes. Also, if we partition the external nodes into $ext_{entry} = \{a\}$ and $ext_{exit} = \{b, c\}$ we see that this partitioning fulfills the conditions for a recursive structure.

Thus, we have found two embeddings of the same structure in the same graph that share no internal nodes as well as a partitioning of the substructure's external nodes, such that the conditions for a recursive structure are fulfilled.

Hence, the considered substructure is recursive.

Note that a recursive structure always has internal counterparts for its external entry- and exit-nodes, as shown in figure 4.5 on the example for the structure shown in figure 4.4. We refer to these internal counterparts as internal entry- and exit-nodes.

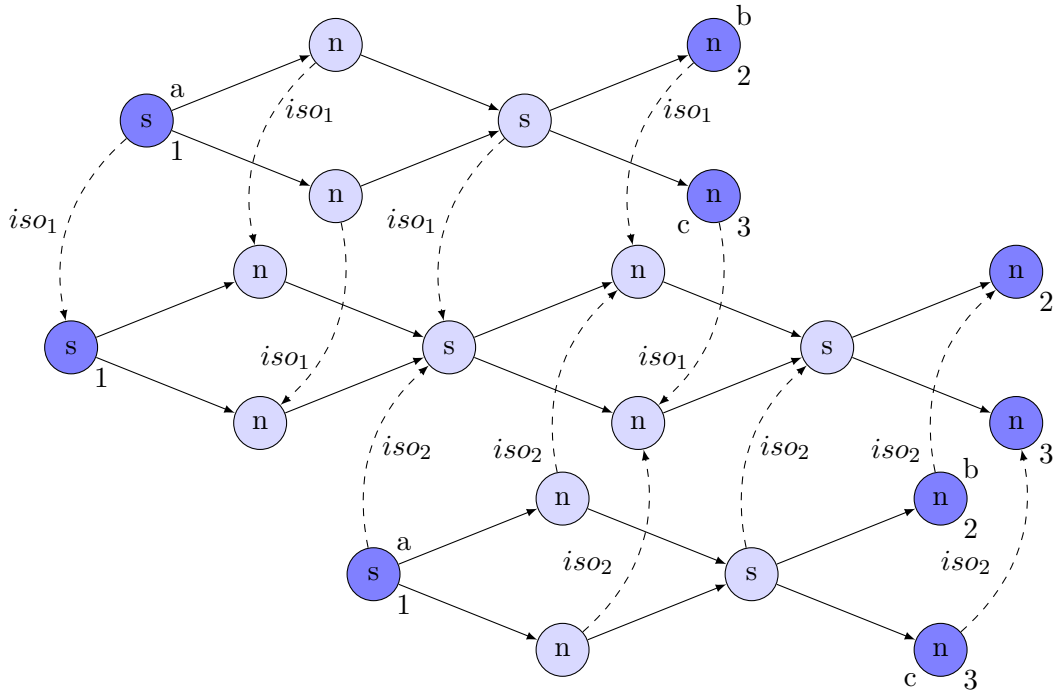


Figure 4.4: Recursive structure

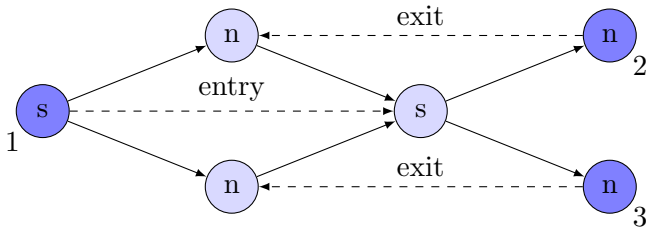


Figure 4.5: Correspondence between internal and external entry- and exit-nodes

The question remains how to handle these recursive substructures. The processing of these structures happens in two parts. First we abstract each occurrence of a chain of this recursive substructure with a single nonterminal, then we add rules stating that this nonterminal edge may be replaced with a chain of concatenations of the recursive structure. This chain may have arbitrary length.

Definition 4.4.2 (Recursive Rules)**.** *Let G be a recursive structure and let ext_{ent}, ext_{exit}, int_{ent} and int_{exit} be its internal and external entry- and exit-nodes, respectively. Furthermore, let X be a nonterminal symbol.*

We then construct a hypergraph $G_{\rightarrow}$ that allows concretization from the external entry-nodes by removing all connections between the internal entry-nodes and the external exit-nodes. Subsequently, we add a single nonterminal edge labeled with X that connects the internal entry-nodes and the external exit-nodes.

We construct a second hypergraph $G_{\leftarrow}$ by repeating the construction above with the external entry-nodes and the internal exit-nodes, thus allowing concretization from the external exit-nodes.

Lastly, we construct a "stop graph" $G_{\bullet}$ to finish the concretization of the recursive structure. This graph is constructed by copying the external entry-nodes and the internal exit-nodes and the connections between them. We then set the internal exit-node as external, thus producing a graph that has no occurrence of the given nonterminal and describes the connections between the entry- and exit-nodes.

Since the external entry- and exit-nodes are a partitioning of all external nodes, this construction retains local concretizability when the resulting rules are added to an already locally concretizable grammar.

All tentacles of the new nonterminal are typed, since those tentacles attached to the exit-nodes do not produce any outgoing pointers. Also, those attached to the entry-nodes produce the outgoing pointers attached to them in the original structure, either when concretizing using the graph $G_{\rightarrow}$ or when using the final graph $G_{\bullet}$.

We give an example for this construction in example 4.4.2.

Finally, we have to abstract all occurrences of this recursive structure. The usual approach used in algorithm 1 is not applicable here, since it requires the embeddings to be non-overlapping. However, when abstracting recursive structures, we are explicitely looking for overlapping embeddings.

For a given structure we search for a "chain" of embeddings $e_1, e_2, \ldots, e_n$, such that every pair e_i, e_{i+1} fulfills the conditions posed to iso_1 and iso_2 in the definition of a recursive structure.

We use this chain to construct the recursive rules first, since this construction needs two embeddings of the structure to determine the partition of internal and external nodes.

We then remove all internal nodes of these embeddings from the original graph and connect the external entry-nodes of e_1 and the external exit-nodes of e_n with a new

nonterminal edge.

This leads us to the approach presented in algorithms 4 and 5.

Algorithm 4 AbstractRecursion(I, Σ)

Input: $I \in DSG_\Sigma, \Sigma$
  $recursionFound := true$
  while $recursionFound = true$ **do**
    $recursionFound := false$
    $(structures, embeddings) :=$ getSubgraphs(I)
    $(I_{old}, \Sigma_{old}) := (I, \Sigma)$
    for each $(structure, embedding) \in (structures, embeddings)$ **do**
      $(I, \Sigma) :=$ AbstractStructure($structure, embedding, I, \Sigma$)
      if $(I, \Sigma) \neq (I_{old}, \Sigma_{old})$ **then**
        $recursionFound := true$
        break
      end if
    end for
  end while
Output: I, Σ

Algorithm 5 AbstractStructure($structure, embeddings, I, \Sigma$)

Input: $structure \in HC_\Sigma, embeddings, I \in DSG_\Sigma, \Sigma$
  $embeddingList := findEmbeddingChain(structure, embeddings)$
  if $\neg embeddingList.empty()$ **then**
    $X :=$ getNewNonterminal(I, Σ)
    $(G_\rightarrow, G_\leftarrow, G_\bullet) :=$ constructRuleGraphs($structure, embeddingList$)
    $P_I := P_I \cup \{X \rightarrow G_\rightarrow, X \rightarrow G_\leftarrow, X \rightarrow G_\bullet\}$
    removeInternalNodes($embeddings, I$)
    $externalNodes := ext_{enter}(embeddingList.front) \cup ext_{exit}(embeddingList.back)$
    addNonterminalEdge($externalNodes, X$)
    removeEmbeddings($embeddings, embeddingList$)
  end if
Output: I, Σ

Example 4.4.2 (Recursion Abstraction)**.** *Figure 4.6 shows the result of abstracting the recursive structure shown in figure 4.4. We have used the two embeddings shown in that figure as the only entries of the chain of embeddings and the only remaining nodes are those shown in figure 4.6(a).*

The graphs constructed from this structure and the two embeddings are shown in the

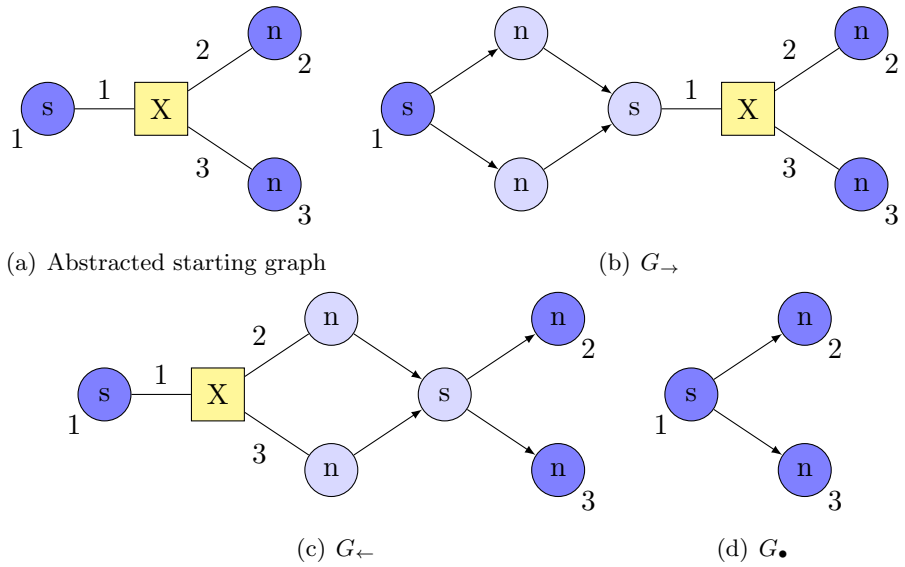


Figure 4.6: Result of the abstraction of a recursive structure

figures 4.6(b), 4.6(c) and 4.6(d). We see that these rules fulfill the requirements for locally concretizable grammars and are also typed, since they do not produce outgoing pointers at the external exit-nodes and they ultimately produce the two outgoing pointers of the single external entry-node.

4.5 Complete Algorithm

Now that we have described the separate parts of the algorithm, we are ready to describe the algorithm as a whole. Since we also want to take into account recursive structures as described in the previous section, we will have to change the general approach presented in algorithm 1. Our approach is to find a single recursive structure, abstract it as far as possible and then start the traversal of subgraphs anew, since the information about subgraphs has gone out-of-date after the first abstractions.

If we do not find a recursive structure, we switch to the MDL-based approach of abstracting common substructures after updating the information on subgraphs once more. We repeat this MDL-based abstraction until there is no subgraph anymore that can be abstracted without an increase in description length.

Before we pass the set of subgraphs and embeddings to the MDL-heuristic, we will have to sanify it, however, since the subgraph enumeration may yield overlapping embeddings. We do this by computing the largest possible set of non-overlapping embeddings for each

structure and encapsulate this process in the SanifyEmbeddings-subroutine.

The whole algorithm is presented in listing 6.

Algorithm 6 Learn($\hat{L}(P), \Sigma$)

Input: $\hat{L}(P) \subseteq HC_\Sigma, \Sigma$
  $I :=$ Init($\hat{L}(P), \Sigma$)
  $(I, \Sigma) :=$ AbstractRecursion(I, Σ)
  $(optSub, optEmbedding) :=$ FindAbstractableSubgraph(I)
  while $optSub \neq null$ **do**
    $(I, \Sigma) :=$ AbstractAllOccurrences($I, optSub, optEmbedding$)
    $(optSub, optEmbedding) :=$ FindAbstractableSubgraph(I)
  end while
Output: I

The algorithm is correct, since its output grammar produces at least the heap configurations given as input. This is easy to see if we consider the three operations that actually manipulate the grammar separately.

As reasoned before, the initial grammar produces exactly the graphs given as input, therefore

$$I = \text{Init}(\hat{L}(P), \Sigma) \Rightarrow L(I) = \hat{L}(P)$$

holds.

The abstraction of all occurrences also keeps the language of the input grammar, since it uses a fresh nonterminal symbol for the abstraction. Since this nonterminal symbol can only be concretized to the graph that was abstracted with it, this replacement also does not change the language of the input grammar, i.e.,

$$(I', \Sigma') = \text{AbstractAllOccurrences}(I, sub, emb) \Rightarrow L(I') = L(I)$$

Finally, the abstraction of recursion only expands the language of the given input grammar, since the only subgraphs removed are concatenations of recursive structures. However, the subroutine also adds rules that state that this abstraction may be replaced by a chain of this structure of arbitrary length, hence especially the original structure can be concretized again. Thus, the following holds:

$$(I', \Sigma') = \text{AbstractRecursion}(I, \Sigma) \Rightarrow L(I') \supseteq L(I)$$

From these observations we can easily deduce that

$$I = \text{Learn}(\hat{L}(P), \Sigma) \Rightarrow L(I) \supseteq \hat{L}(P)$$

holds. Thus, the algorithm is correct.

5 Experimental Results

We have implemented the algorithm presented in the previous section using the Juggrnaut-framework [1], which is a Java-framework providing facilities for modeling hypergraphs and heap abstraction grammars. For the source code, please refer to `http://www.moves.rwth-aachen.de/i2/juggrnaut/`.

We have provided it with several testcases and present the resulting rules and time needed for the relevant parts of the algorithm.

All times are measured by a simple comparison of the values of the Java function `System.nanoTime()` directly before and after the procedure to be measured. All runtimes have been measured three times under the same conditions and decimal places are cut off. In the tabular presentation of the runtimes, Ri denotes the i-th run.

The experiments were conducted using an Intel Core i7 M620 with 2.67 GhZ and the OpenJDK runtime environment in version 1.6.0_24.

For simplification we used only a single hypergraph as the set of input graphs for each run.

5.1 Singly Linked Lists

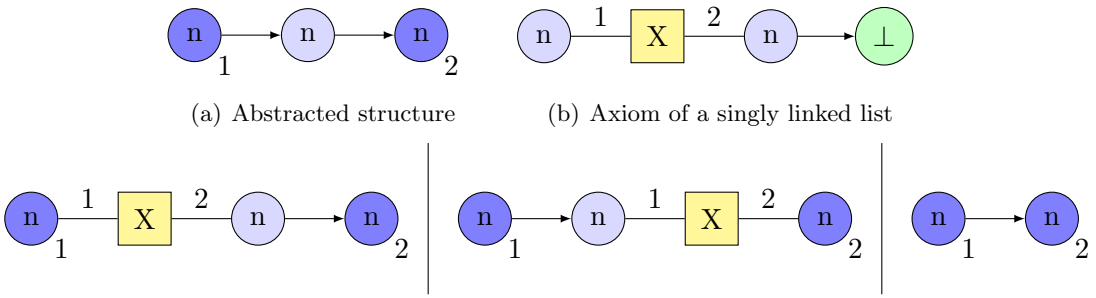
The first testcase consists of singly linked lists with numbers of nodes between 25 and 200. As expected, the algorithm finds the recursive structure shown in figure 5.1(a), abstracts it and creates the rules shown in figure 5.1(c).

The output consists of the axiom shown in figure 5.1(b) and the rules shown. The times needed for enumeration of all subgraphs and the complete abstraction of the recursive structure are shown in table 5.1. Abstraction using the MDL-principle did not take place.

5.2 Circular Singly Linked Lists

The second testcase consists of circular singly linked lists, i.e., lists that are similar to a singly linked list, but in which the next-pointer of the last node points to the first node of the list again.

The resulting rules are exactly the same as those for singly linked lists. The only change is in the axiom of the grammar, which is depicted in figure 5.2. The runtimes for



(a) Abstracted structure (b) Axiom of a singly linked list

(c) Production rules for a singly linked list

Figure 5.1: Results of abstracting a singly linked list

	Subgraph Creation			Complete Recursive Abstraction		
Nodes	R1 [ms]	R2 [ms]	R3 [ms]	R1 [ms]	R2 [ms]	R3 [ms]
25	102	84	84	106	94	106
50	308	276	273	326	309	281
75	443	433	437	493	491	518
100	644	636	648	655	680	712
125	987	1 016	1 002	1 053	1 029	1 038
150	1 511	1 443	1 412	1 549	1 502	1 527
175	1 944	1 846	1 862	2 102	1 918	1 980
200	2 875	2 963	2 847	3 022	3 019	3 043

Table 5.1: Runtimes of processing singly linked lists

constructing these rules are shown in table 5.2. As with singly linked lists, abstraction using the MDL-principle did not take place.

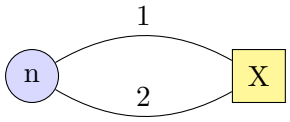


Figure 5.2: Axiom of a circular singly linked list

	Subgraph Creation			Complete Recursive Abstraction		
Nodes	R1 [ms]	R2 [ms]	R3 [ms]	R1 [ms]	R2 [ms]	R3 [ms]
25	76	74	74	97	82	92
50	281	272	229	294	258	292
75	463	473	418	452	466	473
100	663	698	678	652	670	653
125	900	1 009	1 028	1 171	926	1 000
150	1 470	1 416	1 508	1 507	1 538	1 489
175	1 919	1 839	1 910	1 849	1 962	1 988
200	2 735	2 790	2 890	3 019	2 946	3 021

Table 5.2: Runtimes of processing circular singly linked lists

5.3 Nested Lists

In this testcase we construct a nested list as input for the algorithm. A nested list is a list, in which each entry has a second pointer to another list. The lists the pointers of the outer list point to do not share any nodes.

For this example we choose the outer list to be a singly linked non-circular list, whereas the inner lists are singly linked and circular.

Two nonterminals were used by the algorithm. The first nonterminal, X_1, is used to abstract the inner, circular lists and has the same production rules as those shown in figure 5.1(c). The second nonterminal, X_2, is used to abstract the outer, non-circular list. The recursive structure used for inferring the rules for X_2 is shown in figure 5.3.

The runtime needed for inferring each of these nonterminals are shown in table 5.3. The number of inner nodes refers to the number of inner nodes per inner list, i.e., for each outer node there exist the given number of inner nodes.

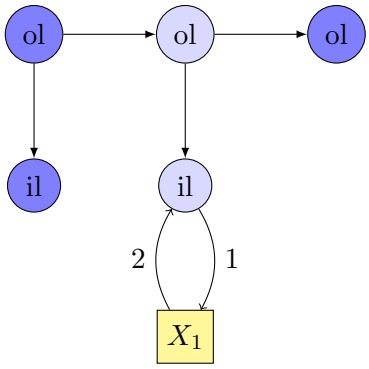


Figure 5.3: Abstracted structure of outer nested list

Nodes		Inner Abstraction			Outer Abstraction		
Outer	Inner	R1 [ms]	R2 [ms]	R3 [ms]	R1 [ms]	R2 [ms]	R3 [ms]
2	2	6	6	14	12	5	6
2	4	19	18	19	5	5	5
4	2	36	29	28	12	25	10
4	4	392	391	399	10	11	12
4	5	1 290	1 239	1 312	35	13	11
5	2	99	98	91	24	22	24
5	4	2 723	2 682	2 699	22	22	22
5	5	51 147	51 593	52 084	22	23	23

Table 5.3: Runtimes of processing nested lists

5.4 Binary Tree

In this section we use the simple binary tree presented in figure 5.4 as input.

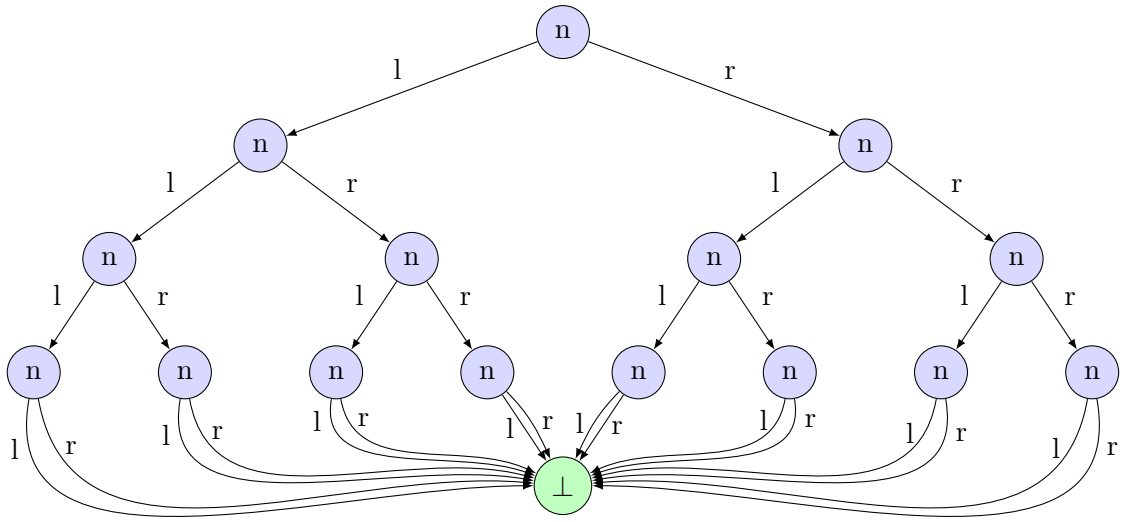


Figure 5.4: Second testcase for the algorithm

The algorithm does not find any recursive structure. We discuss the reasons for this in section 5.5. It subsequently chooses to abstract the structure shown in figure 5.5(a) in its four occurrences.

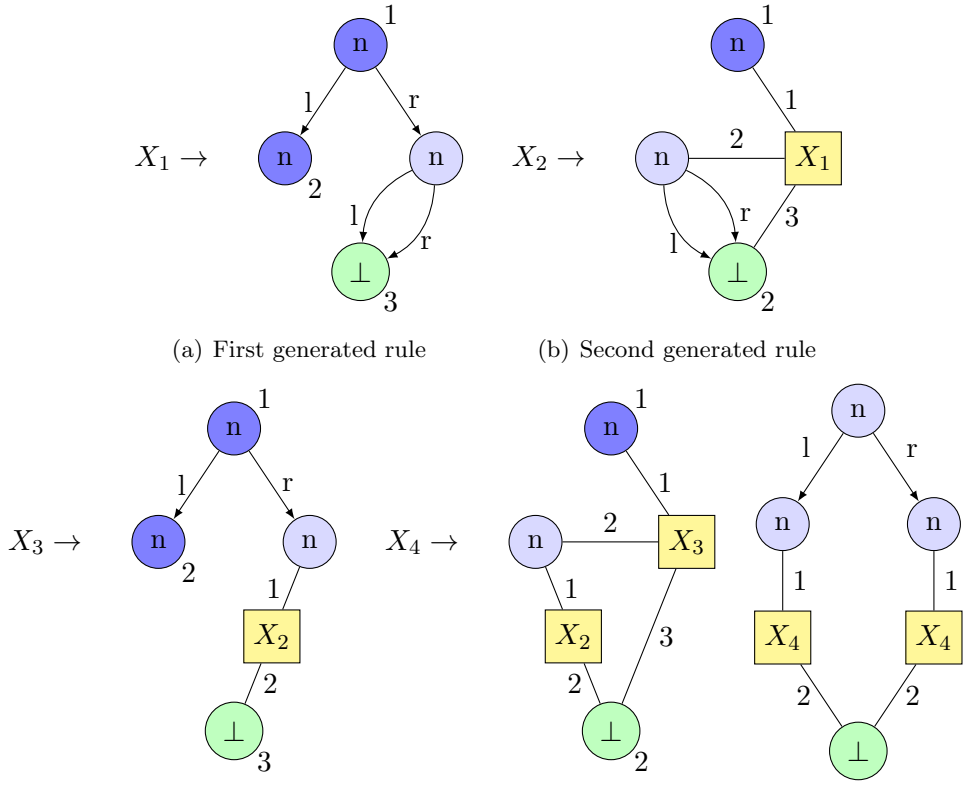
Subsequent iterations fail to find recursive structures as well and abstract the structures shown in figures 5.5(b), 5.5(c) and 5.5(d). The algorithm terminates with the axiom shown in figure 5.5(e).

The runtime for finding each of these structures is presented in table 5.4.

5.5 Discussion

The first and second testcase already shows some interesting results. First of all, the generated rules are exactly those that a human would probably construct for a singly linked list. This should come as no surprise, since we have formalized human intuition for the abstraction of recursive structures.

The more interesting result is shown in the tables 5.1 and 5.2. Let us first look at the runtimes of the subgraph enumeration. We see that these runtimes are still exponential



(a) First generated rule (b) Second generated rule

(c) Third generated rule (d) Fourth generated rule (e) Axiom

Figure 5.5: Results of the second testcase

Inference	R1 [ms]	R2 [ms]	R3 [ms]
X_1	116	121	114
X_2	44	50	61
X_3	41	50	41
X_4	31	27	24
axiom	27	28	24
sum	256	276	264

Table 5.4: Runtimes for the inference of tree-rules

in the number of nodes, which is unavoidable, given the exponential increase in possible subgraphs in graphs with more nodes. However, we also see that the enumeration of all subgraphs takes less than three seconds even for larger graphs, which we think is a reasonable amount of time, especially if we consider that this enumeration also yields all embeddings of these subgraphs.

When we consider the complete abstraction we see that the actual finding of recursive structures only takes about 10% of the total time, which means that further optimization should be targeted at the enumeration of subgraphs instead of the finding of recursive structures.

The example for nested lists show that the approach still works for more complicated structures, as long as they consist of the simple concatenations of structures described in section 4.4. However, this test case also shows that even though we have optimized the runtime of subgraph enumeration, abstraction of larger subgraphs quickly leads to very large runtimes.

The experiment with the binary tree demonstrates that our algorithm is only able to handle simple recursive structures, i.e., those that have clearly defined entry- and exit-points and that consist of concatenations of a single structure. A tree however has two subtrees for every node, and is therefore not recognized by the algorithm.

We also see that the algorithm makes different choices from a human, since it chooses only to abstract the right successors of the nodes on the second level. The reason for this is our method of enumeration, which does not produce the subgraph with a single external parent node and its two successors marked as internal.

6 Conclusion

In this chapter we briefly summarize the contents of this work and present open questions that have arisen during conception and implementation of the inference algorithm.

6.1 Summary

We first introduced hypergraphs and hyperedge replacement grammars analogous to string grammars. We then extended these models to describe single heap configurations and the set of all heap configurations arising during a run of a program.

We subsequently constructed an algorithm for automatic inference of a data structure grammar from a given set of observed heap configurations. For this, we adapted an existing algorithm that produces rules for node substitution and extended it to work for hypergraphs.

We also looked into the efficient implementation of this algorithm, especially in terms of efficient enumeration of all subgraphs of a set of hypergraphs.

The resulting algorithm is able to produce the shortest possible description for a given set of heap configurations, with respect to some measure of description length. It recognizes simple recursions as well and abstracts them with very few rules for each recursive structure.

This algorithm has also been implemented using the Juggrnaut-Framework and tested on some examples, which confirmed the practical applicability of the used concepts.

6.2 Open Questions

Even though the implementation shows that the constructed algorithm can produce concise descriptions of given sets of hypergraphs and recognize simple recursive structures, there are three main points for further work, which have arisen during development.

The computation of non-overlapping embeddings for a given structure works greedily at the moment, which does not always produce the maximal number of non-overlapping embeddings. One will have to find a method to efficiently compute the maximal number of non-overlapping embeddings.

Furthermore, our enumeration of subgraphs is quite efficient, but only produces subgraphs with connected internal nodes. During the abstraction of a binary tree in section

5.4 we have seen that subgraphs with unconnected internal nodes could also prove useful. In further research one will have to investigate ways of efficiently enumerating such subgraphs.

Finally, the mechanism for recognizing recursive structures is currently very limited and only recognizes recursive structures that are connected to a single copy of themselves, which already excludes typical structures such as trees. In further research one will have to devise a formal description of more complicated recursion and produce an efficient method for determining whether or not a subgraph and some of its embeddings describe a recursive structure. A starting point would be to allow multiple sets of exit-nodes, all of which lead to another copy of the structure.

Bibliography

[1] Juggrnaut website. `http://www-i2.informatik.rwth-aachen.de/i2/juggrnaut/`.

[2] Henrik Barthels. Automata-Based Detection of Hypergraph Embeddings. Bachelor's Thesis, RWTH Aachen University, September 2011.

[3] Edmund Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Berlin / Heidelberg, 1982. 10.1007/BFb0025774.

[4] Philippe Codognet and Daniel Diaz. A simple and efficient boolean solver for Constraint Logic Programming. *Journal of Automated Reasoning*, 17:97–128, 1996. 10.1007/BF00247670.

[5] Bruno Courcelle. An Axiomatic Definition of Context-Free Rewriting and its Application to NLC Graph Grammars. *Theoretical Computer Science*, 55(2-3):141–181, December 1987.

[6] Frank Drewes and Hans-Jörg Kreowski. A note on hyperedge replacement. In Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 1–11. Springer Berlin / Heidelberg, 1991.

[7] Encyclopedia Britannica Online. Ockham's Razor. Retrieved 25 July, 2012, from `http://www.britannica.com/EBchecked/topic/424706/Ockhams-razor`.

[8] Robert W. Floyd. Assigning Meanings to Programs. *Mathematical Structures in Computer Science*, pages 19–32, 1967.

[9] Peter D. Grünwald, In Jae Myun, and Mark A. Pitt, editors. *Advances in Minimum Description Length*. The MIT Press, 2005.

[10] Annegret Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg / New York, 1992.

[11] Rudolf Halin. S-functions for graphs. *Journal of Geometry*, 8:171–186, 1976.

[12] Mark H. Hansen and Bin Yu. Model Selection and the Principle of Minimum Description Length. *Journal of the American Statistical Association*, 96(454):746–774, 2001.

[13] Jonathan Heinen, Christina Jansen, Joost-Pieter Katoen, and Thomas Noll. Juggrnaut: Using Graph Grammars for Abstracting Unbounded Heap Structures. tbp.

[14] Jonathan Heinen, Thomas Noll, and Stefan Rieger. Juggrnaut: Graph Grammar Abstraction for Unbounded Heap Structures. In Einar Broch Johnsen and Volker Stolz, editors, *Harnessing Theories for Tool Support in Software, Preliminary Proceedings*, pages 53–67. United Nations University - International Institute for Software Technology, August 2009.

[15] Pascal Van Hentenryck, editor. *Constraint Satisfaction in Logic Programming*. The MIT Press, 1989.

[16] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[17] Christina Jansen. Konstruktion und Inferenz von Hypergraphabstraktionsgrammatiken. Diplomarbeit, RWTH Aachen University, January 2010.

[18] Istvan Jonyer, Lawrence B. Holder, and Diane J. Cook. MDL-Based Context-Free Graph Grammar Induction And Applications. *International Journal on Artificial Intelligence Tools*, 13(1):65–79, 2004.

[19] Istwan Jonyer, Lawrence B. Holder, and Diane J. Cook. Concept Formation Using Graph Grammars. In *Proceedings of the KDD Workshop on Multi-Relational Data Mining*, 2002.

[20] Vincent Maraia. *The Build Master: Microsoft's Software Configuration Management Best Practices*. Addison-Wesley Professional, 2005.

[21] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer Berlin / Heidelberg, 1982.

[22] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55 – 74, 2002.

[23] Jorma Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, 1978.

[24] Neil Robertson and P.D. Seymour. Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

[25] Linus Torvalds. Linux Kernel v1.0.0, March 1994. Retrieved 25 July, 2012, from `http://www.kernel.org/pub/linux/kernel/v1.0/linux-1.0.tar.gz`. Generated using David A. Wheeler's 'SLOCCount'.

[26] Linus Torvalds. Linux Kernel v3.4.4, June 2012. Retrieved 25 July, 2012, from `http://www.kernel.org/pub/linux/kernel/v3.x/linux-3.4.4.tar.gz`. Generated using David A. Wheeler's 'SLOCCount'.