RWTH Aachen University
Software Modeling and Verification Group

Master Thesis

Reconciling Decidability of Separation Logic Entailment and Graph Grammar Language Inclusion

submitted by

Christoph Matheja

Thesis Supervisor:	apl. Prof. Dr. Thomas Noll
Second Examiner:	Prof. Dr. Ir. Joost-Pieter Katoen
Submitted:	October 1, 2014

Remark

This is a revised version in which a more general fragment of tree-like hyperedge replacement grammars is studied. Thus, the introduction of tree-like hyperedge replacement grammars and the corresponding proofs in Chapter 5 and Chapter 6 have been overhauled. In contrast to previous versions, the requirement that all outgoing edges of a vertex are defined by a single derivation step has been dropped. It should be noted that this change is only cosmetic for languages of heaps, because the number of outgoing edges is restricted by a finite set of selectors for each object on the heap. Thus, the changes do not influence the expressiveness of tree-like data structure grammars and we can always construct an equivalent data structure grammar that satisfies the dropped requirement. From a language theoretic perspective, however, dropping this requirement allows us to realize a larger class of graph languages with tree-like hyperedge replacement grammars, e.g. the set of all singly-linked lists in which an additional vertex points to every list element. In addition to that, the revised version of tree-like hyperedge replacement grammars is more flexible and it is easier to check whether a hyperedge replacement grammar is tree like, although the actual proof that every tree-like hyperedge replacement grammar is definable in monadic second-order logic is a bit more involved.

Abstract

Although dynamic data structures are common in modern programming languages, they still pose a complex challenge for formal verification, because their use often leads to infinite state spaces. Thus, formalisms to represent infinite languages of memory states - heaps - by finite structures are needed. An interesting problem for such structures is the language inclusion problem, i.e. the question whether all heaps defined by one specification are also defined by another specification. For example, automata-based LTL model-checking relies on the fact that the inclusion problem for regular languages over infinite words is decidable. In this thesis, we study graphical and logical formalisms to specify (infinite) sets of heaps, like the set of all heaps corresponding to a doubly-linked list. In particular, we identify a syntactic fragment of hyperedge replacement grammars (HRG) that can be translated into sentences in monadic second-order logic (MSO) over hypergraphs. It follows, that the language inclusion problem of the corresponding class of *tree-like* HRGs is decidable. Based on the close relationship between HRGs and separation logic with recursive definitions studied by Dodds [Dod08] and Jansen et al. [JGN14], we show that the notion of tree-like HRGs extends recently studied fragments of separation logic with a decidable entailment problem [IRS13], [IRV14].

Acknowledgements

I would like to thank my advisors apl. Prof. Dr. Thomas Noll and Prof. Dr. Ir. Joost-Pieter Katoen for their supervision of this thesis. In particular, I am grateful to Christina Jansen, who always had time for me when I knocked on the door, for her valuable and constructive suggestions during the development of this work. Finally, I wish to thank my parents for their faith and support.

Declaration of Work

Aachen, October 1, 2014
Christoph Matheja

Contents

CHAPTER 1

Introduction

Software development is naturally accompanied by the question whether a piece of software is actually working as intended. This question is getting more and more important as software is advancing into almost all aspects of daily life. Stormsurge barriers, air traffic control, railway transportation and medical devices are just a few examples of software controlled systems where failures may be catastrophic. Thus, in addition to careful software design and testing, sound mathematical approaches to verify safety-critical programs and program designs are needed. Over the time, a wide variety of formal methods, like static analysis, model-checking and theorem proving have been developed and successfully applied to verify real-world applications.

In parallel, dynamic data structures, i.e. graph-like arrangements of memory on the heap, have become so common in software development that one can hardly think of a modern programming language without libraries or build-in support for data structures, like linked lists or dictionaries. While dynamic data structures add flexibility to software, they also introduce new problems. For example, programmers have to take care to work only with locations on the heap which have properly assigned values and to free locations that are not needed anymore. Otherwise, errors like dereferencing of invalid pointers or memory leaks can occur, which may crash a system completely or lead to serious security threats. Thus, formal verification is even more important for programs manipulating data on the heap.

However, the use of dynamic data structures poses a complex challenge for most verification methods, because the set of all memory states, i.e. heaps, that can be associated with a dynamic data structure is infinite in general and cannot be determined at compile time. For example, it is possible to add, modify and remove arbitrary many objects from a linked list at runtime. Hence, there have been efforts to identify formalisms that are able to represent infinite sets of heaps by finite structures.

These formalisms have to deal with two conflicting requirements. On the one hand, a formalism should abstract from concrete sets of heaps corresponding to a dynamic data structure such that sets of heaps can be properly specified and analyzed. On the other hand, enough information to reason about a dynamic data structure has to be preserved by a specification. For instance, we may abstract from concrete data values and concentrate on the structural composition of a linked list in order to verify that an algorithm never attempts to remove elements from an empty list.

Two approaches to model dynamic data structures are *hyperedge replacement grammars* (HRG) [Hab92] and logical specification languages. Intuitively, HRGs are an extension of context-free string grammars that describe how a graph is constructed by replacing dedicated nonterminal symbols in a graph by other graphs according to a fixed set of production rules. Then, the set of graphs generated by an HRG is the set of all graphs that can be constructed from a fixed initial graph. Since every heap can be represented by a finite labeled graph where every vertex corresponds to an object on the heap and every edge corresponds to a pointer between such objects, HRGs are also suitable to model sets of heaps. As an example, consider Figure 1.1 (a), which illustrates an HRG realizing singly-linked lists. This HRG consists of two production

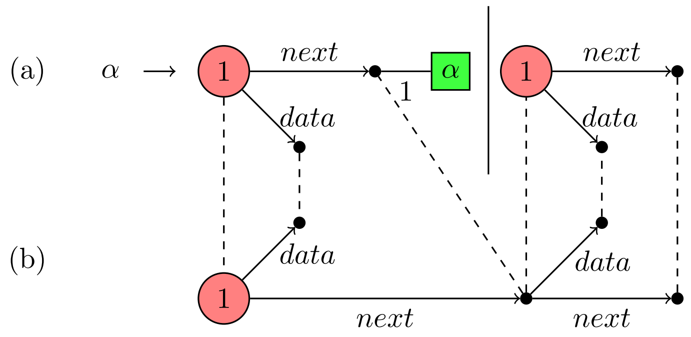


Figure 1.1.: (a) An HRG realising a singly-linked list, (b) A list with two elements derived from the HRG

rules with the same nonterminal symbol α as left-hand side. The first production rule states that a list consists of an initial list element that is equipped with a pointer to some data and a pointer to the remaining list represented by the nonterminal symbol α. The second production rule states that the last element of a list is just a single list element. Figure 1.1 (b) shows a singly-linked list with two elements that is obtained by applying the second production rule to replace the nonterminal symbol α in the first production rule. The dashed lines indicate which vertices in the list have been generated by which elements of both production rules. Note that this modeling approach is close to an actual implementation of a dynamic data structure, because its stepwise construction is encoded in the production rules of an HRG.

In contrast to HRGs, logical specification languages are a declarative approach, i.e. only the properties every heap in a set must have are defined. For instance, a logical specification of a singly-linked list would require that every heap forms a connected acyclic graph in which every object has at most one outgoing and one ingoing edge. One example of a logical specification language to define sets of heaps is *separation logic* [Rey00], which is an extension of Hoare logic [Hoa69] to reason about heap manipulating programs. The main advantage of separation logic is that it supports local reasoning, i.e. it is possible to split heaps into multiple parts that can be verified independently.

Recently, Dodds [Dod08] and Jansen et al. [JGN14] established a link between HRGs and separation logic by showing that a fragment of HRGs known as data structure grammars is equivalent to a restricted class of separation logic formulae. This result is especially interesting, because similar characterizations known from formal language theory have useful applications in formal verification. For example, automata-based LTL model-checking relies on the fact that every LTL formula can be translated into an equivalent automaton over infinite strings and that the language inclusion problem, i.e. the question whether all strings accepted by one automaton are also accepted by another automaton, for these automata is decidable [BK$^+$08]. Furthermore, it follows that the entailment problem, i.e. question whether all models of a given formula are also models of another formula, is decidable for LTL.

Unfortunately, a similar application is not possible for HRGs and separation logic, because the language inclusion problem for HRGs as well as the entailment problem for separation logic is undecidable. However, fragments of separation logic with a decidable entailment problem have been studied by Iosif et al. [IRS13] [IRV14]. Since these fragments are included in the class of separation logic formulae considered by Jansen et al. [JGN14], the question arises which conditions are necessary such that a fragment of HRGs has a decidable language inclusion problem.

In addition to that, the proofs by Iosif et al. are based on the translation of a given separation logic formula into a sentence in *monadic second-order logic* (MSO) over graphs of bounded tree width. For this logic, the entailment problem is known to be decidable (cf. [CE12]). This approach is of interest, because a well-known theorem by Büchi states that the class of regular string languages can be characterized by finite automata as well as by MSO sentences over an appropriate structure of strings [Büc60]. Furthermore, the class of right-linear string grammars also captures exactly the class of regular string languages. Since HRGs are an extension of context-free string grammars to generate graph languages, the question which fragments of HRGs have a decidable language inclusion problem leads to an older language-theoretic question: (How) can the theorem of Büchi be lifted to graph languages, i.e. is there a proper notion of "regular" graph languages?

In this thesis, we study fragments of HRGs with a decidable language inclusion problem and their relation to separation logic and MSO. Our main contribution is the development of a syntactic fragment of hyperedge replacement grammars that can be translated into sentences in monadic second-order logic over hypergraphs. It follows, that the language inclusion problem for this fragment of *tree-like* HRGs is decidable. Intuitively, tree-like HRGs allow the reconstruction of a derivation tree in MSO from a given graph. With the help of this derivation tree, it can then be verified in MSO whether a graph belongs to the language generated by a given tree-like HRG or not. This intuition is compliant with the extensive work of Engelfriet and Courcelle on MSO definable graph languages [Cou90] [Cou91] [SR97] [CE12].

Furthermore, based on the results of Dodds [Dod08] and Jansen et al. [JGN14], we provide a fragment of separation logic that is equivalent to tree-like HRGs generating sets of heaps only. We show that this fragment is strictly more expressive than other fragments of separation logic with a decidable entailment problem which have been proposed in the literature (cf. [IRS13], [IRV14]). Hence, an alternative proof of the results in [IRS13] is obtained. Moreover, an equivalent fragment of HRGs for each of these fragments is developed. We consider some algorithmic consequences of this relationship. For example, we provide an efficient algorithm to check whether the satisfiability problem for the fragment of separation logic proposed in [IRS13] is decidable with the help of tree-like HRGs. In addition to that, the resulting fragments of HRGs and separation logic are compared with respect to their expressiveness. For each fragment, we study the complexity of the emptiness/satisfiability problem and the language inclusion/entailment problem, respectively. For most of these fragments however, no upper complexity bound is known, yet. We also note that it is an open question, whether a set of connected (simple) graphs exists that is MSO definable and realizable by an HRG, but not realizable by a tree-like HRG.

Outline

At first, we collect some general notation used throughout this thesis and formally introduce heaps as they are used in the following in Chapter 2. Then, different approaches to model sets of heaps are studied in Chapter 3 and Chapter 4. We distinguish between graphical modeling of heaps, in which a heap is viewed as a special kind of graph, and logical modeling of heaps, in which the properties a set of heaps must have are specified in a logical language. Chapter 3 formally introduces hyperedge replacement grammars as a formalism to describe context-free graph languages and we recall some basic facts from the literature. Furthermore, we briefly discuss the relation to classical context-free string languages and regular tree languages. In compliance with [JGN14], we report on data structure grammars, which are a restricted class of HRGs realizing sets of heaps only.

In Chapter 4, we look at two approaches to describe hypergraph languages in terms of logical formulae. At first, we present monadic second-order logic over hypergraph models as a natural extension of classical results on the relationship between regular languages and logic. This logic forms the basis of decidability proofs for HRGs and separation logic presented in latter chapters. After that, separation logic with recursive definitions is formally defined in Section 4.2 and we report on the relation between separation logic and HRGs as shown by Jansen et al. [JGN14] in Section 4.3. Chapter 5 discusses the intuition underlying hypergraph languages with a decidable language inclusion problem based on the work of Engelfriet and Courcelle [Cou90] [Cou91] [SR97] [CE12]. Furthermore, we identify a syntactic fragment of HRGs, called *tree-like HRGs*, and show our main result in Section 5.2:

> Every tree-like HRG can be translated into an equivalent formula in monadic second-order logic over hypergraphs.

It follows, that the language inclusion problem for tree-like HRGs is decidable. In Section 5.3, we discuss how tree-likeness can be checked for a given HRG and provide an efficient algorithm to solve the emptiness problem. The classes of hypergraph languages considered in this thesis are compared Chapter 6. We especially show that our fragment of tree-like HRGs extends fragments of separation logic with a decidable entailment problem that have been proposed in the literature by Iosif et al. [IRS13] [IRV14]. Finally, we draw a conclusion and mention some possible future work in Chapter 7.

CHAPTER 2

Preliminaries

In this chapter, we collect some general notions which are mostly independent of logic and hypergraph languages and introduce a common notation which will be used throughout the rest of this thesis. Since all formalisms considered in this thesis are intended to specify sets of heaps, our memory model and heaps are introduced. We assume that the reader has some basic knowledge of mathematical logic and automata theory.

2.1. General Notation

Sets and Sequences We use lower-case letters $x, y, z, ...$ to denote single objects of any type, like variables, functions, vertices, hyperedges etc., and upper-case letters $X, Y, Z, ...$ to denote sets of such objects, respectively. The power set of a given set X is denoted by 2^X. Moreover, the disjoint union of two sets X, Y is denoted by $X \dot{\cup} Y$. More complex collections of objects are represented by calligraphic or Gothic letters. For example, graphs and hypergraphs are denoted by calligraphic letters $\mathcal{A}, \mathcal{B}, \mathcal{H}, ...$, grammars are denoted by the Gothic letter $\mathfrak{G}$ and automata are denoted by the Gothic letters $\mathfrak{A}, \mathfrak{B}$, etc., respectively.

In some constructions, it is useful to consider ordered sequences instead of sets. A finite ordered sequence of objects $x_1, ..., x_n$ is written as a word $\overline{x} = x_1 \cdot x_2 ... \cdot x_n$ where $\overline{x}[i]$ refers to the i-th element x_i of the sequence $\bar{x}$ and $x_i \cdot x_j$ stands for the concatenation of two objects. Whenever the context is clear, we write $x_i x_j$ instead of $x_i \cdot x_j$. We write $\overline{x}[i..j]$ to denote the subsequence $\overline{x}[i] \cdot \overline{x}[i+1] \cdot ... \cdot \overline{x}[j]$ for $i \leq j$. The set of all finite sequences over a given set of objects X is denoted by $X^\star$, where ε is the empty word. Furthermore, the set of all objects in a sequence is denoted by $[\overline{x}] = \{x \mid \exists i.x = \overline{x}[i]\}$ and conversely $\langle X \rangle$ stands for an arbitrary fixed order of all elements in the set X. For a set X and a sequence $\overline{x}$, we write $|X|$ and $|\overline{x}|$ to denote the cardinality of X and $\overline{x}$, respectively. As usual, $\mathbb{N}$ denotes the set of natural numbers. A finite set $A = \{a_1, ..., a_n\}$ is also called an *alphabet* and its elements are called *symbols*. Moreover, we write $X[y/z]$ to denote the set obtained from X by replacing the element z by y.

Functions A partial function f with domain $dom(f) = A$ and image $img(f) = B$ is written $f : A \rightharpoonup B$. For each $x \notin dom(f)$, we write $f(x) = \bot$ where the special symbol $\bot \notin img(f)$ denotes that the function value is undefined. The empty function f with $dom(h) = img(h) = \emptyset$ is denoted by $f_\emptyset$. A *labeling function* is a partial function $f : A \rightharpoonup B$ such that B is an alphabet. Sometimes, we represent a labeling function ℓ by a partition of sets $\overline{N_\ell} = N_{b_1}...N_{b_k}$ where $\overline{N_\ell}[i]$ contains exactly all objects $x \in dom(f)$ such that $\ell(x) = b_i$. We write $\ell_{\overline{N}}$ to denote the labeling function corresponding to the sequence $\overline{N}$ and $\overline{N_\ell}$ to denote the partition corresponding to ℓ, respectively. For convenience, $[x_1 \mapsto y_1, ..., x_n \mapsto y_n]$ denotes the finite partial function $f : \{x_1, ..., x_n\} \rightharpoonup \{y_1, ..., y_n\}$ with $f(x_i) = y_i$ for $i \in [1, n]$. Furthermore, the notation $f[z \mapsto y]$ is used to describe a function g such that $g(x) = f(x)$ for all $x \neq z$ and $g(z) = y$.

Plain Graphs Let A and B be alphabets. A labeled directed plain graph is a tuple $\mathcal{G} = (V, E, \jmath, \ell)$ where V is a finite set of vertices, $E \subseteq V \times V$ is a finite relation of edges between these vertices, $\jmath : E \rightharpoonup 2^B$ is a partial function assigning a set of labels to each edge and $\ell : V \rightharpoonup A$ is a vertex labeling function. We call a plain graph undirected if E is symmetric and $\jmath((x, y)) = \jmath((y, x))$ for each $(x, y) \in E$. Furthermore, a plain graph $\mathcal{G}$ is called *simple* if it contains no loops, i.e. $(x, x) \notin E$ for all $x \in V$. If not stated otherwise, we only consider simple graphs in this thesis. The set of all plain graphs over alphabets A and B is denoted by $PG_{A,B}$. Whenever multiple graphs $\mathcal{H}, \mathcal{G}$ are considered, $V_\mathcal{H}, E_\mathcal{G}$ and so on denote the components V and E in the respective graph $\mathcal{H}$ and $\mathcal{G}$.

Trees A set of sequences $X \subseteq A^\star$ is called *prefix-closed* if for each $\overline{a} \in X$ and each $0 \leq i \leq |\overline{a}|$, we have $\overline{a}[0..i] \in X$. A *tree* over an alphabet A is a partial function $t : \mathbb{N}^\star \rightharpoonup A$ such that $dom(t)$ is a finite prefix-closed set and additionally $t(p \cdot i) \neq \bot$ implies that $t(p \cdot j) \neq \bot$ for each $p \in dom(t), i \in \mathbb{N}, j < i$. Obviously, we can construct a plain graph T for each tree t and therefore identify each sequence $p \in dom(t)$ with a vertex of T. The top-most position ε of a tree is also called the *root*. Analogously, positions without successors are called *leaves*. For inductive definitions on the structure of trees, we will use the following equivalent definition. Let A be a finite ranked alphabet, i.e. there is a function $rk : A \to \mathbb{N}$ assigning a rank to every symbol. Then, every symbol $a \in A$ with $rk(a) = 0$ is a tree. Furthermore, if $t_1, ..., t_k$ are trees and $a \in A$ with $rk(a) = k$ is a symbol of rank k, then $a(t_1, ..., t_k)$ is a tree, too. The set of all trees over an alphabet A is denoted by $Trees_A$.

Context-Free String Grammars A *context-free string grammar* is a tuple $\mathfrak{G} = (N, T, \mathfrak{P}, \xi)$ where N is an alphabet of *nonterminal* symbols, T is an alphabet of *terminal* symbols with $N \cap T = \emptyset$, $\mathfrak{P}$ is a finite set of *production rules* and $\xi \in N$ is the *initial* nonterminal symbol. We denote nonterminal symbols by lower-case Greek letters $\alpha, \beta, \gamma, \xi...$ to distinguish them from terminal symbols. Every production rule $\mathfrak{a} = \alpha \to \overline{a}$ in $\mathfrak{P}$ maps a single nonterminal symbol α to a finite string $\overline{a} \in (N \dot\cup T)^\star$. We say that a string $\overline{v}$ can be derived from a string $\overline{w}$ using a production rule $\mathfrak{a} = \alpha \to \overline{a} \in \mathfrak{P}$, written $\overline{v} \xRightarrow{\mathfrak{a}}_{\mathfrak{G}} \overline{w}$, if $\overline{v} = \overline{v_1} \cdot \alpha \cdot \overline{v_2}$ and $\overline{w} = \overline{v_1} \cdot \overline{a} \cdot \overline{v_2}$. Furthermore, we write $\overline{v} \Longrightarrow_{\mathfrak{G}} \overline{w}$ if there exists a production rule $\mathfrak{a} \in \mathfrak{P}$ such that $\overline{v} \overset{\mathfrak{a}}{\Longrightarrow}_{\mathfrak{G}} \overline{w}$. The transitive closure of the relation $\Longrightarrow_{\mathfrak{G}}$ is denoted by $\Longrightarrow_{\mathfrak{G}}{}^{\star}$. The *language* generated by a context-free string grammar $\mathfrak{G} = (N, T, \mathfrak{P}, \xi)$ is the set of all finite strings that can be derived from the initial nonterminal symbol, i.e. $L(\mathfrak{G}) = \{\overline{w} \in T^{\star} \mid \xi \Longrightarrow_{\mathfrak{G}}{}^{\star} \overline{w}\}$. The class of all languages that can be generated by context-free string grammars is the class of context-free string languages.

Every context-free string grammar can be written as a grammar in *Greibach normal form*, i.e. all production rules either map to the empty string ε or to a string $a\beta_1 ... \beta_k$ for some $k \in \mathbb{N}$ such that $a \in T$ is a terminal symbol and $\beta_1, ..., \beta_k \in N$ are nonterminal symbols. The class of *regular* string languages, i.e. languages accepted by finite string automata, is strictly contained in the class of context-free string languages. Furthermore, every regular string language can be represented by a *right-linear* string grammar, i.e. a context-free string grammar in which every production rule is of the form $\alpha \to a\beta$ or $\alpha \to \varepsilon$ for $a \in T$ and $\beta \in N$.

2.2. Heaps

The basic memory model considered in this thesis is close to the memory model of separation logic as introduced in [Rey00] (see also [O'H12]). More precisely, memory consists of a countable set of *locations* $\texttt{Loc} = \mathbb{N}$. Each location either contains an integer value, a reference to another location or nothing, represented by a reference to the special location `null`. Let $\texttt{Val} = \mathbb{Z} \cup \{\texttt{null}\}$ denote the set of possible values of a location. Note that there is no explicit distinction between integer values and references to other memory locations.

Definition 2.2.1 (Heap)**.** A *heap* is a finite partial function $h : \texttt{Loc} \rightharpoonup \texttt{Val}$ mapping a value to each location in its domain. The empty heap is denoted by $h = []$ and the set of all heaps is denoted by `Heaps`, respectively. ■

Intuitively, a heap is a single state of a program represented by a collection of memory cells. For instance, the heap $h = [3 \mapsto 5, 4 \mapsto x, 5 \mapsto 7, 6 \mapsto y, 7 \mapsto \texttt{null}, 8 \mapsto z]$ describes a possible state of a singly-linked list. In this example, we implicitly encoded a "list object" consisting of two locations $(next, value)$ where $next$ points to the next list object and $value$ refers to actual data. Figure 2.1 illustrates the heap from above where cells of the same color together represent a single list item.

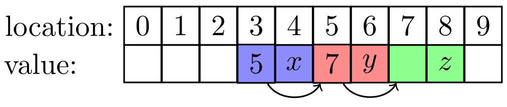


Figure 2.1.: A singly-linked list on the heap

For more complicated objects it can be tedious to encode them directly in heaps using this kind of pointer arithmetic. Thus, we assume that a heap consists of objects which group a finite sequence of continuous locations. Every object is represented by the first location in this sequence and equipped with a finite set S of *selectors* such that every $s \in S$ uniquely identifies one location belonging to the object. To be more precise, let $cn : S \rightharpoonup [0, k-1]$ be a bijective function assigning an offset to each selector. An object of size $k \in \mathbb{N}$ is a successive finite block of locations $i \cdot (i+1) \cdot ... \cdot (i+k-1)$ such that for each $j \in [0, k-1]$ there is a selector $s \in S$ such that $i + cn(s)$ refers to the location $i+j$. If an object is represented by a location x, we also write $x.s$ to denote the location $x + cn(s)$. To describe our previous example in this way, we choose $S = \{next, value\}$ as a set of selectors and get an equivalent heap

$$\begin{aligned} h = [& \\ & L.next \mapsto L.value + 1, L.value \mapsto x, \\ & L.next.next \mapsto L.next.value + 1, L.next.value \mapsto y, \\ & L.next.next.next \mapsto \texttt{null}, L.next.next.value \mapsto z \\ &] \end{aligned}$$

where the initial list object L is placed at location 3 again. Since every object has a finite set of selectors, there exists a maximal finite set of selectors S which is denoted by Ⓢ. For simplicity, we assume that each object has exactly size $\mid$ Ⓢ $\mid$ and that locations $1, 1+ \mid$ Ⓢ $\mid, ..., n\cdot \mid$ Ⓢ $\mid$ are used to represent n objects on a heap. This assumption is not a semantic restriction, but may increase the amount of memory needed. For a given heap h, we denote this set of locations by $\texttt{Obj}(h)$.

We discuss how sets of heaps are formally represented by graph grammars and logical formalisms in Chapter 3 and Chapter 4, respectively. Note that all formalisms considered in this thesis are only used specify the composition of dynamic data structures, but not actual data, i.e. arbitrary concrete (integer) values cannot be represented. Thus, it is possible to express that the second component of a list item points to a dedicated location x, but not that x equals 23. If such concrete values are needed, the definitions presented in this thesis must be adapted to support integer arithmetic or infinite sets of labels.

CHAPTER 3

Abstract Graphical Representations of Heaps

The idea of modeling data structures with the help of plain graphs may be one of the oldest in computer science. Since a heap $h : \texttt{Loc} \rightharpoonup \texttt{Val}$, as introduced in Chapter 2, describes a single state in memory, it is straightforward to define a corresponding plain graph $\mathcal{H} = (V, E, \jmath, \ell)$ where each vertex corresponds to a location in $dom(h) \cup img(h)$, each edge corresponds to a selector connecting two locations and the labeling functions assign the index of a location to a vertex and the selector to an edge, respectively. These graphs can be used to represent a single memory configuration. However, in order to model dynamic data structures, we have to model (possibly infinite) sets of heaps, for instance all possible heaps corresponding to a doubly-linked list. Hence, a finite mechanism to describe infinite sets of heaps is required. For strings, a wide range of formalisms like automata, grammars, syntactic monoids and monadic second-order logic is well-known. We will see that some of these formalisms can be lifted to describe sets of graphs and especially heaps while for others there is no known equivalent for graphs.

This chapter deals with graph grammars to define infinite sets of labeled *hypergraphs*. Graph grammars have been extensively studied since 1979 (cf. [Ehr79]) in various forms and are typically classified into node replacement grammars and edge replacement grammars, i.e. based on the criterion how nonterminals are replaced. In this work, we consider *hyperedge replacement grammars* which were introduced by Habel (cf. [Hab92]) and have been studied under the name of context-free graph grammars by Engelfriet and Courcelle among equivalent algebraic and logical formalisms (see chapter 3 of [SR97] and [CE12] for an overview).

We introduce hypergraphs and some necessary definitions to formally describe hyperedge replacement. Then, hyperedge replacement grammars are introduced and we discuss some useful properties. Finally, we describe a restricted class of hyperedge replacement grammars which ensures that only valid heaps are generated. Our introduction of hyperedge replacement grammars is mainly based on [Hab92] and [JGN14].

3.1. Hypergraphs

A *hypergraph* is an extension of a plain graph such that edges are not represented by a relation between vertices, but are objects on their own connecting one or more vertices. Informally, a *hyperedge* is an object connected to an ordered sequence of *source vertices* and *target vertices*. Figure 3.1 provides a graphical representation of a single hyperedge

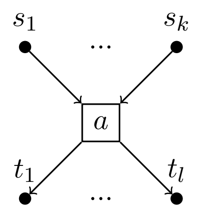


Figure 3.1.: A hyperedge with its tentacles

labeled with a together with its source vertices $s_1, ..., s_k$ and target vertices $t_1, ..., t_l$. We call a vertex *attached* to a hyperedge if it is either a source vertex or a target vertex of it. Furthermore, connections between a hyperedge and its source and target vertices are also called *tentacles* and their direction is used to indicate which vertices are source vertices and target vertices, respectively. The *rank* of a hyperedge e, written $rk(e)$, equals the sum attached vertices of e. When considering labeled graphs, we assume that the rank is already determined by the label of a hyperedge.

Definition 3.1.1 (Labeled Hypergraph)**.** Let A be a finite alphabet with a ranking function $rk : A \rightharpoonup \mathbb{N}$. A *labeled hypergraph* (HG) over A is a tuple $\mathcal{H} = (V, E, src, tgt, lab, \overline{ext})$ where

- V is a finite set of vertices,
- E is a finite set of hyperedges with $V \cap E = \emptyset$,
- $src : E \to V^\star$ maps each hyperedge to a sequence of source vertices,
- $tgt : E \to V^\star$ maps each hyperedge to a sequence of target vertices,
- $lab : V \cup E \to A$ is a labeling function, and
- $\overline{ext} \in V^\star$ is a sequence of external vertices.

For each hyperedge $e \in E$, we require that the rank of its label coincides with the rank of e, which means $rk(lab(e)) = rk(e) =|src(e)| + |tgt(e)|> 0$. Furthermore, the set of all hypergraphs over A is denoted by HG_A. ■

For simplicity, we define an attachment function $att : E \to V^\star$ where $e \mapsto src(e)\cdot tgt(e)$ for each $e \in E$ which will be used whenever we are not interested in the direction of tentacles. The number of external vertices of a labeled hypergraph $\mathcal{H}$ is also called the *rank* of $\mathcal{H}$ and denoted by $rk(\mathcal{H}) = |\overline{ext_{\mathcal{H}}}|$. From now on, we do not distinguish between the terms graph and hypergraph. The same holds for edge and hyperedge.

Example 3.1.1. *Figure 3.2 shows a graphical representation of a hypergraph $\mathcal{H}$ with two hyperedges of rank 4 and 2, respectively. In our graphical notation, hyperedges are drawn as boxes with their edge label inside. External vertices are represented by red circles with the position of the vertex in $\overline{ext}$ inside. Analogously, tentacles of a hyperedge e are labeled with the index of the corresponding attached vertex in $src(e)$ and $tgt(e)$, respectively. All other vertices are represented by dots. To improve readability, a hyperedge with exactly one source vertex and exactly one target vertex may be drawn as a directed edge between both vertices without an intermediate box and without labels to denote the index of attached vertices. Furthermore, the orientation of tentacles may be dropped. In this case, tentacles are labeled with indices $1, ..., rk(e)$ according to the mapping $att(e)$.* ■

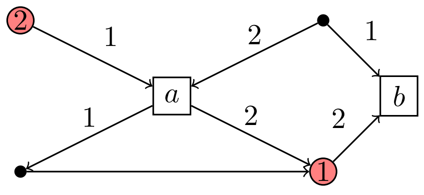


Figure 3.2.: Graphical representation of a hypergraph

For a hyperedge e of rank two with $|src(e)|=|tgt(e)|$, we also write $x \xrightarrow{a} y$ instead of $src(e) = x, tgt(e) = y$ and $lab(e) = a \in A$. Obviously, we can understand every hypergraph $\mathcal{H}$ as a labeled plain graph if we understand hyperedges as vertices. More precisely, $\mathcal{H}$ can be represented by a bipartite plain graph in which one class of vertices represents the vertices $V_{\mathcal{H}}$ and one class represents the hyperedges $E_{\mathcal{H}}$, respectively.

Definition 3.1.2 (Plain Graph Encoding)**.** Let $\mathcal{H} = (V, E, src, tgt, lab, \overline{ext}) \in HG_A$ be a labeled hypergraph and k be the maximal rank of all hyperedges in $\mathcal{H}$. The plain graph encoding $plain : HG_A \to PG_{A \times 2^{[\overline{ext}]}, \{\perp\} \cup \{1,...,k\}}$ is defined as $plain(\mathcal{H}) = (V \dot{\cup} E, F, \jmath, \ell)$, where the edge relation is defined as

$$F = \{(x,y) | y \in E, x \in [src(y)]\} \cup \{(y,x) | y \in E, x \in [tgt(y)]\}.$$

The edge labeling function is given by

$$\jmath : F \to \{\perp\} \cup \{1, ..., k\}, (x,y) \mapsto \begin{cases} j & \text{if } y \in E, x = src(y)[j] \\ j & \text{if } x \in E, y = tgt(y)[j] \\ \perp & \text{otherwise} \end{cases}$$

and the vertex labeling function is given by

$$\ell : V \cup E \rightarrow A \times 2^{[\overline{ext}]}, v \mapsto (lab(v), \{i : \overline{ext}[i] = v\}).$$

■

Note that the graphical representation of hypergraphs described in 3.1.1 corresponds to the plain graph encoding of a hypergraph where set notation for vertex labels and the label $\perp$ are left out to improve readability.

Two hypergraphs $\mathcal{H}$ and $\mathcal{G}$ are *isomorphic*, written $\mathcal{H} \cong \mathcal{G}$, if they are equal up to the names of vertices and hyperedges. For a given hypergraph $\mathcal{H}$, the set of all isomorphic hypergraphs is denoted by $\lceil \mathcal{H} \rceil$ and called an *abstract* hypergraph. Since none of the formalisms considered in this work is capable of distinguishing between isomorphic hypergraphs, we assume every set of hypergraphs to be closed under isomorphisms.

Definition 3.1.3 (Hypergraph Language)**.** A set $L \subseteq HG_A$ of hypergraphs is a *hypergraph language* if it is closed under isomorphisms, i.e. $\mathcal{H} \in L$ implies $\lceil \mathcal{H} \rceil \subseteq L$ for all $\mathcal{H} \in L$. ■

3.2. Hyperedge Replacement Grammars

In order to model dynamic data structures, we have to describe the set of all heaps corresponding to possible memory configurations of these data structures. Since a single hypergraph can model only one such memory configuration, a formalism to describe infinite sets of hypergraphs is required. In this section, we introduce graph grammars such that infinite sets of hypergraphs and heaps can be described by finite structures. Analogously to context-free string grammars or regular tree grammars, we need a set of nonterminal and a set of terminal symbols to describe a grammar generating hypergraphs. Thus, for the rest of this thesis, we assume that a finite alphabet A is partitioned into a set T of terminal symbols and a set N of nonterminal symbols, i.e. $A = N \dot{\cup} T$. We denote nonterminal symbols by lower-case Greek letters $\alpha, \beta, \gamma, ...$ in order to distinguish them from terminal symbols. A hyperedge is called nonterminal if it is labeled with a nonterminal symbol. In our graphical notation, nonterminal hyperedges will be represented by green boxes labeled with the respective nonterminal. Furthermore, we require that A is ranked by a function $rk : A \rightarrow \mathbb{N}$ and that vertices are only labeled with terminal symbols of rank 0.

The key concept to describe a grammar generating hypergraph languages is the replacement of a nonterminal hyperedge e by another hypergraph of matching rank. Intuitively, the hyperedge e is removed and every attached vertex is merged with an external vertex of the hypergraph replacing e according to the sequence $att(e) = src(e) \cdot tgt(e)$. This idea is illustrated in Figure 3.3 in which an edge e with two source vertices and two target vertices is replaced by a hypergraph $\mathcal{G}$ of rank four. In this case, the two source vertices of e are merged with the first two external vertices of $\mathcal{G}$ and the two target vertices with the last two external vertices, respectively. All other vertices and edges, including other tentacles to source and target vertices of e are preserved. Formally, we define hyperedge replacement of an edge e in a graph $\mathcal{H}$ by a graph $\mathcal{G}$ as the union of both graphs such that the sequences $att_{\mathcal{H}}(e)$ and $\overline{ext_{\mathcal{G}}}$ are merged as follows.

Definition 3.2.1 (Hyperedge Replacement)**.** Let $\mathcal{H}, \mathcal{G} \in HG_A$ be two hypergraphs with disjunct vertices and hyperedges. Furthermore, let $e \in E_{\mathcal{H}}$ be a hyperedge with $lab(e) \in N$, $|src_{\mathcal{H}}(e)| = \ell$, $|tgt_{\mathcal{H}}(e)| = k$ such that $rk(e) = rk(\mathcal{G}) = l + k$.

Then, the hypergraph $\mathcal{H}'$ obtained from replacing e by $\mathcal{G}$, written $\mathcal{H}[e/\mathcal{G}]$, is defined as $\mathcal{H}' = (V, E, src, tgt, lab, \overline{ext})$ where

- $V = V_{\mathcal{G}} \,\dot{\cup}\, V_{\mathcal{H}} \setminus ([src_{\mathcal{H}}(e)] \cup [tgt_{\mathcal{H}}(e)])$,
- $E = E_{\mathcal{G}} \,\dot{\cup}\, E_{\mathcal{H}} \setminus \{e\}$,
- $src(h) = \begin{cases} merge(src_{\mathcal{H}}(h)) & \text{if } h \in E_{\mathcal{H}} \setminus \{e\} \\ src_{\mathcal{G}}(h) & \text{if } h \in E_{\mathcal{G}} \end{cases}$
- $tgt(h) = \begin{cases} merge(tgt_{\mathcal{H}}(h)) & \text{if } h \in E_{\mathcal{H}} \setminus \{e\} \\ tgt_{\mathcal{G}}(h) & \text{if } h \in E_{\mathcal{G}} \end{cases}$
- $lab(h) = \begin{cases} lab_{\mathcal{H}}(h) & \text{if } h \in V_{\mathcal{H}} \,\dot{\cup} (E_{\mathcal{H}} \setminus \{e\}) \\ lab_{\mathcal{G}}(h) & \text{if } h \in V_{\mathcal{G}} \,\dot{\cup}\, E_{\mathcal{G}}, \text{ and} \end{cases}$
- $\overline{ext} = merge(\overline{ext_{\mathcal{H}}})$.

Here, $merge : (V_{\mathcal{G}} \,\dot{\cup}\, V_{\mathcal{H}})^\star \to V^\star$ is the application of the mapping

$$f : (V_{\mathcal{G}} \,\dot{\cup}\, V_{\mathcal{H}}) \to V, v \mapsto \begin{cases} \overline{ext_{\mathcal{G}}}[i] & \text{if } v = src_{\mathcal{H}}[i], i \in [1, l] \\ \overline{ext_{\mathcal{G}}}[l + j] & \text{if } v = tgt_{\mathcal{H}}[j], j \in [1, k] \\ v & \text{otherwise} \end{cases}$$

to every element of a given sequence of vertices merging source and target vertices of e with the corresponding external vertices of $\mathcal{G}$.

■

We now formally introduce hyperedge replacement grammars and the hypergraph languages realized by them.

Definition 3.2.2 (Hyperedge Replacement Grammar)**.** A *hyperedge replacement grammar* (HRG) is a tuple $\mathfrak{G} = (N, T, \mathfrak{P})$ where N is a set of nonterminal symbols, T is a set of terminal symbols such that $A = N \,\dot{\cup}\, T$ is ranked by a function $rk : A \to \mathbb{N}$. The last component $\mathfrak{P}$ is a set of *production rules* of the form $\mathfrak{r} = \xi \to \mathcal{X}$ where $\xi \in N$ is a nonterminal symbol and $\mathcal{X} \in HG_A$ is a hypergraph over A with $rk(\mathcal{X}) = rk(\xi)$. The set of all HRGs is denoted by $\mathcal{HRG}$. ■

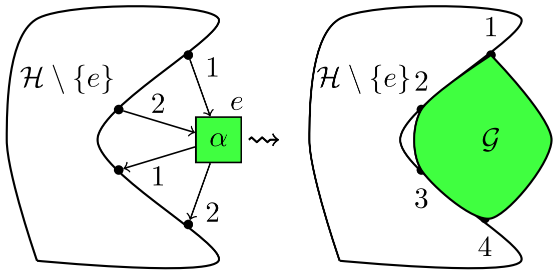


Figure 3.3.: Hyperedge replacement of an edge e by a graph $\mathcal{G}$

As the following definition shows, an HRG generates a hypergraph by stepwise replacement of nonterminal hyperedges according to its set of production rules.

Definition 3.2.3 (HRG Derivation)**.** Let $\mathfrak{G} = (N, T, \mathfrak{P})$ be an HRG, $\mathcal{H} \in HG_{N \dot{\cup} T}$ be a hypergraph and $\mathfrak{x} = \xi \rightarrow \mathcal{X} \in \mathfrak{P}$ be a production rule. We say that $\mathcal{H}$ *derives* $\mathcal{H}'$ by $\mathfrak{x}$, written $\mathcal{H} \overset{\mathfrak{a}}{\Longrightarrow}_{\mathfrak{G}} \mathcal{H}'$, if there exists a hyperedge $e \in E_{\mathcal{H}}$ with $lab_{\mathcal{H}}(e) = \xi$ such that $\mathcal{H}' \cong \mathcal{H}[e/\mathcal{X}]$. The transitive closure of this relation for all production rules of $\mathfrak{G}$ is denoted by $\Longrightarrow^{\star}_{\mathfrak{G}}$. Moreover, we write $\mathcal{H} \overset{n}{\Longrightarrow}_{\mathfrak{G}} \mathcal{H}'$ for $n \in \mathbb{N}$ to denote that $\mathcal{H}'$ can be derived from $\mathcal{H}$ using at most n derivation steps. ■

We call a production rule $\mathfrak{a} = \alpha \rightarrow \mathcal{A}$ *terminal* if it maps to a hypergraph $\mathcal{A}$ which contains no nonterminal hyperedges. Note that our definition of HRGs is not equipped with a starting symbol as it is common for context-free string grammars, but we can start derivations for any hypergraph or set of hypergraphs over a fitting alphabet. For simplicity, however, we usually start with a special hypergraph which is close to a starting symbol of a context-free string grammar.

Definition 3.2.4 (Handle)**.** For a nonterminal symbol $\xi \in N$ with $rk(\xi) = n$, the corresponding *handle* $\xi^{\bullet}$ is defined as the hypergraph

$$\xi^{\bullet} = (v_1...v_n, e, \{e \mapsto v_1...v_n\}, \{e \mapsto \varepsilon\}, \{e \mapsto \xi\}, \varepsilon).$$

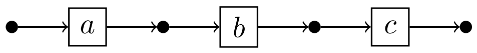
■

Thus, a handle consists of a single nonterminal hyperedge e with with $rk(e)$ many source vertices. Note that any other choice of attachments with $att(e) = v_1...v_n$ could be used as an initial hypergraph. For simplicity, however, we choose a fixed attachment $src(e) = v_1...v_n$ in order to obtain unique handles. Now, the language generated by an HRG is the set of all hypergraphs without nonterminal edges that can be derived from a given handle.

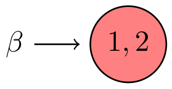
Definition 3.2.5 (HR-Languages)**.** Let $\mathfrak{G} = (N, T, \mathfrak{P})$ be an HRG. The HR-language generated by $\mathfrak{G}$ with initial nonterminal $\alpha \in N$ is defined as $L(\mathfrak{G}, \alpha) = \{\lceil \mathcal{H} \rceil \in HG_T \mid \alpha^{\bullet} \Longrightarrow^{\star}_{\mathfrak{G}} \mathcal{H}\}$. A language is called an *HR-language*, if there exists an HRG generating it. The class of all HR-languages is denoted by $\mathcal{HRL}$. ■

Two HRGs $\mathfrak{G}_1, \mathfrak{G}_2$ are called *equivalent* if their languages are equal for all starting symbols, i.e. $L(\mathfrak{G}_1, \alpha) = L(\mathfrak{G}_2, \alpha)$ for all $\alpha \in N$. Before we discuss further properties of HRGs, we consider some examples.

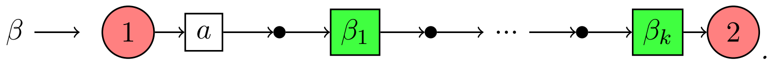
Example 3.2.1 (Context-Free String Languages)**.** *Obviously, every finite word w over an alphabet A can be uniquely represented by a hypergraph $graph(w)$. For instance, $graph(abc)$ is given by the following hypergraph. For every context-free string lan-*



guage $L \subseteq A^\star$, there exists an HRG $\mathfrak{G}$ and a nonterminal α such that $L(\mathfrak{G}, \alpha) = \{graph(w) \mid w \in L\}$. To see this, let $\mathfrak{G} = (N, T, \mathfrak{P}, \alpha)$ be a context-free string grammar in Greibach normal form generating L. We construct a suitable HRG $\mathfrak{G}' = (N, T, \mathfrak{P}')$ as follows. For each production rule $\beta \to \varepsilon \in \mathfrak{P}$, $\mathfrak{P}'$ contains a production rule of the form



and for each production rule $\beta \to a\beta_1 \ldots \beta_k \in \mathfrak{P}$ with $\beta_1, \ldots, \beta_k \in N$, $\mathfrak{P}'$ contains a production rule



Then, it is straightforward to see that $L(\mathfrak{G}', \alpha) = \{graph(w) \mid w \in L\}$. ■

For context-free string languages, it is well-known that the language inclusion problem, i.e. the question whether $L \subseteq L'$ for two context-free string languages L and L', is undecidable. Thus, as an immediate consequence of Example 3.2.1 we observe that

Proposition 3.2.1. *The language inclusion problem for HR-languages is undecidable.* ■

Furthermore, Example 3.2.1 shows an HRG with a production rule in which a single vertex corresponds to multiple external vertices. Intuitively, this enables an HRG to delete vertices and edges between these vertices. Although this behavior can be useful to define HR-languages as the one above, it often complicates proofs if a single vertex of a hypergraph is allowed to correspond to multiple external vertices or multiple attached vertices of the same nonterminal hyperedge. Fortunately, a theorem by Habel shows that this is never required to realize an HR-language.

Proposition 3.2.2 (Well-Formedness Theorem, [Hab92])**.** *For every HRG an equivalent HRG in which no production rule contains a vertex which corresponds to multiple external vertices or multiple attached vertices of the same hyperedge can be constructed.* ■

A consequence of the well-formedness theorem is that every HR-language can be generated by a *monotone* HRG, i.e. an HRG in which the size of the generated hypergraph increases with every derivation step [Hab92]. From now on, we assume that an HRG is well-formed in the sense of Proposition 3.2.2 if not explicitly stated otherwise. As another example, we consider regular tree grammars (cf. [SR97] chapter 1 or [CDG$^+$07]), which can be understood as a special case of hyperedge replacement grammars.

Definition 3.2.6 (Regular Tree Grammar)**.** A *regular tree grammar*, is an HRG $\mathfrak{G} = (N, T, \mathfrak{P})$ with an initial nonterminal $\xi \in N$ such that for every production rule $\mathfrak{a} = \alpha \to \mathcal{A} \in \mathfrak{P}$, $\mathcal{A}$ is a tree with exactly one external vertex, the root, and every hyperedge $e \in E_{\mathcal{A}}$ is of rank 1 and attached to a leaf. Furthermore, every leaf in $\mathcal{A}$ is attached to at most one hyperedge. ■

The class of trees realizable by regular tree grammars is known as the class of *regular tree languages*. This class is a natural extension of regular string languages to tree languages and preserves some nice properties, like closure under Boolean operations and, in contrast to general HR-languages, decidability of the inclusion problem.

Example 3.2.2 (Rooted Binary Trees with Linked Leaves)**.** *A more complex example of a data structure modeled by HRGs is provided in Figure 3.4. This HRG $\mathfrak{G}$ generates (complete) binary trees in which each vertex is additionally linked to its parent and all leaves are connected by a singly-linked list from left to right. In order to keep the HRG readable, we allow the root and the rightmost leaf to be linked to some vertex outside the tree instead of the parent and the next leaf, respectively. Note that the third and the fourth external vertex in the first production rule are used as "forward references" to denote the leftmost leaf of this and the next subtree. A possible derivation of $\mathfrak{G}$ is shown in Figure 3.5. This derivation starts with the handle $\alpha^\bullet$ and applies the left production rule two times (first to the handle, second to the left subtree of the resulting hypergraph). Then, the right production rule is applied to the remaining nonterminals resulting in a rooted tree with links to parents and connected leaves without nonterminal hyperedges. Thus, the resulting hypergraph is in $L(\mathfrak{G}, \alpha)$. For simplicity, the labels p, l, r, n of selector edges denoting parent vertex, left and right child and the next leaf are omitted.* ■

An important property of HRGs, which justifies the term context-free graph grammar used by Engelfriet and Courcelle ([SR97], [Cou91], [CE12]), is that an HRG derivation can be decomposed into independent derivation steps just as context-free string grammars.

Proposition 3.2.3 (Decomposition Theorem [Hab92])**.** *Let $\mathfrak{G}$ be an HRG, α a nonterminal and $\mathcal{H}$ a hypergraph. Then, $\alpha^\bullet \overset{k+1}{\Longrightarrow}_{\mathfrak{G}} \mathcal{H}$ for some $k \geq 0$ if and only if there exists a hypergraph $\mathcal{F}$ such that $\alpha^\bullet \overset{1}{\Longrightarrow}_{\mathfrak{G}} \mathcal{F}$ and for each nonterminal hyperedge $e \in E_{\mathcal{F}} = \{e_1 ... e_n\}$ there exists a derivation $e^\bullet \overset{k}{\Longrightarrow}_{\mathfrak{G}} \mathcal{H}_e$ such that $\mathcal{H} = \mathcal{F}[e_1/\mathcal{H}_{e_1}, ..., e_n/\mathcal{H}_{e_n}]$.* ■

Furthermore, the decomposition theorem explains how derivation trees can be constructed for HRGs. The following definition of derivation trees is based on [Hab92].

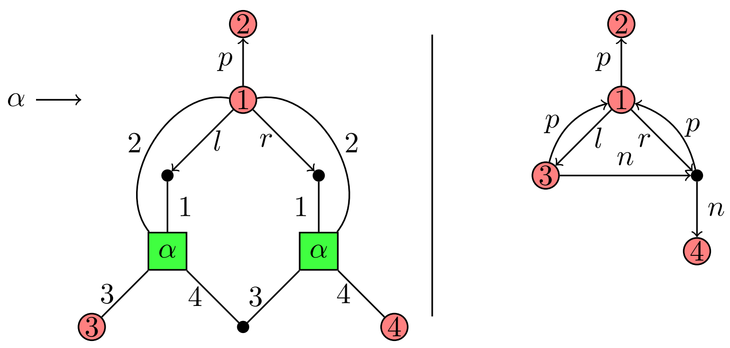


Figure 3.4.: HRG generating binary trees with linked leaves

Note that alternative versions of derivation trees have been proposed in the literature, especially in [SR97] and [Cou91].

Definition 3.2.7 (Derivation Tree)**.** Let $\mathfrak{G} = (N, T, \mathfrak{P})$ be an HRG and $prk : \mathfrak{P} \to \mathbb{N}$ be a function assigning the number of nonterminals occurring on the right-hand side to every production rule, i.e. $prk(\mathfrak{a}) = |\{e \in E_\mathfrak{a} \mid lab(e) \in N\}|$ for every $\mathfrak{a} = \alpha \to \mathcal{A} \in \mathfrak{P}$. The set of all *derivation trees* of $\mathfrak{G}$ with initial nonterminal $\xi \in N$, written $\mathfrak{T}_\mathfrak{G}(\xi)$, is the least set of trees over the ranked alphabet $\mathfrak{P}$ (with ranking function prk) which can be obtained according to the following inductive definition:

- If $\mathfrak{x} = \xi \to \mathcal{X} \in \mathfrak{P}$ is a production rule with $prk(\mathfrak{x}) = 0$, the tree $t = \mathfrak{x}$ consisting of a single vertex labeled with $\mathfrak{x}$ is a derivation tree in $\mathfrak{T}_\mathfrak{G}(\xi)$.
- If $\mathfrak{x} = \xi \to \mathcal{X} \in \mathfrak{P}$ is a production rule with $prk(\mathfrak{x}) = n$ and $e_1, ..., e_n$ are the nonterminal hyperedges in $\mathcal{X}$ labeled with $\alpha_1, ..., \alpha_n$, then the tree $t = \mathfrak{x}(t_1, ..., t_n)$ is a derivation tree in $\mathfrak{T}_\mathfrak{G}(\xi)$ for any selection of derivation trees $t_1, ..., t_n$ in $\mathfrak{T}_\mathfrak{G}(\alpha_1)$,...,$\mathfrak{T}_\mathfrak{G}(\alpha_n)$.

■

As an example, consider the derivation tree in Figure 3.6 where $\mathfrak{a}_1$ denotes the left and $\mathfrak{a}_2$ denotes the right production rule of the grammar shown in Figure 3.4, respectively. This derivation tree corresponds to the example derivation shown in Figure 3.5. The derivation tree in this example defines a partial order of derivation steps. For instance, in order to obtain the final hypergraph shown in Figure 3.5, the left production rule $\mathfrak{a}_1$ must be applied first. However, it does not matter whether the derivation continues by replacing the left or the right nonterminal first, because the two children of the root are not ordered.

In general, Habel showed that every derivation respecting the partial order induced by a derivation tree yields the same hypergraph and for every derivation tree there exists a corresponding derivation and vice versa [Hab92]. Although a large class of interesting hypergraphs can be generated by HRGs, there are some limitations. Probably the most

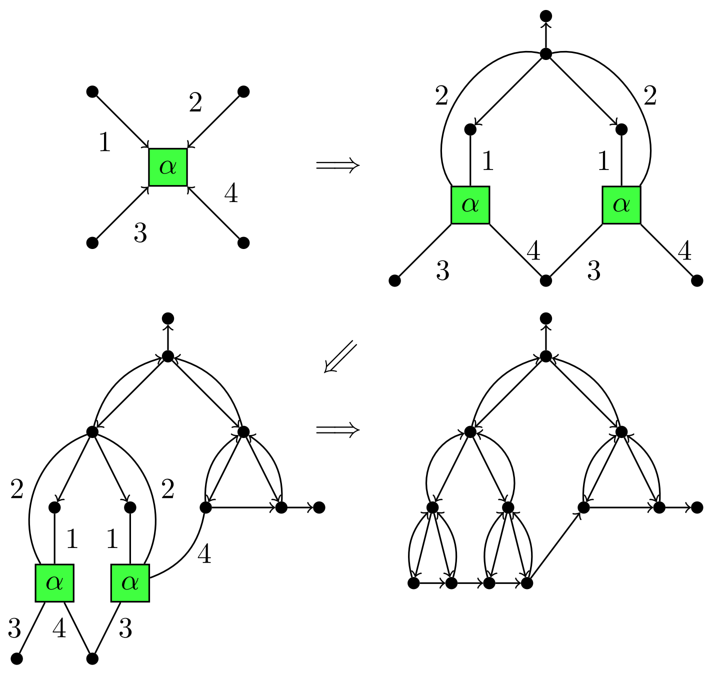


Figure 3.5.: Example derivation of HRG presented in Example 3.2.2

important one is that all hypergraphs which can be generated by HRGs have bounded tree width ([Hab92], [Cou90], [SR97], [CE12]). Intuitively, tree width is a complexity measure for graphs indicating how similar a graph is to a tree (cf. [RS84]). For example, every complete graph consisting of n vertices has tree width $n-1$, i.e. the class of all complete graphs does not have bounded tree width. Furthermore, the classes of all graphs, bipartite graphs and square grids do not have bounded tree width and therefore cannot be generated by an HRG [Hab92]. Finally, we note that HRGs are unable to count and therefore cannot generate hypergraph languages like the class of all balanced trees, i.e. trees in which the height of subtrees differs at most by one [KS94].

3.3. Data Structure Grammars

Hyperedge replacement grammars are a more general formalism than actually needed to model sets of heaps and can generate hypergraphs which do not correspond to any heap. For instance, a vertex may have multiple outgoing edges labeled with the same selector which is clearly impossible for any heap. Thus, in compliance with [JGN14], we restrict the class of considered hypergraphs to heap configurations.

Definition 3.3.1 (Heap Configuration)**.** Let $A = N \dot{\cup} T$ be an alphabet. A hypergraph $\mathcal{H} \in HG_A$ is a *heap configuration* if

1. there are no external vertices, i.e. $\overline{ext_{\mathcal{H}}} = \varepsilon$, and

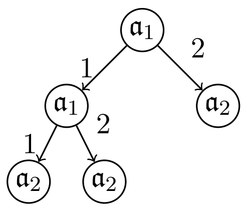


Figure 3.6.: Example of a derivation tree corresponding to Figure 3.5

2. for every vertex $v \in V_{\mathcal{H}}$ and every $l \in A$, there is at most one hyperedge $e \in E_{\mathcal{H}}$ with $lab_{\mathcal{H}}(e) = l$ and $v \in [src_{\mathcal{H}}(e)]$.

We denote the set of all heap configurations over A by HC_A. ■

Then, whenever we model data structures with HRGs, hypergraph languages consisting only of heap configurations are considered. The corresponding class of HRGs that derives heap configurations only is called the class of data structure grammars. Note that all examples in the previous section belong to this class.

Definition 3.3.2 (Data Structure Grammar)**.** An HRG $\mathfrak{G} \in \mathcal{HRG}$ over an alphabet $A = N \dot{\cup} T$ is called a *data structure grammar* if $L(\mathfrak{G}, \xi) \subseteq HC_A$ for all nonterminals $\xi \in N$. The set of all hypergraph languages realizable by DSGs is denoted by $\mathcal{DSL}$. ■

CHAPTER 4

Logical Descriptions of Heaps

We discussed how sets of heaps can be represented by data structure grammars in Chapter 3. Graphical descriptions of heaps typically contain more information than just the final set of heaps belonging to a dynamic data structure. For instance, the production rules of a data structure grammar also specify how heaps are constructed step-by-step. Thus, DSGs can be seen as a formalism which is close to an implementation of a data structure. A more declarative approach are logical specification languages. In contrast to DSGs, logical formalisms abstract from the way heaps are constructed and specify only the properties every heap in a language must have. This makes logical formalisms well-suited to specify properties of data structures for verification. For example, a property like "every heap corresponds to a connected graph" is hard to specify by data structure grammars, because production rules to derive every possible connected graph are required. However, it is straightforward to define this property by a formula in an appropriate logical language. Intuitively, such a formula requires the existence of an undirected path from every location in a heap to every other location.

In this chapter, two logical languages are considered. At first, we show how properties of heaps can be specified in *monadic second-order logic* (MSO). Similar to HRGs extending context-free string grammars, this approach can be viewed as an extension of regular string languages to hypergraph languages. The theoretical background is a famous theorem by Büchi which states that the class of regular string languages can be characterized by monadic second-order logic formulae over an appropriate model of strings [Büc60]. We present monadic second-order logic for arbitrary structures and then show how languages of strings, trees and hypergraphs can be defined in MSO. Furthermore, we report on decidability and complexity results for MSO. These results are due to Engelfriet and Courcelle who extensively studied monadic second-order logic over hypergraph models in order to find a "regular" fragment of hypergraph languages [Cou90], [Cou91], [SR97], [CE12].

In addition to that, we consider *separation logic* which is an extension of Hoare logic (cf. [Hoa69]) introduced by Reynolds specifically designed to describe heaps ([Rey00], [Rey02]). The practical relevance of separation logic stems from the principle of local reasoning such that disjunct parts in the heap can be verified independently. Automated reasoning on separation logic formulae is supported by a wide range of tools like Heap-

Hop [VLC10], jStar [DP08], Predator [DPV11], Smallfoot [BCO06], Thor [MTLT08], SpaceInvader [BCC+07], Xisa [CR08] and Infer [CD11]. Most of these tools add constraints to separation logic in order to make automatic reasoning practically feasible. For instance, SpaceInvader and Infer restrict recursion such that only list-like structures can be specified. We report on fragments of separation logic equipped with more general recursive definitions and discuss the relationship between separation logic and data structure grammars based on results by Dodds [Dod08] and Jansen et al. [JGN14].

4.1. Monadic Second-Order Logic over Graphs

Monadic second-order logic (MSO) is an extension of classical predicate logic in which variables may additionally range over sets of objects. As a result of this, statements about the transitive closure of (binary) relations can be defined in MSO, i.e. properties like connectedness of graphs can be formulated in this logical language. Monadic second-order logic is not one logic, but a logical framework in which the expressive power depends on the domain and available relations of the structure under consideration. In our case, we want to define structures and relations such that heaps and hypergraphs can be specified in MSO. However, we first define MSO in terms of general relational structures and provide some example structures from formal language theory in order to stress the close relationship between monadic-second order logic and regular languages. Then, these example structures are extended to hypergraph models. The relations available in an instance of MSO logic, are collected in a set S, called the *signature*, containing relational symbols of fixed arity, i.e. S is ranked. For a given signature S, objects in the domain of an instance of MSO logic must specify relations for each symbol $R \in S$. These objects are called S-structures.

Definition 4.1.1 (Relational S-Structure)**.** An S-structure is a pair $\underline{\mathcal{A}} = (A, g)$ where A is a non-empty set of objects called the *universe* of $\underline{\mathcal{A}}$ and g is a mapping such that $g(R)$ is an n-ary relation over A for each n-ary relational symbol R in S. Especially, for $n = 0$, $g(R)$ is called a *constant*. We denote the set of all structures over S by $\mathfrak{S}[S]$. ■

For convenience, we usually write $R^{\underline{\mathcal{A}}}$ or just R instead of $g(R)$ whenever the meaning is clear from the context and $R(x_1, ..., x_n)$ instead of $(x_1, .., x_n) \in R$. Furthermore, if $S = \{R_1, ..., R_n\}$ is a finite signature, $\underline{\mathcal{A}} = (A, R_1^{\underline{\mathcal{A}}}, ..., R_n^{\underline{\mathcal{A}}})$ denotes the relational structure $\underline{\mathcal{A}} = (A, g)$ with $g = [R_1 \mapsto R_1^{\underline{\mathcal{A}}}, ..., R_n \mapsto R_n^{\underline{\mathcal{A}}}]$. The syntax of MSO is composed of standard logical operators, variables representing single objects (called first-order variables), variables representing sets and relational symbols from a given signature. Formally, let $FVar = \{x, y, z, ...\}$ be a countable set of first-order variable names and $SVar = \{X, Y, Z, ...\}$ be a countable set of set variable names, respectively.

Definition 4.1.2 (Syntax of Monadic Second-Order Logic)**.** Let S be a signature of relational symbols. The set of all MSO formulae over S, denoted by $MSOF[S]$, is defined by structural induction as follows. The atomic formulae of $MSOF[S]$ are

- the formula *true* which holds for every S-structure,

- relational symbols, i.e. $R(x_1, ..., x_n)$ is a formula for every symbol $R \in S$ of arity $n \geq 0$,
- equality of first-order variables, i.e. $x = y$ for $x, y \in FVar$,
- membership to set variables, i.e. $x \in Y$ for $x \in FVar, Y \in SVar$, and
- inclusion of set variables, i.e. $X \subseteq Y$ for $X, Y \in SVar$.

Furthermore, if φ and ψ are in $MSOF[S]$, then

- the negation $\neg\varphi$,
- the disjunction $\varphi \vee \psi$, and
- the existential quantifications $\exists x.\varphi$ and $\exists X.\varphi$ where x and X are variables occurring in φ

are in $MSOF[S]$. ■

We usually denote MSO formulae by Greek letters $\varphi, \psi, \vartheta, ...$ or short textual descriptions. A variable is called *free* in a formula φ if it is not introduced by a quantifier $\exists$ inside of φ. Analogously to the parameters of relational symbols, we write $\varphi(\overline{X}, \overline{x})$ to denote that exactly the variables in the sequences $\overline{X}$ and $\overline{x}$ occur free in φ. If a formula has no free variables it is also called a *sentence*.

So far, a formula in $MSOF[S]$ is just a syntactic construct. The meaning of such a formula and its variables depends on the S-structure that is used to interpret it. Formally, for a given S-structure $\underline{A} = (A, g)$, an *interpretation* is a partial mapping $\imath : FVar \,\dot{\cup}\, SVar \rightharpoonup A \,\dot{\cup}\, 2^A$ such that for each first-order variable $x \in FVar$, we have $\imath(x) \in A$ and for each set variable $X \in SVar$, we have $\imath(X) \in 2^A$, respectively. Thus, an interpretation assigns an object (or a set of objects) from the universe to each variable occurring in a formula. An interpretation is called *compatible* with a formula φ if its domain ranges over all variables occurring in φ. We define the semantics of $MSOF[S]$ formulae in terms of a binary relation "$\models$" that specifies for a given S-structure $\underline{A}$ and a formula $\varphi(\bar{X}, \bar{x})$ whether $\underline{A}$ *satisfies* or *is a model of* $\varphi(\bar{X}, \bar{x})$.

Definition 4.1.3 (Semantics of $MSOF[S]$ Formulae)**.** Let S be a signature and $\underline{A} \in \mathfrak{S}[S]$ a structure over S. For a compatible interpretation $\imath$ and $MSOF[S]$ formulae $\varphi(\overline{X}, \overline{x})$ and $\psi(\overline{Y}, \overline{y})$, the *model relation* $\models^{\imath} \subseteq \mathfrak{S}[S] \times MSOF[S]$ is defined inductively as follows.

$$\begin{aligned}
\underline{\mathcal{A}} &\models^{\imath} \mathit{true} \\
\underline{\mathcal{A}} &\models^{\imath} R(\overline{X}, \overline{x}) && \text{iff } (\imath(\overline{X}), \imath(\overline{x})) \in R^{\underline{\mathcal{A}}} \text{ for each } R \in S \\
\underline{\mathcal{A}} &\models^{\imath} x = y && \text{iff } \imath(x) = \imath(y) \\
\underline{\mathcal{A}} &\models^{\imath} x \in Y && \text{iff } \imath(x) \in \imath(Y) \\
\underline{\mathcal{A}} &\models^{\imath} X \subseteq Y && \text{iff } \imath(X) \subseteq \imath(Y) \\
\underline{\mathcal{A}} &\models^{\imath} \neg\varphi(\overline{X}, \overline{x}) && \text{iff not } \underline{\mathcal{A}} \models^{\imath} \varphi(\overline{X}, \overline{x}) \\
\underline{\mathcal{A}} &\models^{\imath} \varphi(\overline{X}, \overline{x}) \vee \psi(\overline{Y}, \overline{y}) && \text{iff } \underline{\mathcal{A}} \models^{\imath} \varphi(\overline{X}, \overline{x}) \text{ or } \underline{\mathcal{A}} \models^{\imath} \psi(\overline{Y}, \overline{y}) \\
\underline{\mathcal{A}} &\models^{\imath} \exists x.\varphi(\overline{X}, \overline{x}) && \text{iff exists } a \in A \text{ such that } \underline{\mathcal{A}} \models^{\imath[x \mapsto a]} \varphi(\overline{X}, \overline{x}) \\
\underline{\mathcal{A}} &\models^{\imath} \exists X.\varphi(\overline{X}, \overline{x}) && \text{iff exists } M \subseteq A \text{ such that } \underline{\mathcal{A}} \models^{\imath[X \mapsto M]} \varphi(\overline{X}, \overline{x})
\end{aligned}$$

Furthermore, we say that $\underline{\mathcal{A}}$ *is a model of* $\varphi(\overline{X}, \overline{x})$, written $\underline{\mathcal{A}} \models \varphi(\overline{X}, \overline{x})$, if there exists a compatible interpretation $\imath$ such that $\underline{\mathcal{A}} \models^{\imath} \varphi(\overline{X}, \overline{x})$. ■

It is straightforward to show that common logical operators like inequality $x \neq y$, conjunction $\varphi \wedge \psi$, implication $\varphi \rightarrow \psi$, equivalence $\varphi \leftrightarrow \psi$ and universal quantification $\forall x.\varphi(x)$ can be defined in $MSOF[S]$ for any signature S. The corresponding proofs are left to the reader. A *first-order formula* is an $MSOF[S]$ formula φ which does not contain any set variables.

Let $C \subseteq \mathfrak{S}[S]$ be a class of S-structures. With the help of the model relation, we can also define the *language* specified by an $MSOF[S]$ formula φ in C as the set of all structures in C that are models of φ, i.e. $L(\varphi) = \{\underline{\mathcal{A}} \in C \mid \underline{\mathcal{A}} \models \varphi\}$. Analogously to HRGs, we call two formulae φ and ψ *equivalent*, written $\varphi \equiv \psi$, if their languages are the same and say that a language $L \subseteq C$ of S-structures is *definable* in $MSOF[S]$ if there exists a sentence φ such that $L(\varphi) = L$. The set of all languages $L \subseteq C$ definable in $MSOF[S]$ is denoted by $MSOL[S, C]$.

In the context of MSO logic, we consider three decision problems which coincide with the emptiness problem, the word problem and the inclusion problem for languages over the same set of structures. Formally, let φ be an $MSOF[S]$ sentence.

1. The *satisfiability problem* is the question whether there exists a model $\underline{\mathcal{A}} \in C$ of φ, i.e. whether $L(\varphi) \neq \emptyset$.

2. The *model-checking problem* is the question whether a given structure $\underline{\mathcal{A}} \in C$ is a model of φ, i.e. whether $\underline{\mathcal{A}} \in L(\varphi)$.

3. The *entailment problem* is the question whether all models in C of a sentence φ are also models of a sentence ψ, i.e. whether $L(\varphi) \subseteq L(\psi)$.

Note that decidability of the satisfiability problem already implies decidability of the entailment problem due to the fact that $MSOF[S]$ is closed under Boolean operations and the sentence $\varphi \rightarrow \psi$ is satisfiable if and only if all models of φ are also models of ψ.

So far, we introduced MSO as an abstract logical framework that depends on the signature and structures under consideration. Now, we consider some examples of MSO logic from formal language theory and discuss the decision problems from above for these concrete instances of MSO. These examples lead the way towards a definition of structures and signatures to define hypergraph languages in MSO.

Example 4.1.1 (Regular String Languages). *A classical result for MSO is that the class of regular string languages can be characterized in MSO for an appropriate class of structures. Let $\overline{w} = w_1...w_n \in A^\star$ be a string over an alphabet A.*

In order to describe finite words by structures, we employ the signature $S = \{succ\} \dot\cup \{lab_a \mid a \in A\}$. Here, succ is a binary relational symbol that denotes the successor relation between positions in a string. Every relation lab_a represents a set containing all positions in a string that are labeled with the symbol a. The corresponding S-structure or word model *consists of all positions in the string $\overline{w}$ and defines the relations succ and lab_a according to the intuition from above. Formally, the word model of $\overline{w}$ is defined as $\underline{w} = (\{1, ..., |w|\}, succ^w, (lab_a^w)_{a\in A})$ consisting of a universe of all positions in the word, a successor relation succ with $(i, j) \in succ^w$ iff $j = i + 1$ and $1 \leq i <|w|$ and the label of each position given by sets $lab_a = \{i \in \{1, ..., n\} \mid w_i = a\}$ for each $a \in A$. Let $\underline{A^\star}$ denote the set of all word models over A.*

A famous theorem by Büchi states that for every language L of finite strings, there exists an $MSOF[S]$ sentence φ (interpreted over $\underline{A^\star}$) such that $L = L(\varphi)$ if and only if L is a regular string language, i.e. L can be represented by a finite state automaton [Büc60]. For example, the formula $\forall x(lab_a(x) \to \exists y(succ(x, y) \wedge lab_b(y)))$ defines the language of all strings in which every position labeled with a is followed by a position labeled with b. Since emptiness, inclusion and the word problem are decidable for finite automata over words, it directly follows that these problems are decidable for $MSOL[S, \underline{A^\star}]$, too. ■

Büchi's theorem can also be lifted to regular tree languages (see Definition 3.2.6) for an appropriate structure of trees.

Example 4.1.2 (Regular Tree Languages). *Let t be a finite tree over a ranked alphabet A with maximal rank k. As for strings, every position in a tree is labeled with some symbol from A, i.e. we again use a set $\{lab_a \mid a \in A\}$ of sets to represent the labeling in the signature. However, a vertex in a tree may have more than one successor. Thus, k successor relations, one for each possible child, are added to the signature S, i.e. formally we have $S = \{succ_i \mid 1 \leq i \leq k\} \dot\cup \{lab_a \mid a \in A\}$ This change is also reflected in the* tree model *corresponding to t which is defined as the S-structure $\underline{t} = (dom(t), succ_1^t, ..., succ_n^t, (lab_a^t)_{a\in A})$ where $(x, y) \in succ_i^t$ iff $y = x.i \in dom(t)$. The sets lab_a are defined analogously to the string case. Let $\underline{Trees_A}$ denote the set of all tree models over A. As an example, consider the formula $\exists x.\forall y \bigwedge_{1\leq i\leq k} \neg succ_i(y, x)$ which asserts that x represents a position in a tree that is not the child of any other position, i.e. x represents the root of a tree.*

Doner showed that Büchi's theorem for regular string languages also holds for regular tree languages and tree models, i.e. for each language T of finite trees, there exists an $MSOF[S]$ sentence φ (interpreted over tree models only) with $L(\varphi) = T$ if and only if T

is a regular tree language, i.e. there exists a finite tree automaton accepting T [Don70]. Again, the decidability of entailment, model-checking and satisfiability directly follows from the decidability of the respective problems for finite tree automata (cf. [CDG+ 07]). ■

The following result on the complexity of the satisfiability problem is due to Meyer and holds for MSO over word models as well as MSO over tree models [Mey74].

Proposition 4.1.1 (Meyer, 1974)**.** *For any algorithm deciding the satisfiability problem for*

- *MSO formulae over word models,*
- *MSO formulae over tree models, or*
- *MSO formulae over models representing linear orders,*

there exists no constant $k \in \mathbb{N}$ such that the number of steps required by the algorithm can be bound by a function of the form $2^{2^{\cdot^{\cdot^{\cdot}}}}\}k - times$. ■

Thus, even if the satisfiability problem is decidable for MSO over a certain class of structures, no efficient algorithms exist to solve this problem. Especially with an increasing number of variables, the satisfiability problem quickly becomes practically infeasible. Nonetheless, tools like Mona [HJJ+95] exist to solve the satisfiability problem for MSO and turn out to work well for small MSO formulae.

The previous two examples show the close relationship between MSO logic, automata and regular languages with all their nice properties like closure under Boolean operations and decidability of the emptiness and language inclusion problem. This raises the question whether this relationship can also be lifted to hypergraph languages, i.e. is there a notion of "regular" hypergraph languages? Furthermore, the relationship between such a notion and HRGs is of interest, because HRGs are a natural extension of context-free string grammars to hypergraph languages. For the special case of regular tree grammars, this relation is already clarified by Example 4.1.2. In the general case, we need a proper model of a labeled hypergraph $\mathcal{H} = (V, E, src, tgt, lab, \overline{ext})$. At least two different versions of graph models are common in the literature (cf. [CE12]). The first is based on the adjacency matrix of a graph, which means the domain consists only of vertices and edges between vertices are represented by binary relational symbols. This approach is a natural extension of word models and tree models in which these edge relations are called successor relations. However, edges cannot be directly represented by variables with this approach. Thus, it is difficult to describe hypergraphs in which a hyperedge is attached to many different vertices. A possible workaround is to consider the plain graph encoding (see Definition 3.1.2) of a hypergraph as the basis for a hypergraph model. The alternative to this approach, which will be used in this thesis, is based on the incidence matrix of a hypergraph and allows hyperedges to be part of the domain and therefore to be directly represented by variables.

Definition 4.1.4 (Hypergraph Model)**.** Let $\mathcal{H} = (V, E, src, tgt, lab, \overline{ext})$ be a hypergraph over an alphabet A. The *hypergraph model* corresponding to $\mathcal{H}$ is defined as the relational structure

$$\underline{\mathcal{H}} = (V \,\dot{\cup}\, E, source^{\mathcal{H}}, target^{\mathcal{H}}, (lab_a^{\mathcal{H}})_{a \in A}) \text{ where}$$

$source^{\mathcal{H}}$ is a binary relation corresponding to tentacles of source vertices, i.e. $(v, e) \in source^{\mathcal{H}}$ iff $v \in [src(e)]$, and $target^{\mathcal{H}}$ is a binary relation corresponding to tentacles of target vertices, i.e. $(e, v) \in target^{\mathcal{H}}$ iff $v \in [tgt(e)]$. The labeling is defined analogously to word and tree models, i.e. $x \in lab_a^{\mathcal{H}}$ iff $lab(x) = a$. ■

In our hypergraph model, external vertices are not represented, but can easily be added by another unary relation similar to the labeling sets. In the literature, MSO over hypergraph models using the adjacency matrix is often denoted by MSO_1F and MSO over hypergraph models as defined in Definition 4.1.4 is denoted by MSO_2F, respectively [CE12], [SR97]. Note that the choice of a hypergraph model is not just a matter of taste, but leads to logics of different expressiveness. Obviously, every property expressed in MSO using hypergraph models based on the adjacency matrix are also expressible in MSO_2F. However, Courcelle showed that a property like "every loop-free graph has a perfect matching" can be defined in MSO_2F, but not in MSO_1F. Before we discuss decidability problems for MSO_2F, we consider some examples. Obviously, we can define a formula specifying that there exists a directed connection via a single hyperedge between two vertices, e.g. using the formula

$$x \circ\!\!\rightarrow\!\!\circ\, y \triangleq \exists e(source(x, e) \wedge target(e, y)).$$

Here, $\triangleq$ is to be read "is defined as" to avoid confusion with the equality operator in MSO. We write $x \circ\!\!\rightarrow\!\!\circ\, y$ instead of $\varphi(x, y)$ to denote this formula to improve readability in more complex formulae. Analogously, the formula $x \circ\!\!-\!\!\circ\, y \triangleq x \circ\!\!\rightarrow\!\!\circ\, y \vee y \circ\!\!\rightarrow\!\!\circ\, x$ denotes that x and y are connected via a hyperedge.

Example 4.1.3. *A prominent application of MSO formulae is to define transitive closures of binary relations (see for example [CE12] or the original paper by Thatcher and Wright [TW68]). Let $\varphi(x, y)$ be an $MSOF[S]$ formula defining a binary relation $R^{\underline{A}} = \{(u, v) \mid \underline{A} \models \exists u \exists v \varphi(u, v)\}$ for a suited structure $\underline{A}$ over S. Then, a pair (u, v) lies in the transitive closure of R if every set containing u and closed under R also contains v. More formally, the transitive closure of R can be defined in $MSOF[S]$ by a formula $closure_\varphi(x, y)$ as follows.*

$$closed_\varphi(X) \triangleq \forall u \forall v(u \in X \wedge \varphi(u, v) \rightarrow v \in X)$$
$$closure_\varphi(x, y) \triangleq \forall X(x \in X \wedge closed_\varphi(X) \rightarrow y \in X)$$

Note that $closed_\varphi(X)$ is always satisfied if X represents the empty set or all vertices and edges of a hypergraph. Thus, the use of a universal quantification over all sets X is required. For instance, the transitive closure of $\varphi(x, y) \triangleq x \circ\!\!\rightarrow\!\!\circ\, y$ specifies all pairs of *vertices (u, v) such that there exists a directed path from u to v. Other applications of the transitive closure of binary relations are connectedness of graphs and the existence of cycles.* ■

According to Definition 3.1.1, every hyperedge is connected to at least one vertex. Furthermore, the first parameter of $source(x, y)$ is required to represent a vertex and the second parameter is required to represent a hyperedge, respectively. The converse holds for $target(x, y)$. Thus, the formula $\varphi(x) \triangleq \neg\exists y.(source(y, x) \vee target(x, y))$ is satisfied if and only if x represents a vertex. For convenience, we introduce the following auxiliary formulae to make sure that a variable can only represent either a vertex or a hyperedge.

$$\begin{aligned}
vertex(x) &\triangleq \neg\exists y.(source(y,x) \vee target(x,y)) \text{ “}x\text{ is a vertex”}\\
edge(x) &\triangleq \exists y.(source(y,x) \vee target(x,y)) \text{ “}x\text{ is a hyperedge”}\\
vertexSet(X) &\triangleq \forall x.x \in X \rightarrow vertex(x) \text{ “}X\text{ is a set of vertices”}\\
edgeSet(X) &\triangleq \forall x.x \in X \rightarrow edge(x) \text{ “}X\text{ is a set of hyperedges”}
\end{aligned}$$

To keep notation simple, we introduce specialized quantifiers to require the existence of a variable representing either a vertex or an edge. Formally, we write $\exists^{\mathrm{v}}\, x.\varphi(x)$ to require that a vertex x exists satisfying $\varphi(x)$, i.e. $\exists x(vertex(x) \wedge \varphi(x))$ and $\forall^{\mathrm{v}}\, x.\varphi(x)$ to require that all vertices x satisfy $\varphi(x)$, i.e. $\forall x(vertex(x) \rightarrow \varphi(x))$. Analogously, we can define specialized quantifiers $\exists^{\mathrm{e}}\, x.\varphi(x)$, $\forall^{\mathrm{e}}\, x.\varphi(x)$ to require that x represents an edge. These quantifiers can be lifted to set quantifiers by using the formulae $vertexSet(X)$ and $edgeSet(X)$ instead of $vertex(x)$ and $edge(x)$.

Example 4.1.4 (Heap Property)**.** *We already discussed that not every hypergraph represents a valid heap in Section 3.3. However, the language of all hypergraphs corresponding to a valid heap is MSO_2F definable as follows.*

$$\begin{aligned}
heaps \triangleq &\bigwedge_{s \in S} \forall^{\mathrm{v}}\, x\, \forall^{\mathrm{e}}\, y\, \forall^{\mathrm{e}}\, z[\\
&(y \neq z \wedge source(x,y) \wedge source(x,z)) \rightarrow \neg(lab_s(y) \wedge lab_s(z))\\
&] \wedge \forall^{\mathrm{e}}\, x[\bigvee_{s \in S} lab_s(x) \wedge (\\
&\exists^{\mathrm{v}}\, s\, \exists^{\mathrm{v}}\, t(source(s,x) \wedge target(x,t)\\
&\wedge \forall^{\mathrm{v}}\, z(source(z,x) \rightarrow z = s \wedge target(x,z) \rightarrow z = t))\\
&)]
\end{aligned}$$

Intuitively, the formula heaps first asserts that no vertex has two outgoing selector edges with the same label. Then, it is required that every hyperedge is labeled with some selector label and connected to exactly one source vertex and one target vertex, i.e. every hyperedge is of rank two. Thus, every MSO_2F formula φ can be interpreted over valid heaps only by considering the conjunction $\varphi \wedge heaps$. ■

We now turn to the decision problems mentioned earlier in this section for MSO_2F. For a given finite hypergraph $\mathcal{H}$ and a given MSO_2F formula φ, it is decidable whether $\underline{\mathcal{H}} \models \varphi$ holds, because it suffices to apply the semantics of MSO_2F and there are only finitely many choices for every (set) variable.

What about the satisfiability problem? As shown in [EFE95] (see also [CE12]), the satisfiability problem for a set of arbitrary finite hypergraph models defined in MSO_2L is undecidable. However, if only graphs of bounded tree width are considered, the problem becomes decidable.

Proposition 4.1.2 (Courcelle [CE12])**.** *The satisfiability problem for hypergraph models of bounded tree width in MSO_2L is decidable.* ■

It is straightforward to see that linear orders as well as tree models can be defined in MSO_2L and therefore the complexity results of Proposition 5.3.1 also hold for MSO_2L. Thus, a reduction to MSO_2L does not yield any upper complexity bound for the satisfiability problem. Proposition 4.1.2 will be used in Chapter 5 to construct a fragment of hyperedge replacement grammars with a decidable language inclusion problem.

4.2. Separation Logic

For practical applications, monadic second-order logic is infeasible due to the high complexity of the satisfiability problem (see Proposition 5.3.1). In addition to that, MSO formulae tend to become complex, because even simple properties like the existence of a path between two vertices already requires multiple set quantifications. Separation logic is an extension of Hoare logic for program verification by Reynolds to deal with programs accessing and manipulating memory on the heap [Rey00]. In contrast to previously discussed approaches, the indirection via hypergraph languages is dropped which means separation logic is directly interpreted over heaps as underlying structure. Furthermore, Separation logic is designed to allow *local reasoning*, i.e. formulae should only talk about the locations which are actually accessed by a program [O'H12]. In order to sharpen the intuition on local reasoning, consider the following approach to define a (first-order or MSO) logic working directly on heap structures.

> For a heap $h : \texttt{Loc} \rightharpoonup \texttt{Val}$ defined over a finite set of selectors S, a *heap model* is a structure $\underline{h} = (dom(h) \cup img(h) \cup \{\texttt{null}\}, \texttt{null}, (\overset{s}{\mapsto})_{s \in S})$ where $\texttt{null}$ is a constant symbol referring to the null reference and $\overset{s}{\mapsto} \subseteq dom(h) \times img(h)$ is a binary relation, called a *pointer assertion*, such that $(x, y) \in \overset{s}{\mapsto}$ if and only if $h(x.s) = y$.

Unfortunately, this logic quickly leads to complicated proof rules even for simple programs, because every formula specifies a set of global program states including areas of memory which are not touched by the program in question. Since such untouched areas of memory can still invalidate pre- and postconditions of a specification, stronger specifications than one would intuitively think of are often needed to obtain proper proofs. The following example by O'Hearn [O'H12] highlights this problem.

Example 4.2.1. *Let $tree(p) \wedge reach(p,n)$ be a formula asserting that p is the root of a directed binary tree and n is a vertex in this tree. Additionally, let $\neg allocated(n)$ be a formula to ensure that the location described by n is not allocated, i.e. points to* `null`*. Now, assume we want to show that a program $DisposeTree(p)$ recursively frees all memory allocated by the tree with root p. We might write $\{tree(p) \wedge reach(p,n)\}DisposeTree(p)\{\neg allocated(n)\}$ to specify this, where the leftmost component is a precondition and the rightmost component is the corresponding postcondition. Unfortunately, this specification is insufficient to obtain a valid proof, because it does not explain what happens to vertices which are not in the currently considered subtree. For instance, if p points to the left subtree of the overall tree t, it does not rule out that $DisposeTree$ already frees some memory associated with the right subtree of t. Of course we can strengthen our conditions, e.g. add a conjunct $\forall m.\neg reach(p,m) \to allocated(m)$ to the precondition and a conjunct $\forall m.\neg reach(p,m) \to allocated(m)$ to the postcondition, respectively, in order to assert that vertices not belonging to the current subtree are not disposed. However, this blows up the size of formulae in pre- and postconditions.* ■

Separation logic simplifies this issue by introducing a new logical operator $*$ called the *separating conjunction*, which can be used to partition the heap into independent areas of memory. Then, given a formula $\varphi * \psi$, a theorem prover can reason about the properties φ and ψ in the respective part of the heap without taking other locations into account. Note that the concept of the separating conjunction is not an artificial construct, but is close to memory models of real-world programming languages. For example, the most recent specification of the C++ programming language contains the following requirement:

> "Two or more threads of execution can update and access separate memory locations without interfering with each other[...]" [ISO12].

We now introduce the syntax and semantics of separation logic with recursive definitions. Intuitively, simple separation logic consists of two types of formulae: classical first-order logic with operators $\exists x.\varphi, \varphi \vee \psi, \neg\varphi$ etc. and heap formulae emp, $x.s \mapsto y$, $x.s \mapsto$ `null`, and $\varphi * \psi$. The set of all simple separation logic formulae is denoted by `SSLF`. More formally, analogous to Section 4.1, an *interpretation* for (simple) separation logic formulae is a finite partial function $\jmath : Var \rightharpoonup$ `Val` assigning a value to each variable. Furthermore, an interpretation is called *safe* if every variable refers to a proper object on the heap, i.e. $img(\jmath) \subseteq \{1, 1+ \mid Ⓢ \mid, 1 + 2\cdot \mid Ⓢ \mid, ...\}$. In this thesis, only safe interpretations are considered. In compliance with terminology from programming languages, interpretations for separation logic formulae are also called the stores in the literature [O'H12], [Rey00].

Definition 4.2.1 (Semantics of Simple Separation Logic)**.** Let h : `Loc` $\rightharpoonup$ `Val` be a heap and $\jmath$ an interpretation. For separation logic formulae φ and ψ, the model relation

$\models^{\jmath} \subseteq \texttt{Heaps} \times \texttt{SSLF}$ is defined inductively as follows.

$$\begin{aligned}
h \models^{\jmath} emp &\text{ iff } dom(h) = \emptyset \\
h \models^{\jmath} x.s \mapsto y &\text{ iff } dom(h) = \{\jmath(x) + cn(s)\}, h(\jmath(x) + cn(s)) = \jmath(y) \\
h \models^{\jmath} x.s \mapsto \texttt{null} &\text{ iff } dom(h) = \{\jmath(x) + cn(s)\}, h(\jmath(x) + cn(s)) = \texttt{null} \\
h \models^{\jmath} \varphi * \psi &\text{ iff } \text{ exists } h_1, h_2 \in \texttt{Heaps} \text{ such that } h = h_1 \dot{\cup} h_2 \text{ and} \\
&\quad h_1 \models^{\jmath} \varphi \text{ and } h_2 \models^{\jmath} \psi
\end{aligned}$$

The semantics of first-order formulae $\exists x.\varphi$, $\varphi \vee \psi$, $\neg\varphi$, $x = y$, etc. is defined as in Definition 4.1.3. Again, we write $h \models \varphi$ if there exists a compatible interpretation $\jmath$ such that $h \models^{\jmath} \varphi$. ■

Note that the semantics of a pointer assertion $x.s \mapsto y$ is different from our previously defined naive version of heap models in order to allow local reasoning. Now, a pointer assertion captures a heap with *exactly one* pointer $x.s \mapsto y$, i.e. no larger heap h containing this pointer also satisfies $x.s \mapsto y$. We can compose larger heaps using the separating conjunction $*$ as illustrated in the following example.

Example 4.2.2. *Let $S = \{1, ..., n\}$ be a set of selectors. A heap assigning a value to each selector of the object represented by x is specified by the formula*

$$x \mapsto (y_1, ..., y_n) \triangleq x.1 \mapsto y_1 * x.2 \mapsto y_2 * ... * x.n \mapsto y_n.$$

■

Since the separating conjunction requires the existence of a proper partition of the heap, there are some subtle differences between $*$ and the "classical" conjunction $\wedge$. For instance, although $\exists x \exists y (x.1 \mapsto y \wedge \neg x.1 \mapsto y)$ is clearly unsatisfiable, this is not the case for $\exists x \exists y (x.1 \mapsto y * \neg(x.1 \mapsto y))$. Consider for example the heap $h = [23 \mapsto 42]$ where $\jmath(x) = 22$. In this case, for $x = 23$ and $y = 42$, there exists a partition $h = [23 \mapsto 42] \dot{\cup} [\,]$ such that $x.1 \mapsto y$ is true in the first component and $\neg(x.1 \mapsto y)$ is true for the empty heap. Conversely, $\exists x \exists y (x.1 \mapsto y * x.1 \mapsto y)$ is obviously unsatisfiable, but $\exists x \exists y (x.1 \mapsto y \wedge x.1 \mapsto y)$ is equivalent to $x.1 \mapsto y$. Reynolds showed in [Rey00] that the separating conjunction satisfies the same equivalence rules as the classical conjunction.

Proposition 4.2.1. *Let φ, ψ, ϑ be (simple) separation logic formulae. Then, the following equivalences hold.*

$$\begin{aligned}
\varphi * \psi \equiv& \psi * \varphi \\
(\varphi * \psi) * \vartheta \equiv& \varphi * (\psi * \vartheta) \\
(\varphi \vee \psi) * \vartheta \equiv& (\varphi * \vartheta) \vee (\psi * \vartheta) \\
(\exists x.\varphi) * \psi \equiv& \exists x(\varphi * \psi) \textit{ if } x \textit{ does not occur free in } \psi
\end{aligned}$$

■

A consequence of the change in the semantics of pointer assertions to capture exactly the specified pointer is that simple separation logic formulae are very restricted in their capability to describe sets of heaps, because only single memory configurations can be specified. Thus, simple separation logic is unsuited to specify dynamic data structures like linked lists and trees.

This is overcome by adding recursive predicates to simple separation logic such that inductive structures can be defined. Let $\mathtt{Pred} = \{\pi, \rho, ...\}$ be a set of *predicate names* such that each predicate name $\rho \in \mathtt{Pred}$ is associated with a number of parameters $x_1, ..., x_n$, i.e. Pred is ranked. Syntactically, a formula in separation logic with recursive definitions is a simple separation logic formula where additionally predicate calls $\rho(x_1, ..., x_n)$ with variables $x_1, ..., x_n$ may occur as atomic formulae. The set of all separation logic formulae with recursive definitions is denoted by $\mathtt{SLF}_{RD}$. In order to define a proper semantics for $\mathtt{SLF}_{RD}$ including predicate calls, we have to clarify the environment in which predicates are interpreted first. Note that there are several equivalent approaches to define the semantics of predicate calls for separation logic in the literature, for example by unfolding trees (see [IRS13]) and by fixpoint semantics (see [JGN14], [O'H12]). In this thesis, the latter approach is used.

Definition 4.2.2 (Predicate Definition)**.** Let $\rho \in \mathtt{Pred}$ be a predicate name. Then, a *predicate definition* of ρ is a formula

$$\rho(x_1, ..., x_n) \triangleq \bigvee_{j=1}^{m} \rho_j(x_1, ..., x_n)$$

where $\rho_j(x_1, ..., x_n)$ is an $\mathtt{SLF}_{RD}$ formula for each $j \in [1, m]$ and some $n, m \in \mathbb{N}$. We collect predicate definitions for all predicate names in an *environment* $\Gamma_{\mathtt{Pred}}$ and write $\Gamma_{\mathtt{Pred}}(\rho)$ to denote the predicate definition in $\Gamma_{\mathtt{Pred}}$ that belongs to $\rho \in \mathtt{Pred}$. The set of all environments is denoted by Env. ■

We now extend the model relation for simple separation logic formulae by a fixpoint semantics for predicate definitions.

Definition 4.2.3 (Semantics of Environments)**.** Let Γ be an $\mathtt{SLF}_{RD}$ environment over a set Pred of predicate names. The *predicate interpretation* η_Γ of Γ is the least fixpoint of the function $f_\Gamma : (\mathtt{Pred} \to 2^{\mathtt{Loc}^\star \times \mathtt{Heaps}}) \rightharpoonup (\mathtt{Pred} \to 2^{\mathtt{Loc}^\star \times \mathtt{Heaps}})$ with respect to set inclusion, where $f_\Gamma(\eta)(\rho) = \{(l, h) \mid h, \eta \models^{\{x_j \mapsto l(j) \mid j \in [1,n]\}} \rho_1 \vee ... \vee \rho_n\}$ for each predicate definition $\rho = \rho_1 \vee ... \vee \rho_n \in \Gamma$. ■

Together with the semantics of simple separation logic formulae presented in Definition 4.2.1, the rule

$$h, \eta_\Gamma \models^{\jmath} \rho(x_1, ..., x_n) \text{ iff } (\jmath(x_1) \cdot ... \cdot \jmath(x_n), h) \in \eta_\Gamma(\rho)$$

yields the semantics of separation logic formulae. The set of all languages realizable by separation logic formulae is collected in the class $\mathtt{SL}_{RD}$. Before we discuss fragments of separation logic in more detail, we look at an example to illustrate how data structures can be specified in $\mathtt{SL}_{RD}$.

Example 4.2.3. *Recall the HRG $\mathfrak{G}$ specifying binary trees with additional edges from vertices to their parents and connected leaves as presented in Example 3.2.2. We can also provide a predicate definition in $\texttt{SLF}_{RD}$ to define the corresponding set of heaps.*

$$\begin{aligned}\rho(x,p,leaf_l,leaf_r) \triangleq & [x \mapsto (\texttt{null},\texttt{null},p,leaf_r) \wedge x = leaf_l] \\ & \vee[\exists l \exists r \exists m (x \mapsto (l,r,p,\texttt{null}) * \rho(l,x,leaf_l,m) * \rho(r,x,m,leaf_r))]\end{aligned}$$

*Every vertex in the tree is represented by an object $x = (leftChild, rightChild, parent, nextLeaf)$ where the first two components point to the left and right subtree, respectively, the third component points to the parent of a vertex and the last component points to the next leaf (when read from left to right) or to **null** if x is not a leaf. The predicate definition $\rho(x,p,leaf_l,leaf_r)$ takes four parameters where x is the currently considered vertex in the tree, p its parent vertex and $leaf_l$ and $leaf_r$ are the leftmost and rightmost leaves in the subtree with x as root, respectively. Note that these parameters correspond to the external vertices of the HRG $\mathfrak{G}$ illustrated in Figure 3.4 which realizes the same set of heaps. Furthermore, every predicate call corresponds to one occurrence of a nonterminal hyperedge in $\mathfrak{G}$.* ■

4.3. The Relation between $\texttt{SLF}_{RD}$ and $\mathcal{HRL}$

A hyperedge replacement grammar can be viewed as a formalism which is composed of operations that add and relabel vertices and edges in each derivation step (cf. [SR97], [CE12]). Separation logic with recursive definitions is equipped with similar mechanisms. Intuitively, a “derivation step” corresponds to the unrolling of a predicate call and every pointer assertion $x.s \mapsto y$ requires the existence of a selector edge. In order to illustrate the relationship between $\texttt{SLF}_{RD}$ and $\mathcal{HRL}$, recall Example 3.2.1 which shows that every context-free string grammar can be translated into an HRG. Since every string over a finite alphabet can be represented by a heap, for instance by a singly-linked list where each list element stores a label in a second component, we show that a context-free string grammar $\mathfrak{G} = (N, T, \mathfrak{P}, \xi)$ in Greibach normal form can also be defined by an $\texttt{SLF}_{RD}$ formula. For each nonterminal $\alpha \in N$, we introduce a new predicate name α taking two parameters l (left) and r (right), i.e. the corresponding predicate definition is $\alpha(l,r)$. Then, if $\mathfrak{a} = \alpha \to \varepsilon \in \mathfrak{P}$ is a production rule, we define a formula $\alpha_{\mathfrak{a}}(l,r) \triangleq l \mapsto (\texttt{null}, \varepsilon) \wedge l = r$. Every production rule of the form $\mathfrak{a} = \alpha \to a\beta_1...\beta_k \in \mathfrak{P}$ is represented by a formula

$$\alpha_{\mathfrak{a}}(l,r) \triangleq \exists x \exists y_1 ... \exists y_{k-1}[l \to (x,a) * \beta_1(x,y_1) * \beta_2(y_1,y_2) * ... * \beta_k(y_{k-1},r)].$$

Then, it is straightforward to see for predicate definitions $\alpha(l,r) = \bigvee_{\mathfrak{a}\in\mathfrak{P}} \alpha_{\mathfrak{a}}(l,r)$ and an initial formula $\exists l \exists r.\xi(l,r) \wedge l \neq r$ that the language $L(\mathfrak{G})$ is specified. Furthermore, we directly obtain from the undecidability of the language inclusion problem for string languages that the entailment problem and also the satisfiability problem for $\texttt{SLF}_{RD}$ is undecidable.

The similarity of this construction to an analogous construction for HRGs (see Example 3.2.1) should be noted. We can observe that a predicate is introduced for every nonterminal hyperedge and a predicate definition contains as many disjunctions as there are production rules for the respective nonterminal. In addition to that, the parameters of a predicate correspond to the external vertices of every production rule. In this section, we will see that this is not a coincidence. We report on work of Jansen et al. (cf. [JGN14]) which provides an effective translation between a fragment of $\mathtt{SLF}_{RD}$ and heaps realizable by DSGs. Note that $\mathtt{SL}_{RD}$ and $\mathcal{HRL}$ are incomparable in general as shown in Lemma 6.1.1. Hence, only HR-languages realized by data structure grammars (see Definition 3.3.2) and a restricted class of $\mathtt{SLF}_{RD}$ formulae, denoted by $\mathtt{SLF}_{HR}$, are considered in this section.

Since all heaps satisfying an $\mathtt{SLF}_{HR}$ formula φ must specify sets of graphs with bounded tree width (see Section 3.2), we have to limit possible interpretations for variables x which never occur on the left hand side of a pointer assertion. Otherwise, a formula like $\sigma(x, y) \triangleq \exists z \exists z'.\sigma(z, z') \vee x = y$ has models with arbitrarily many variables which can point to each other in every possible way. Thus, there are graphs corresponding to some of these models forming a square grid of arbitrary size, i.e. the set of heaps satisfying $\sigma(x, y)$ does not have bounded tree width. This is prevented by the following assumption.

Definition 4.3.1 (Dangling Pointer Assumption)**.** For a given $\mathtt{SLF}_{RD}$ formula φ, we assume that for all interpretations $\jmath$ satisfying φ and all quantified variables x, y occurring in φ or in predicate definitions used inside of φ, it holds that $\jmath(x) \neq \jmath(y)$ unless explicitly stated otherwise. ■

Note that this assumption is global in the sense that variables introduced during different calls of the same predicate are assumed to be different if not stated otherwise. From now on, we take the dangling pointer assumption for granted for every $\mathtt{SLF}_{HR}$ formula. Furthermore, we restrict the syntax of $\mathtt{SLF}_{RD}$ as follows.

Definition 4.3.2 (The Fragment $\mathtt{SLF}_{HR}$)**.** The syntax of the fragment $\mathtt{SLF}_{HR}$ is the set of $\mathtt{SLF}_{RD}$ formulae which can be obtained by the following grammar with initial nonterminal symbol T.

$$\begin{aligned}
E &::= x \in Var \mid \mathtt{null} && \text{“expressions”}\\
P &::= x = y \mid x \neq y \mid P \wedge P && \text{“pure formulae”}\\
S &::= emp \mid x.s \mapsto E && \text{“spatial formulae”}\\
H &::= S \mid H * H \mid \exists x : H \mid \rho(E_1, ..., E_k) && \text{“heap formulae”}\\
R &::= H \mid P \wedge R \mid \exists x : R && \text{“predicate rules”}\\
T &::= R \mid T \vee T && \text{“top-level formulae”}
\end{aligned}$$

Here, $\rho(E_1, ..., E_k)$ is assumed to be a predicate name from a fixed set of such names `Pred`. The semantics of $\mathtt{SLF}_{HR}$ formulae is the same as the semantics of $\mathtt{SLF}_{RD}$ formulae which has been specified in Definition 4.2.1 and Definition 4.2.3. ■

Intuitively, the fragment $\mathtt{SLF}_{HR}$ restricts the use of the classical conjunction and negation to *pure formulae*, i.e. formulae of the form $\bigwedge x = y$, and allows negation only to compare variables. This is a consequence of the fact that HR-languages are in general not closed under intersection and complement. Also note that the formula *true* is not allowed, because no DSG is able to generate all heaps. Furthermore, $\texttt{SLF}_{HR}$ formulae as well as data structure grammars can only specify the composition of dynamic data structures, but not actual data, because expressions cannot represent arbitrary concrete values. This can at least partially be lifted by adding a fixed set of constant values to the production rules of pure formulae or expressions in the grammar of Definition 4.3.2. In the context of HR-languages, this corresponds to using labeled vertices to represent values. One of the main theorems of [JGN14] is the following.

Proposition 4.3.1 (Equivalence of $\texttt{SL}_{HR}$ and $\mathcal{DSL}$)**.** *Let A be an alphabet. Then, there exists a translation function $hrg[\![.]\!] : \textbf{\texttt{Env}} \rightharpoonup DSG_A$ such that for every $\texttt{SL}_{HR}$ environment Γ with top-level formula φ it holds that that*

$$h, \eta_\Gamma \models \varphi \;\; \textit{iff} \;\; graph(h) \in L(hrg[\![\Gamma]\!], \varphi).$$

Here, $graph(h)$ is the encoding of a heap h into a hypergraph as described at the beginning of Chapter 3. Conversely, there exists a translation function $env[\![.]\!] : \mathcal{DSG}_A \rightharpoonup \texttt{SLF}_{HR}$ such that for every DSG $\mathfrak{G}$ over an alphabet A with initial nonterminal φ it holds that

$$\mathcal{H} \in L(\mathfrak{G}, \varphi) \;\; \textit{iff} \;\; heap(\mathcal{H}), \eta_{env[\![\mathfrak{G}]\!]} \models env[\![\mathfrak{G}]\!](\varphi)$$

where $heap(\mathcal{H})$ is the interpretation of a hypergraph $\mathcal{H}$ as a heap. ■

A direct benefit of an effective translation between (a fragment of) separation logic and data structure grammars is that HRGs have been studied for almost thirty years and some interesting results can also be applied to separation logic with recursive definitions. For instance, Habel et al. showed that it can be checked in linear time with respect to the size of the derivation tree of an HRG $\mathfrak{G}$, whether all graphs generated by $\mathfrak{G}$ satisfy a given *compatible* graph property [Hab92]. Among these compatible graph properties are connectedness, k-boundedness and edge-colorability. Since the translation of $\texttt{SLF}_{HR}$ formulae to DSGs is also possible in linear time, Proposition 4.3.1 implies that it can also be verified whether all heaps satisfying a given $\texttt{SLF}_{HR}$ formulae have such graph properties in linear time. We present the essential steps of the method to construct a DSG from an $\texttt{SLF}_{HR}$ environment and vice versa, because these constructions are further needed in Chapter 6. For more details, including the required correctness proofs, we refer to [JGN14].

From $\texttt{SLF}_{HR}$ to $\mathcal{DSG}$ In order to identify variables of a given $\texttt{SLF}_{RD}$ formula φ with vertices in a hypergraph, we introduce a respective vertex labeling. A hypergraph $\mathcal{H}$ is called a *tagged heap configuration* if it is a heap configuration according to Definition 3.3.1 and additionally every vertex is labeled with a set of variables, i.e. $lab_\mathcal{H}(v) \in 2^{Var \cup \neg Var \cup \{\texttt{null}, \neg\texttt{null}\}}$ for each each $v \in V_\mathcal{H}$. Here, $\neg Var = \{\neg x \mid x \in Var\}$ introduces negated variables to denote that a vertex does explicitly not correspond to a variable. We write $tag_\mathcal{H}$ to denote the restriction of the domain of $lab_\mathcal{H}$ to vertices of $\mathcal{H}$ and denote the set of all tagged heap configurations over an alphabet A by THC_A. Furthermore, $tag^+_{\mathcal{H}}$ denotes the restriction of $tag_{\mathcal{H}}$ to variables from Var and `null` only. Since vertices with non-disjoint labeling are associated with common variables, they should point to the same location and therefore be merged. This is performed by the following operator.

Definition 4.3.3 (Unification Operator)**.** Let $\mathcal{H} = (V_{\mathcal{H}}, E_{\mathcal{H}}, src_{\mathcal{H}}, tgt_{\mathcal{H}}, lab_{\mathcal{H}}, \varepsilon)$ be a tagged heap configuration without external vertices. Furthermore, let $f : V_{\mathcal{H}} \rightharpoonup V$ be a function such that for all $u, v \in V_{\mathcal{H}}$, we have that $f(u) = f(v)$ if and only if $tag^+_{\mathcal{H}}(u) \cap tag^+_{\mathcal{H}}(v) \neq \emptyset$ and for all $x \in Var$ the set $\{x, \neg x\}$ is not included in $tag_{\mathcal{H}}(u) \cap tag_{\mathcal{H}}(v)$. If the second condition is violated, the tagging corresponds to a formula which is not satisfiable and the unification operator can directly map to the empty hypergraph $\mathcal{H}_\emptyset = (\emptyset, \emptyset, src_\emptyset, tgt_\emptyset, lab_\emptyset, \varepsilon)$. Otherwise, the *unification* $\Downarrow \mathcal{H}$ is defined as the tagged heap configuration $\Downarrow \mathcal{H} = (V, E_{\mathcal{H}}, src, tgt, lab, \varepsilon)$ where $V = f(V_{\mathcal{H}})$, $att(e)[i] = f(att(e)[i])$, $lab(e) = lab_{\mathcal{H}}(e)$ and

$$lab(v) = \bigcup_{u \in \{w \in V_{\mathcal{H}} \mid f(w) = f(v)\}} lab_{\mathcal{H}}(u)$$

for every hyperedge $e \in E_{\mathcal{H}}$, vertex $v \in V$ and index $i \leq |att(e)|$, respectively. ■

Furthermore, the disjoint union of two tagged heap configurations $\mathcal{H}_1, \mathcal{H}_2 \in THC_A$ without external vertices is defined as

$$\begin{aligned}\mathcal{H}_1 \bowtie \mathcal{H}_2 = \Downarrow (&V_{\mathcal{H}_1} \,\dot\cup\, V_{\mathcal{H}_2}, E_{\mathcal{H}_1} \,\dot\cup\, E_{\mathcal{H}_2}, src_{\mathcal{H}_1} \,\dot\cup\, src_{\mathcal{H}_2}, \\ &tgt_{\mathcal{H}_1} \,\dot\cup\, tgt_{\mathcal{H}_2}, lab_{\mathcal{H}_1} \,\dot\cup\, lab_{\mathcal{H}_2}, t_{\mathcal{H}_1} \,\dot\cup\, t_{\mathcal{H}_2}).\end{aligned}$$

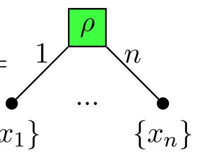
We proceed as follows. At first, an $\texttt{SLF}_{HR}$ formula is translated into a set of tagged heap configurations including nonterminal hyperedges. Then, a function *expose* is introduced to define the external vertices of each heap configuration. Finally, the production rules of an appropriate data structure are derived from this set of hypergraphs.

Definition 4.3.4 ($\texttt{SLF}_{HR}$ to THC_A)**.** The mapping $tgraph[\![.]\!] : \texttt{SLF}_{HR} \rightharpoonup 2^{THC_A}$ translating a given $\texttt{SLF}_{HR}$ formula into a set of tagged heap configurations is defined by structural induction. The base cases for atomic formulae are:

- The empty heap is mapped to the empty hypergraph, i.e. $tgraph[\![emp]\!] = \mathcal{H}_\emptyset$.
- Pointer assertions $x.s \mapsto y$ are mapped to an s-labeled terminal edge between vertices tagged with x and y, i.e. $tgraph[\![x.s \mapsto y]\!] = \{\Downarrow \mathcal{H}_s\}$ where

$$\mathcal{H}_s = \begin{tikzcd} \overset{\{x\}}{\bullet} \arrow[r, "s"] & \overset{\{y\}}{\bullet} \end{tikzcd}.$$

- Predicate calls are mapped to nonterminal hyperedges with vertices corresponding to the parameter variables as attached vertices, i.e. $tgraph[\![\rho(x_1, ..., x_n)]\!] = \{\Downarrow \mathcal{H}_\rho\}$ where

$\mathcal{H}_\rho =$


.

Assuming there already exist translations for formulae φ and ψ, the composite cases are treated as follows:

- $\mathit{tgraph}[\![\varphi \vee \psi]\!] = \mathit{tgraph}[\![\varphi]\!] \cup \mathit{tgraph}[\![\psi]\!]$.
- $\mathit{tgraph}[\![\varphi * \psi]\!] = \mathit{tgraph}[\![\varphi]\!] \bowtie \mathit{tgraph}[\![\psi]\!]$.
- The tagging of existentially quantified variables is removed, i.e.

  $$\mathit{tgraph}[\![\exists x.\varphi]\!] = \{\mathcal{H} \mid \mathcal{H}' \in \mathit{tgraph}[\![\varphi]\!], \mathit{tag}_{\mathcal{H}} = [v \mapsto \mathit{tag}_{\mathcal{H}'}(v) \setminus \{x\} \mid v \in V_{\mathcal{H}'}]\}.$$

- The tagging is modified according to pure formulae $x = y$, i.e.

  $$\mathit{tgraph}[\![x = y \wedge \varphi]\!] = \{\Downarrow \mathcal{H} \mid \mathcal{H}' \in \mathit{tgraph}[\![\varphi]\!]\}$$

  where $\mathcal{H}$ equals $\mathcal{H}'$ except for the tagging which is defined for all $v \in V_{\mathcal{H}}$ by

  $$\mathit{tag}_{\mathcal{H}}(v) = \begin{cases} \mathit{tag}_{\mathcal{H}'}(v) \cup \{x, y\} & \text{if } \{x, y\} \cap \mathit{tag}_{\mathcal{H}'}(v) \neq \emptyset \\ \mathit{tag}_{\mathcal{H}'}(v) & \text{otherwise.} \end{cases}$$

- The tagging for pure formulae $x \neq y$ is modified analogously by

  $$\mathit{tag}_{\mathcal{H}}(v) = \begin{cases} \mathit{tag}_{\mathcal{H}'}(v) \cup \{\neg y\} & \text{if } x \in \mathit{tag}_{\mathcal{H}'}(v) \\ \mathit{tag}_{\mathcal{H}'}(v) \cup \{\neg x\} & \text{if } y \in \mathit{tag}_{\mathcal{H}'}(v) \\ \mathit{tag}_{\mathcal{H}'}(v) & \text{otherwise.} \end{cases}$$

■

It remains to define the external vertices of each tagged heap configuration in $\mathit{tgraph}[\![\varphi]\!]$ and to define an environment based on these heap configurations. Intuitively, external vertices correspond to the parameters $\overline{x} = x_1, ..., x_n$ of a predicate name ρ which is formalized by the function $\mathit{expose}(\mathcal{H}, \overline{x}) = (V_{\mathcal{H}}, E_{\mathcal{H}}, \mathit{src}_{\mathcal{H}}, \mathit{tgt}_{\mathcal{H}}, \mathit{lab}_{\mathcal{H}}, v_1...v_n)$ for each tagged heap configuration $\mathcal{H}$ where $\mathit{tag}_{\mathcal{H}}(v_i) = \{x_i\}$ for $i \in [1, n]$. The HRG corresponding to an $\texttt{SLF}_{HR}$ formula φ with environment Γ is then obtained by introducing production rules $\rho \to \mathit{expose}(\mathcal{H}, \overline{x})$ where $\mathcal{H}$ is a tagged heap configuration associated with the predicate call $\rho(\overline{x})$.

Definition 4.3.5 ($\texttt{SLF}_{HR}$ Environment to $\mathcal{DSG}$)**.** For an environment Γ, the translation function $\mathit{hrg}[\![.]\!] : \texttt{Env} \rightharpoonup \mathit{DSG}_A$ is defined as $\mathit{hrg}[\![\Gamma]\!] = (N, T, \mathfrak{P})$, where $N = \texttt{Pred}$ is the set of predicate names, $T = S$ is the set of selectors and $\mathfrak{P}$ consists of all production rules of the form $\rho \to \mathit{expose}(\mathcal{H}, \overline{x})$ with $\mathcal{H} \in \mathit{tgraph}[\![\rho(\overline{x})]\!]$ for each formula $\rho(\overline{x}) \in \Gamma$.
■

From $\mathcal{DSG}$ to $\texttt{SLF}_{HR}$ For the converse direction, again a tagging function is used to identify variables and vertices.

Definition 4.3.6 (Default Tagging)**.** The *default tagging* of a heap configuration $\mathcal{H}$ is defined as $tag : V_{\mathcal{H}} \rightharpoonup 2^{Var \cup \{\texttt{null}\}}$ with

$$tag(v) = \begin{cases} \{\texttt{null}\} & \text{if } v = v_{\texttt{null}} \\ \{x_j\} & \text{if } v = \overline{ext_{\mathcal{H}}}[j] \\ \{r\} & \text{otherwise, where } r \text{ is a variable which has} \\ & \text{not already occurred in the construction.} \end{cases}$$

We denote a hypergraph $\mathcal{H}$ together with its default tagging by $\mathcal{H}^t$. Note that we implicitly assumed the existence of a unique vertex $v_{\texttt{null}}$. ■

The translation of DSGs to $\texttt{SLF}_{HR}$ formulae works in two steps. At first, we present an inductive procedure $form[\![.]\!] : THC_A \rightharpoonup \texttt{SLF}_{HR}$ to obtain $\texttt{SLF}_{HR}$ formulae specifying a single heap configuration. In this procedure, every terminal edge is replaced by a pointer assertion, every nonterminal edge by a predicate call and for each vertex tagged $\{x\}$, a variable x is introduced. This subformulae are composed using the separating conjunction. Then, for each nonterminal $\xi \in N$, a predicate definition is obtained by the disjunction of all formulae corresponding to production rules with ξ as left hand side. The union of these predicate definitions yields the desired environment. More formally, for a DSG $\mathfrak{G} = (N, T, \mathfrak{P})$ the corresponding $\texttt{SLF}_{HR}$ environment is given by

$$env[\![\mathfrak{G}]\!] = \bigcup_{\xi \in N} \{\rho_\xi(x_1, ..., x_{rk(\xi)}) = \bigvee_{\mathfrak{r}=\xi \to \mathcal{X} \in \mathfrak{P}} form[\![\mathcal{X}^t]\!]\}.$$

It remains to formally define the function $form[\![.]\!]$.

Definition 4.3.7. The translation function $form[\![.]\!] : THC_A \rightharpoonup \texttt{SLF}_{HR}$ for a hypergraph $\mathcal{H}^t$ with its default tagging t is defined inductively as follows.

- For every terminal hyperedge $e = u \xrightarrow{s} v$ with $t(u) = x$ and $t(v) = y$, we have $form[\![e]\!] = x.s \mapsto y$.
- For every nonterminal hyperedge e with $lab(e) = \xi$, $att(e) = \overline{v}$ and $\overline{y} = t(\overline{v})$, we have $form[\![e]\!] = \rho_\xi(\overline{y})$.
- For every hypergraph $\mathcal{H}^t$, let $\{r_1, ..., r_m\}$ be the set of variables tagging non-external vertices in $\mathcal{H}^t$. Then, $form[\![\mathcal{H}^t]\!] = \exists r_1 ... \exists r_m.emp$ if $E_{\mathcal{H}^t} = \emptyset$ and

  $$form[\![\mathcal{H}^t]\!] = \exists r_1 ... \exists r_m.form[\![e_1]\!] * ... * form[\![e_n]\!]$$

  for $E_{\mathcal{H}^t} = \{e_1, ..., e_n\}$, respectively.

■

From the two constructions presented in this section and Proposition 4.3.1 it directly follows that for each $\texttt{SLF}_{HR}$ formula φ, an equivalent formula ψ exists such that

1. ψ contains no pure formulae, and
2. in every predicate definition $\bigvee_i \rho_i(\overline{x})$ belonging to ψ, every disjunct $\rho_i(\overline{x})$ is of the form
$$\rho_i(\overline{x}) \triangleq \exists r_1 ... \exists r_{m_i} . \vartheta_1(\overline{y_1}) * ... * \vartheta_{n_i}(\overline{y_{n_i}})$$
where $\vartheta_j(\overline{y_j})$ is either a pointer assertion or a predicate call and $[\overline{y_j}] \subseteq \{r_1, ..., r_{m_i}\} \cup [\overline{x}]$.

In addition to that, we obtain the undecidability of the satisfiability problem and the entailment problem for $\mathtt{SLF}_{HR}$ formulae as an immediate consequence of Proposition 4.3.1 and Example 3.2.1. As an example of this construction, consider the translation of context-free string grammars to $\mathtt{SLF}_{HR}$ formulae at the beginning of this section.

CHAPTER 5

Decidability of Entailment and Inclusion

Previously, we have seen formalisms - hyperedge replacement grammars, monadic second-order logic, separation logic - to specify sets of heaps and hypergraphs. Each of these formalisms has an undecidable language inclusion and entailment problem if no further constraints are added. For instance, Example 3.2.1 and Proposition 4.3.1 show that hyperedge replacement grammars and separation logic formulae are expressive enough to specify any context-free string language which is already sufficient to show undecidability of the language inclusion problem. However, a theorem by Courcelle yields a positive result for monadic second-order logic over hypergraph models of bounded tree-width. More precisely, recall Proposition 4.1.2, which states that

> for an MSO_2 formula φ, it is decidable whether there exists a hypergraph model of bounded tree width satisfying φ.

As a result of this, it is decidable whether the formula $\varphi \rightarrow \psi$ is satisfiable for two MSO_2F sentences φ and ψ. Thus, any formalism realizing only MSO_2F definable sets of hypergraphs (or heaps) of bounded tree width has a decidable language inclusion problem. In this chapter, we develop a syntactic fragments of HRGs with a decidable language inclusion problem. The underlying intuition is discussed first based on results by Engelfriet and Courcelle. Then, we introduce the class of tree-like HRGs and show that every tree-like HRG can be transformed into an equivalent MSO_2F sentence. By Proposition 4.1.2 and the fact that HRGs generate only graphs of bounded tree-width, this means that the language inclusion problem is decidable for tree-like HRGs. Due to the close relationship between data structure grammars and separation logic presented in Section 4.3, this result can be applied to show decidability of the entailment problem for several fragments of $\texttt{SLF}_{HR}$. These fragments are studied in Chapter 6.

5.1. HRGs with a Decidable Inclusion Problem

For languages of finite strings and finite trees, the question of MSO definability is answered by Büchi's theorem: Exactly the regular languages (of strings or trees) can be characterized by MSO sentences over appropriate structures (see Example 4.1.1 and Example 4.1.2).

The proof of Büchi's theorem constructs an MSO sentence for each regular language by encoding the accepting runs of a finite automaton in MSO. Analogously, a relationship between context-free string languages and MSO can be established with the help of right-linear string grammars, i.e. context-free string grammars where all production rules are of the form $\alpha \to a\beta$ or $\alpha \to \varepsilon$ for $\alpha, \beta \in N$ and $a \in T$. It can be shown that right-linear string grammars realize exactly the regular languages, i.e. the class of regular string language is a strict subset of the class of context-free string languages. Applying Büchi's theorem yields that exactly the languages realized by right-linear string grammars can be characterized in MSO.

Unfortunately, there is no generally accepted notion of finite automata for hypergraph languages which could serve as a link between logical formalisms and HRGs. However, Engelfriet and Courcelle extensively studied graph languages in an alternative framework using so-called *hyperedge replacement algebras* (HRA) [Cou90], [Cou91], [SR97], [CE12]. We report only informally on their most important results in order to gain a better understanding of the relationship between MSO_2L and $\mathcal{HRL}$.

For a finite alphabet A, the set of all finite strings over A, denoted by $A^\star$, can be generated by an algebra $\mathbb{A} = (A^\star, \cdot, \varepsilon, (a)_{a \in A})$ consisting of all finite words over A with concatenation of symbols $\cdot$, the empty word ε as neutral element and every symbol $a \in A$ as a constant. This structure is known as the *syntactic monoid*. Its equivalent to describe finite hypergraphs are hyperedge replacement algebras (HRA). Analogously to syntactic monoids, an HRA has the empty hypergraph $\mathcal{H}_\emptyset$ (see Section 4.3) as neutral element and is equipped with a set of constants. However, these constants are not symbols, but either a single labeled external vertex or two labeled external vertices connected by an edge. Since composition of hypergraphs is more complicated than concatenation of strings, HRAs are additionally equipped with three operations:

1. $\mathcal{H}_1 //_I \mathcal{H}_2$ denotes the *parallel composition* of two hypergraphs, i.e. the union of $\mathcal{H}_1$ and $\mathcal{H}_2$ in which common external vertices in the set of indices I are merged. All external vertices of both hypergraphs remain external (although their position in the sequence of external vertices may change).

2. $forget(\mathcal{H})$ removes the last external vertex of $\mathcal{H}$ from the sequence of external vertices.

3. $rename_f(\mathcal{H})$ changes the order of external vertices according to a permutation f.

Figure 5.1 illustrates how these operations can be applied to construct larger graphs. Analogously to $A^\star$ denoting the set of all words over an alphabet, we write $HRA^\star$ to denote the class of all hypergraphs that can be generated by an HRA. The relationship between languages in $HRA^\star$ and languages realizable by HRGs is established by the following theorem.

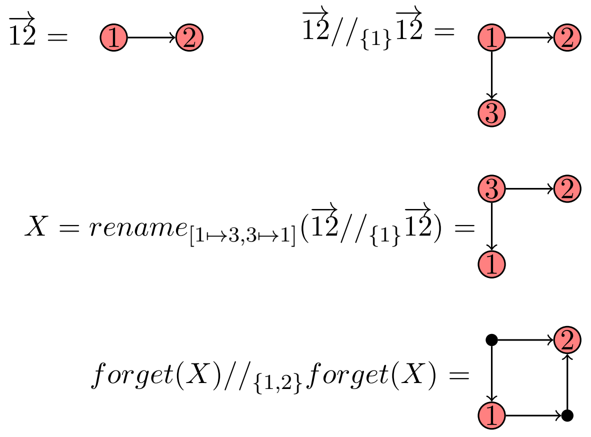

$\overrightarrow{12} =$
$\overrightarrow{12} //_{\{1\}} \overrightarrow{12} =$
$X = rename_{[1 \mapsto 3, 3 \mapsto 1]}(\overrightarrow{12} //_{\{1\}} \overrightarrow{12}) =$
$forget(X) //_{\{1,2\}} forget(X) =$
1
2
3

Figure 5.1.: Illustration of HRA operations

Proposition 5.1.1 (Equationality Theorem [SR97])**.** *A hypergraph language $L \in HRA^\star$ is realizable by an HRG if and only if there exists a regular tree language T and a partial mapping $h : Trees \rightharpoonup HRA^\star$ such that $L = h(T)$.* ■

Intuitively, the equationality theorem states that every HR-languages is the result of manipulating all trees in a regular tree language T by adding, removing and relabeling vertices and hyperedges, respectively. For example, T can be chosen to be the set of derivation trees of an HRG realizing L (cf. [SR97]).

A similar characterization exists for context-free string grammars: Every context-free string language $L \subseteq A^\star$ can be characterized by a regular tree language T corresponding to the set of derivation trees of an appropriate context-free string grammar and a partial function $h : Trees \rightharpoonup A^\star$ such that h maps every derivation tree t to the string obtained from reading the leaves of t from left to right (cf. [SR97]). Furthermore, we can observe that L is a regular string language if the inverse image h^{-1} is an MSO definable mapping. For example, this is the case for right-linear string grammars. The converse direction does not hold in general (cf. [Cou91]).

Thus, the languages realizable by HRGs form the context-free fragment of $HRA^\star$. What about regular languages? From a high-level perspective, a finite automaton takes an input, e.g. a string or a tree, and computes a "run", i.e. a sequence of states from a finite state set. If the last state of a run belongs to a dedicated set of final states, the input is considered to belong to the language accepted by the automaton. This abstract view on automata is used by Courcelle [Cou90] [CE12] to define the class of *recognizable* languages. More precisely, a language L over an F-algebra $\mathbb{M}$ - a set M equipped with a fixed set F of operators on this set - of objects is called recognizable, if there exists a finite F-algebra $\mathbb{Q}$, a homomorphism $val : \mathbb{M} \rightharpoonup \mathbb{Q}$ and a set $R \subseteq Q$ such that $L = val^{-1}(R)$. Here, $\mathbb{Q}$ defines the structure of the state set, val determines how a run is computed and R corresponds to the set of final states. Thus, recognizability means that a language L is given exactly by the "accepting runs" of val ending in a final state defined by the set

R. Similar notions like *finiteness* [SR97] and *compatibility* [Hab92] have been studied in the literature and compared by Lautemann [Lau91].

Example 5.1.1 (Recognizability and Regular String Languages)**.** *For a regular string language $L \subseteq A^\star$ over a finite alphabet A, the algebra $\mathbb{M}$ can be chosen to be the syntactic monoid $\mathbb{A} = (A^\star, \cdot, \varepsilon, (a)_{a\in A})$ as defined previously. Let $\mathfrak{B} = (S, A, q_0, \delta, R)$ be a deterministic finite automaton accepting L. This automaton can be represented by an algebra $\mathbb{Q} = (Q, \cdot, q_0, (q_a)_{a\in A})$ with $Q = S \dot\cup \{q_a \mid a \in A\}$ and $q \cdot q_a = \delta(q, a)$. Then, for $val : \mathbb{A} \rightharpoonup \mathbb{Q}$ with $a_1 ... a_n \mapsto q_{a_1} \cdot ... \cdot q_{a_n}$, it follows that $L = val^{-1}(R)$. Thus, as formally shown in [CE12], recognizability and regularity are equivalent for string languages.* ■

A hypergraph language L is called *HR-recognizable* if $L \in HRA^\star$ is recognizable. Several results indicate that HR-recognizability is well-suited to lift the concept of regular languages to hypergraphs, just like HRGs lift context-free languages to hypergraphs. For example, it can be shown that HR-languages are closed under intersection with HR-recognizable languages (cf. [Cou91], [CE12]) and that HR-recognizable languages are closed under Boolean operations (cf. [SR97]). Engelfriet notes in [Eng91] that the class of MSO_2F definable hypergraph languages is the smallest class with the same closure properties as regular string and tree languages that contains elementary graph languages like all simple graphs. Furthermore, one direction of Büchi's theorem (see Example 4.1.1) also works for HR-recognizable hypergraph languages.

Proposition 5.1.2 (Recognizability Theorem [CE12])**.** *Every MSO_2F definable hypergraph language is HR-recognizable.* ■

The converse direction, however, does not hold. Engelfriet and Courcelle showed that the mapping h used in the equationality theorem corresponds to val^{-1} if an HR-language L is HR-recognizable [SR97]. In addition to that it was shown that h is an MSO_2L definable mapping, i.e. there exists a finite set of MSO_2F formulae with at most two free variables specifying exactly the partial function h.

In contrast to the string case, the class of HR-recognizable languages is incomparable to the class of HR-languages. To see this, note that every HR-language is of bounded tree width, but the language of all connected graphs is MSO_2F definable. Figure 5.2 illustrates the relationship between HR-languages and HR-recognizable languages. Moreover, no proper characterization of HR-recognizable languages in terms of automata is known and Engelfriet and Courcelle state that

> "there are uncountably many [...] HR-recognizable sets of graphs, and this fact prevents any exact characterization of these sets in terms of graph automata or logical formulas. Such a characterization would generalize nicely the classical characterization of the recognizable (i.e., the regular) languages in terms of finite automata and monadic second-order sentences, but it cannot exist" [CE12].

It is an open problem whether such an exact characterization exists for recognizable graph languages of bounded tree-width. Results by Lapoire [Lap98] indicate that a

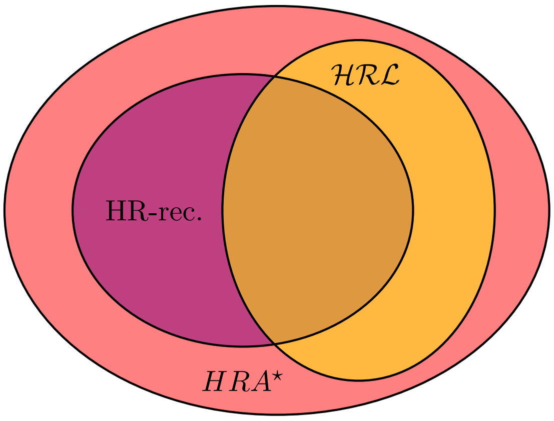


Figure 5.2.: Relationships between $HRA^\star$, $\mathcal{HRL}$ and HR-recognizability

hypergraph language of bounded tree-width is recognizable if and only if it is definable in an extended version of MSO_2F called counting monadic second-order logic. If this conjecture holds, the class of HR-recognizable languages (of bounded tree width) would be a proper fragment of the class of context-free graph languages as it is the case for string languages.

The equationality and recognizability theorem provide an intuition to find syntactic fragments of HRGs which realize only MSO_2L definable hypergraph languages: We have to ensure that all graphs $\mathcal{H} \in HG_T$ generated by an HRG $\mathfrak{G} = (N, T, \mathfrak{P})$ contain "enough structure" such that a derivation tree T of $\mathcal{H}$ can be reconstructed from $\mathcal{H}$ in MSO_2F. Then, we can verify whether $\mathcal{H}$ is really the result of a derivation corresponding to T, i.e. whether $\mathcal{H} = h(T)$ holds. More formally, we have to define an MSO_2F formula $\varphi_{\mathfrak{a}}(x)$ for each production rule $\mathfrak{a} \in \mathfrak{P}$ such that $\varphi_{\mathfrak{a}}(x)$ is satisfied if and only x is a vertex that belongs to a derivation tree in which x is labeled with $\mathfrak{a}$. Furthermore, we need formulae $\psi_d(x, y)$ for each $1 \leq d \leq max\{rk(\alpha) \mid \alpha \in N\}$ such that $\psi_d(x, y)$ is satisfied if and only if x and y occur in the derivation tree and y is the d-th child of x. Together these formulae define a derivation tree in a given hypergraph.

As an example, consider our recurring HRG shown in Figure 3.4. Every hypergraph $\mathcal{H}$ generated by this HRG is a binary tree in which every vertex is also connected to its parent vertex and leaves are additionally connected from left to right by a singly-linked list. Then, every inner vertex of $\mathcal{H}$ can be uniquely identified with a production rule of the derivation tree corresponding to the derivation of $\mathcal{H}$: We identify an inner vertex of $\mathcal{H}$ with the production rule $\mathfrak{a}_2$ if all of its children are leaves and with $\mathfrak{a}_1$ otherwise. The restriction of $\mathcal{H}$ to these vertices and edges yields the corresponding derivation tree. This is illustrated in Figure 5.3 where the left hypergraph is the result of the derivation shown in Figure 3.5 and the right hypergraph is the corresponding derivation tree.

If our syntactic fragment of HRGs should cover as many languages as possible, we cannot expect that every hypergraph contains its derivation tree as a subgraph and that

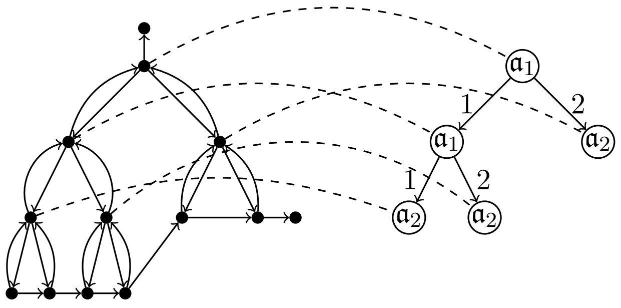


Figure 5.3.: Extracting a derivation tree from a hypergraph generated by the HRG shown in Example 3.2.2

every vertex can be uniquely identified with a production rule as in the previous example Consider, for example, the HRG in Figure 5.4 that models a doubly-linked list in which new elements can be added on both ends of the list. The single nonterminal symbol α

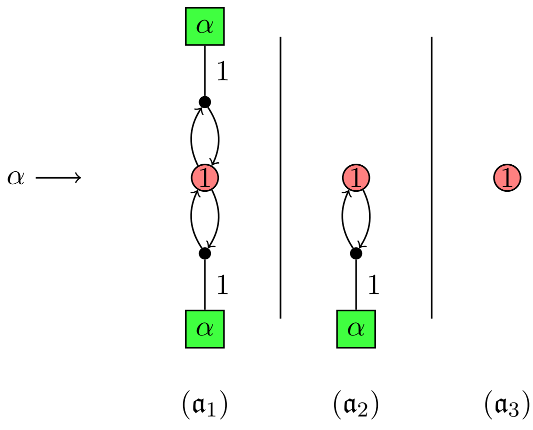


Figure 5.4.: HRG generating doubly-linked lists

serves a placeholder representing the remaining list. Although every vertex generated by this HRG can be identified with a root vertex of a production rule, the derivation tree of a given doubly-linked list is not unique for lists with more than two elements. For instance, different derivation trees can be obtained starting with $\alpha^\bullet$ for a doubly-linked list with four elements as illustrated in Figure 5.5.

5.2. Tree-Like Hyperedge Replacement Grammars

In compliance with the intuition presented in the previous section, we now develop a syntactic condition such that all HRGs satisfying this condition yield MSO_2F definable

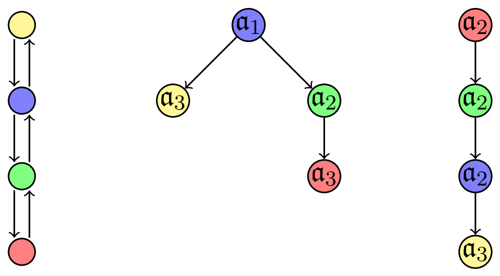


Figure 5.5.: A doubly-linked list and two possible derivation trees

HR-languages. This condition is called *tree-likeness*, because it ensures that a suited tree t can be reconstructed from each hypergraph $\mathcal{H}$ in an HR-language such that t is sufficient to describe $\mathcal{H}$ again. In order to define this condition properly, we need some notation first. Without loss of generality, we assume that the sets of vertices $V_\mathcal{H}$ and rhyperedges $E_\mathcal{H}$ of a hypergraph $\mathcal{H}$ are in an arbitrary fixed order to simplify constructions. For a hypergraph $\mathcal{H}$, let $free(\mathcal{H})$ be a function mapping $\mathcal{H}$ to the set of vertices of $\mathcal{H}$ attached to nonterminal hyperedges only. We write $\overline{free(\mathcal{H})}$ to denote that the set $free(\mathcal{H})$ is ordered according to the fixed order of $V_\mathcal{H}$.

Furthermore, we denote a *path* in $\mathcal{H}$ by an alternating sequence $v_0 \overset{e_1}{—} v_1 \overset{e_2}{—} ... \overset{e_n}{—} v_n$ of vertices and hyperedges such that $v_{i-1}, v_i \in att_\mathcal{H}(e_i)$ for all $1 \leq i \leq n$. Note that we do not require that a path is directed according to the functions *src* and *tgt*.

Definition 5.2.1 (Tree-like Hypergraph)**.** A hypergraph $\mathcal{H} = (V, E, src, tgt, lab, \overline{ext})$ over an alphabet $A = N \dot\cup T$ is called *tree-like* if it satisfies the conditions

1. *terminal ranking:* every terminal hyperedge has rank 2, i.e. for all $e \in E$ with $lab(e) \in T$, we have $rk(e) = 2$,

2. *context attachment:* for the sequence $e_1...e_n$ of nonterminal hyperedges in $E_\mathcal{H}$, there exists a sequence of attached vertices $\overline{context_\mathcal{H}} = a_1...a_n$, called the *context vertices*, such that $a_i \in [att_\mathcal{H}(e_i)] \setminus (free(\mathcal{H}) \cup \{root_\mathcal{H}\})$ for $i \in [1, n]$ and $a_i \neq a_j$ for $i \neq j$, and

3. *rootedness:* there exists an external vertex $root_\mathcal{H} \in [\overline{ext}]$ such that for each vertex $v \in V_\mathcal{H} \setminus free(\mathcal{H})$ there exists a path

$$\rho \triangleq root_\mathcal{H} = v_0 \overset{e_1}{—} v_1 \overset{e_2}{—} ... \overset{e_n}{—} v_n = v$$

from $root_\mathcal{H}$ to v in $\mathcal{H}$ such that $lab(e_i) \notin N$ and $v_j \notin [\overline{ext}] \cup [\overline{context}]$ for each $i \in [1, n]$ and $j \in [1, n-1]$.

■

The first condition is a simplification which allows us to represent tentacles of terminal hyperedges by variables in MSO_2F. Note that terminal hyperedges of rank higher than two can be simulated by using vertices with a special terminal label similar to the plain graph encoding presented in Definition 3.1.2. Thus, the terminal ranking condition is less restrictive than it seems.

The *rootedness* condition ensures that every tree-like hypergraph $\mathcal{H}$ contains a vertex such that every non-free vertex can be reached from this vertex by an undirected path containing only terminal hyperedges. This allows us to represent a production rule $\mathfrak{a} = \alpha \to \mathcal{A}$, where $\mathcal{A}$ is tree-like, by a single vertex $root_{\mathcal{A}}$. Thus, if all production rules of an HRG map to tree-like hypergraphs, extracting a derivation tree from a hypergraph $\mathcal{H}$ means to identify all root vertices in $\mathcal{H}$ and to label them with the corresponding production rules.

Now, assume that all root vertices can be reconstructed from a hypergraph $\mathcal{H}$. We still have to determine the relations between these vertices, i.e. find the children of each root vertex. Since every root vertex corresponds to a production rule $\mathfrak{a} = \alpha \to \mathcal{A}$ (its label), we have to identify the root vertices corresponding to derivation steps that replaced the nonterminals occurring in $\mathcal{A}$. In general, this is not possible. To see this, recall Example 3.2.1 in which an HRG is constructed for every context-free string grammar in Greibach normal form. This HRG contains nonterminals, e.g. β_k, that are only connected to free vertices. Intuitively, we cannot reconstruct that β_k belongs to a given root vertex r because a root vertex r_k representing a production rule to replace β_k is only reachable from r via the intermediate nonterminals $\beta_1, ..., \beta_{k-1}$. Since these nonterminals can be replaced by hypergraphs of arbitrary size, we cannot reconstruct the connection between r and r_k in MSO_2F.

For tree-like hypergraphs, the *context attachment* condition ensures that every nonterminal hyperedge is connected to at least one vertex, called the *context vertex*, which is reachable from the root vertex. A context vertex represents the position of a root vertex after replacing the nonterminal hyperedge it corresponds to. Hence, context vertices should be merged with root vertices when applying hyperedge replacement.

A tree-like hypergraph can be divided into three components as illustrated in Figure 5.6. The first component is a connected graph containing only terminal hyperedges

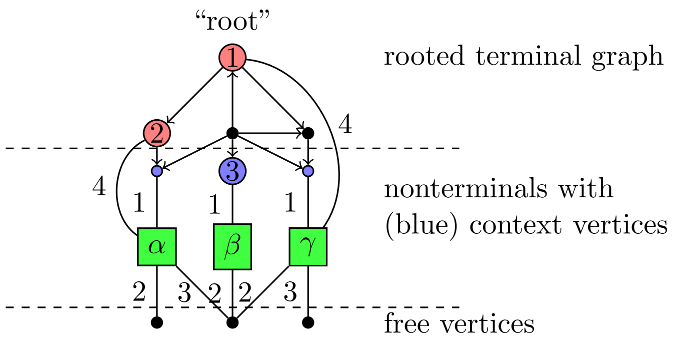


Figure 5.6.: Illustration of a tree-like hypergraph

with a distinguished root vertex. The context vertices (in blue) and their respective nonterminal hyperedges form the second component and must be reachable from the root vertex. They may have additional attachments to vertices in the graph and to free vertices which form the third component. Free vertices can be seen as "forward references". For example, the graphs in Figure 3.6 are tree-like and use free vertices to represent the leaves of a tree. Note that right-linear string grammars and regular tree grammars (see Definition 3.2.6) also require a strict separation between a terminal part (a single symbol or a tree) and a nonterminal part (a single nonterminal or at most one nonterminal at each leaf).

We stress that the context attachment condition requires that the root vertex and the context vertices of a tree-like hypergraph must be different. Otherwise, an HRG where all production rules map to tree-like hypergraphs may generate graph languages which are not definable in MSO_2F. For example, consider the HRG shown in Figure 5.7. This HRG realizes the language of "even stars" - a single vertex pointing to $2n$

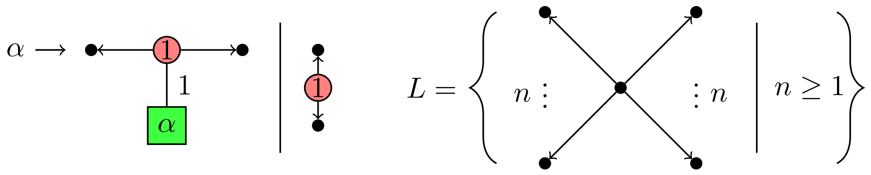


Figure 5.7.: An HRG generating even stars

other vertices - and cannot be specified in MSO_2F. Note that the left production rule of Figure 5.7 is not tree-like, because the single external vertex must correspond to the root as well as the context vertex of the only nonterminal hyperedge. We can define another HRG generating the language of even stars that works with two external vertices as illustrated in Figure 5.8. In this HRG, however, the left hypergraph is not

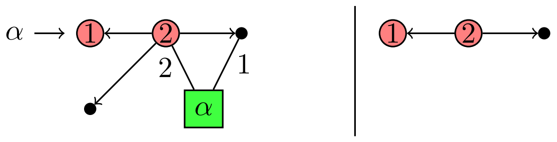


Figure 5.8.: Another HRG generating even stars

tree-like. To see this, note that choosing the first external vertex as root vertex violates the rootedness condition and choosing the second external vertex violates the context attachment condition, respectively. Thus, both conditions cannot be dropped in order to define a class of HRGs in which every production rule maps to tree-like hypergraphs and every language is MSO_2F definable.

Towards a formal definition of MSO_2F definable HRGs, we observe that there exists a spanning tree of the subgraph induced by the vertices $V_{\mathcal{H}} \setminus free(\mathcal{H})$ in $\mathcal{H}$ for every tree-like hypergraph $\mathcal{H}$. Furthermore, (a subtree of) such a spanning tree can be chosen such that $root_{\mathcal{H}}$ is the root vertex and all vertices in $\overline{context_{\mathcal{H}}}$ are leaves, because $\mathcal{H}$ is connected (see Figure 5.6). For each tree-like hypergraph $\mathcal{H}$, let $T_{\mathcal{H}}$ denote a minimal (w.r.t. inclusion) undirected tree in $\mathcal{H}$ with $root_{\mathcal{H}}$ as root and exactly $\overline{context_{\mathcal{H}}}$ as leaves. We additionally require that only the root vertex is taken as external vertex and all attachments of nonterminal hyperedges except for the respective context vertices are removed in this tree. Thus, we have $\overline{ext_{T_{\mathcal{H}}}} = root_{\mathcal{H}}$ and $att_{T_{\mathcal{H}}}(e)[1] \in [\overline{context_{\mathcal{H}}}]$ for each nonterminal hyperedge e. Moreover, we remember the original index of the external vertex $root_{\mathcal{H}}$ and the indices of $\overline{context_{\mathcal{H}}}$ as attached vertices. The result is a triple $(T_{\mathcal{H}}, k, \bar{l})$ such that $\overline{ext_{T_{\mathcal{H}}}}[1] = \overline{ext_{\mathcal{H}}}[k]$ and $att_{\mathcal{H}}(e_i)[\bar{l}[i]] = att_{T_{\mathcal{H}}}(e_i)[1]$ for each $i \in [1, \overline{context_{\mathcal{H}}}]$.

Then, a tree-like HRG is defined as follows.

Definition 5.2.2 (Tree-like HRG)**.** A *tree-like* HRG $\mathfrak{G} = (N, T, \mathfrak{P})$ is an HRG such that

1. for each production rule $\mathfrak{a} = \alpha \to \mathcal{A} \in \mathfrak{P}$, the hypergraph $\mathcal{A}$ is tree-like,
2. replacing every production rule $\mathfrak{a} = \alpha \to \mathcal{A}$ by $\alpha \to T_{\mathcal{A}}$ induces a regular tree grammar (see Definition 3.2.6) that respects the original indices of the triple $(T_{\mathcal{A}}, k, \bar{l})$, and ■

The set of all tree-like HRGs is denoted by $\mathcal{THRG}$ and the class of all hypergraph languages realizable by these HRGs is denoted by $\mathcal{THRL}$. Analogously to Chapter 3, the restriction of tree-like HRGs realizing only sets of heaps is denoted by $\mathcal{TDSG}$ and the corresponding class of hypergraph languages is denoted by $\mathcal{TDSL}$, respectively. The second condition is required to ensure that the composition of context vertices and root vertices really results in proper derivation trees. This is not guaranteed by having only production rules that map to tree-like hypergraphs. For instance, consider a counterexample consisting of three production rules shown in Figure 5.9. Below these production rules are the respective trees that should form a regular tree grammar. However, by Definition 5.2.1, there exists exactly one assignment of attached vertices to context vertices for each production rule due to the context attachment condition. Thus, there is a mismatch between external and attached vertices in the first two production rules, because the remaining vertex attached to β in the first production rule corresponds to the first attachment in the original graph ($\bar{l} = 1$) and the remaining external vertex in the second production rule corresponds to the second external vertex ($k = 2$), respectively. Hence, as illustrated by the examples in Figure 5.9 and Figure 5.7, no condition of tree-like HRGs can be dropped without adding undesirable hypergraph languages to the class of realizable languages.

We can observe that every root vertex of each production rule $\mathfrak{a} = \alpha \to \mathcal{A}$ corresponds to the same external vertex for all production rules with nonterminal α as left-hand side. Otherwise, no regular tree grammar can be obtained from these production rules as required in Definition 5.2.2. Note that there is one exception: If the initial nonterminal

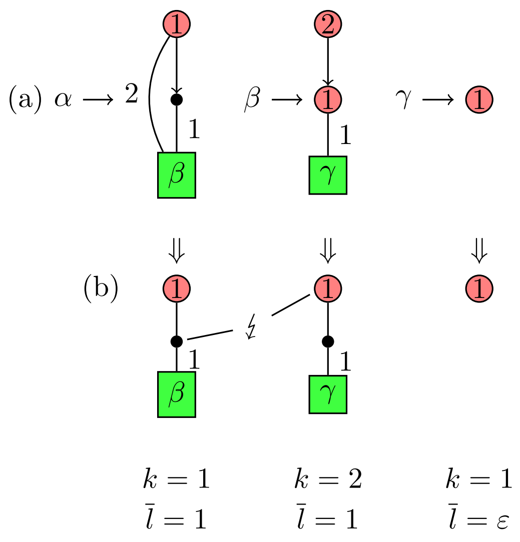


Figure 5.9.: (a) HRG containing no proper regular tree grammar, (b) only possible choice for a regular tree grammar (with a mismatch between attached and external vertices)

ξ never occurs on the right-hand side of a production rule, production rules corresponding to ξ may map to hypergraphs with different external vertices as root. However, in this case the order of external vertices does not matter at all. Thus, we can always change the order of vertices in the sequences of attached and external vertices. For convenience, we introduce a normal form for tree-like HRGs.

Definition 5.2.3 (Normalized Tree-like Hypergraph)**.** A tree-like hypergraph $\mathcal{H}$ is in normal form if $root_{\mathcal{H}} = \overline{ext_{\mathcal{H}}}[1]$ and for each vertex $v \in \overline{context_{\mathcal{H}}}$ corresponding to a nonterminal hyperedge e, we have $v = att_{\mathcal{H}}(e)[1]$. ■

Analogously, a tree-like HRG where every production rule maps to a normalized tree-like hypergraph is called a normalized tree-like HRG. By our previous observation, we can transform a given tree-like HRG into a normalized one as follows. For each nonterminal α, let k be the index of the external vertex that corresponds to the root vertex. Then, in every production rule the first and the k-th tentacle of every nonterminal hyperedge are switched. Furthermore, the first and the k-th external vertex of every production rule $\mathfrak{a} = \alpha \to \mathcal{A}$ are switched. This is obviously possible in polynomial time. For normalized tree-like hypergraphs, the indices of external and attached vertices coincide with the remembered original vertices, i.e. mismatches of external and attached vertices in the regular tree grammar are ruled out. From now on, we assume that every tree-like hypergraph is normalized. Our goal is to show the following theorem.

Theorem 5.2.1 (MSO_2L Definability of Tree-like HRGs)**.** *For every tree-like HRG $\mathfrak{G} = (N, T, \mathfrak{P})$ and every nonterminal $\xi \in N$, there exists an MSO_2F formula $\Psi_{\mathfrak{G}}^{\xi}$ such* *that for all finite hypergraphs $\mathcal{H}$ over $N \dot{\cup} T$, it holds that $\mathcal{H} \in L(\mathfrak{G}, \xi)$ if and only if $\underline{\mathcal{H}} \models \Psi^{\xi}_{\mathfrak{G}}$.* ■

Together with Proposition 4.1.2, this theorem implies decidability of the language inclusion problem for tree-like HRGs. For the rest of this section, let $\mathfrak{G} = (N, T, \mathfrak{P})$ be a tree-like HRG, $\xi \in N$ a nonterminal and $\xi^{\bullet} = \mathcal{H}_0 \xRightarrow{\mathfrak{a}_1}_{\mathfrak{G}} \ldots \xRightarrow{\mathfrak{a}_n}_{\mathfrak{G}} \mathcal{H}_n = \mathcal{H} \in L(\mathfrak{G}, \xi)$ a derivation of $\mathfrak{G}$. To simplify notation, we write $\mathfrak{a}$ instead of $\mathcal{A}$ whenever it is clear from the context that $\mathcal{A}$ is the right-hand side of a production rule $\mathfrak{a} = \alpha \to \mathcal{A}$. Analogously, $V_{\mathfrak{a}}$, $E_{\mathfrak{a}}$, $free(\mathfrak{a})$, etc. refers to $V_{\mathcal{A}}$, $E_{\mathcal{A}}$, $free(\mathcal{A})$, etc. In addition to that, let $\overline{\xi_{\mathfrak{a}}} = \xi_1 \ldots \xi_k$ denote a fixed sequence of labels of nonterminal hyperedges in $\mathfrak{a}$. Without loss of generality, we assume that for every production rule $\mathfrak{a} = \alpha \to \mathcal{A} \in \mathfrak{P}$ and every vertex $v \in V_{\mathfrak{a}}$, the set of outgoing edges of v, written $vOut_{\mathfrak{a}} = \{e \in E_{\mathfrak{a}} \mid v \in [src_{\mathfrak{a}}(e)]\}$, are in some fixed order $\overline{vOut_{\mathfrak{a}}} = e_1 \ldots e_n$. Analogously, we assume that all incoming edges of v in $\mathfrak{a}$, written $vIn_{\mathfrak{a}} = \{e \in E_{\mathfrak{a}} \mid v \in [tgt_{\mathfrak{a}}(e)]\}$, are negatively ordered $\overline{vIn_{\mathfrak{a}}} = e_{-1} e_{-2} \ldots e_{-k}$. Since there are only finitely many production rules and each production rule maps to a finite hypergraph, let

$$width_{\mathfrak{G}} = \max_{v \in V_{\mathfrak{a}}, \mathfrak{a} \in \mathfrak{P}} |vIn_{\mathfrak{a}}| + |vOut_{\mathfrak{a}}|$$

be the maximal number of ingoing and outgoing edges *inside* a single production rule for all vertices of all production rules in $\mathfrak{P}$. Furthermore, let $attached_{\mathfrak{a}}$ denote the set of all vertices in $\mathfrak{a}$ which are attached to some nonterminal hyperedge, $allocated_{\mathfrak{a}}$ be the set of vertices in $\mathfrak{a}$ with at least one outgoing terminal edge and $out_{\mathfrak{a}} = [\overline{ext_{\mathfrak{a}}}] \setminus \{root_{\mathfrak{a}}\}$ be the set of external vertices except the root, respectively. In order to keep MSO_2F formulae comprehensible, we introduce additional notation to denote certain types of hyperedges and tentacles as presented in Table 5.1.

Towards a proof of Theorem 5.2.1, we proceed as follows. At first, we assume that we already know all vertices in $\mathcal{H}$ corresponding to a vertex in a derivation tree as illustrated in Figure 5.3. By definition of tree-like HRGs, each of these vertices corresponds to $root_{\mathfrak{a}}$ for some production rule $\mathfrak{a} = \alpha \to \mathcal{A} \in \mathfrak{P}$.

In Section 5.2.1, we construct a formula $graph_{\mathfrak{a}}(\overline{D}, r)$ which takes such a root vertex $r = root_{\mathfrak{a}}$ and a labeling function dir represented by set variables $\overline{D}$ (see Chapter 2) and asserts that all vertices and hyperedges in $\mathfrak{a}$ are present, connected as in $\mathfrak{a}$ and all terminal edges are labeled according to the order of $vIn_{\mathfrak{a}}$ and $vOut_{\mathfrak{a}}$ for each vertex $v \in V_{\mathfrak{a}}$. Then, a path *inside* of $\mathfrak{a}$ can be uniquely specified by a finite word over the alphabet $Direction = \{-width_{\mathfrak{G}}, \ldots, -1\} \cup \{1, 2, \ldots, width_{\mathfrak{G}}\}$. We will use a fixed set of such words to find certain vertices like attached vertices and external vertices if only the root and the corresponding production rule is known.

Next, in Section 5.2.2, the notion of *path-connected* derivation trees is introduced as an extended form of derivation trees which have been presented in Definition 3.2.7. We show that every hypergraph generated by a tree-like HRG contains a path-connected derivation tree as a subgraph and provide MSO_2F formulae to characterize it.

After that, it remains to verify that external and attached vertices introduced by production rules are merged correctly as in an HRG derivation. We show how tree-walking automata running on path-connected derivation trees can be applied to verify

Edge Notation	MSO_2F Definition	Explanation
$v \circ\!\!\rightarrow e$	$source(v, e)$	e is an outgoing edge of v
$v \overset{\ell}{\circ\!\!\rightarrow} e$	$v \circ\!\!\rightarrow e \wedge e \in D_\ell$	e is the ℓ-th outgoing edge of v in some production rule
$e \rightarrow\!\!\circ\, v$	$target(e, v)$	e is an ingoing edge of v
$e \overset{\ell}{\rightarrow\!\!\circ}\, v$	$e \rightarrow\!\!\circ\, v \wedge e \in D_\ell$	e is the ℓ-th ingoing edge of v in some production rule
$u \overset{e,\ell}{\circ\!\!\rightarrow\!\!\circ}\, v$	$u \overset{\ell}{\circ\!\!\rightarrow} e \wedge e \rightarrow\!\!\circ\, v$	e is the ℓ-th directed edge connecting u and v
$u \circ\!\!\rightarrow\!\!\circ\, v$	$\exists e(u \circ\!\!\rightarrow e \wedge e \rightarrow\!\!\circ\, v)$	u has an outgoing edge leading to v
$u \overset{\ell}{\circ\!\!\rightarrow\!\!\circ}\, v$	$\exists e(u \overset{\ell}{\circ\!\!\rightarrow} e \wedge e \rightarrow\!\!\circ\, v)$	the ℓ-th outgoing edge of u leads to v
$u \circ\!\!-\!\!\circ\, v$	$u \circ\!\!\rightarrow\!\!\circ\, v \vee v \circ\!\!\rightarrow\!\!\circ\, u$	u and v are connected by some edge
$u \overset{\ell}{\circ\!\!-\!\!\circ}\, v$	$u \overset{\ell}{\circ\!\!\rightarrow\!\!\circ}\, v \vee v \overset{\ell}{\circ\!\!\rightarrow\!\!\circ}\, u$	u and v are connected by the ℓ-th edge of u
$u \overset{e,\ell}{\circ\!\!-\!\!\circ}\, v$	$u \overset{e,\ell}{\circ\!\!\rightarrow\!\!\circ}\, v \vee v \overset{e,\ell}{\circ\!\!\rightarrow\!\!\circ}\, u$	u and v are connected by the ℓ-th edge e of u

Table 5.1.: edge notations with corresponding MSO_2F definitions

correct hyperedge replacement for tree-like HRGs in Section 5.2.3.

Finally, we prove the correctness of the overall construction in Section 5.2.4.

5.2.1. Navigating through Production Rules

Assume that $\mathfrak{a} \rightarrow \mathcal{A} \in \mathfrak{P}$ is a production rule of $\mathfrak{G}$ and that a vertex $r \in V_\mathcal{H}$ corresponds to $root_\mathfrak{a}$ in a derivation of $\mathcal{H}$. Furthermore, let $dir : E_\mathcal{H} \rightharpoonup 2^{Direction}$ be a labeling function. Intuitively, this labeling function orders all ingoing and outgoing terminal edges defined inside a single production rule for each vertex. For a given vertex v, we call a terminal hyperedge e *positive* and assign it a positive label from $Direction$ if v is the source of e. Analogously, a terminal hyperedge e is called *negative* and gets a negative label if v is the target of e and v is not external. In addition to the direction, we remember the production rule $\mathfrak{a} \in \mathfrak{P}$ that introduced an edge e and its source and target in this production rule. The result is a tuple $(d, \mathfrak{a}, v_i, v_j)$ with $d \in Direction$, $\mathfrak{a} \in \mathfrak{P}$ and two vertices v_i, v_j occurring in $\mathfrak{a}$. Let $\overline{D}$ be the encoding of a labeling function mapping every hyperedge to a set of these tuples as described in Chapter 2. Then, $\overline{D}$ is a finite sequence, because the number of production rules is finite and only directions and vertices inside a production rule are considered. We now provide MSO_2F formulae to label all terminal hyperedges appropriately with directions in $\overline{D}$.

For simplicity, assume that $\{v_1, ..., v_n\} \subseteq V_\mathfrak{a}$ with $r = v_1$ is the set of non-free vertices in $\mathfrak{a}$ and $E_\mathfrak{a} = \{e_1, ..., e_m\}$ is the set of terminal hyperedges. It is straightforward to provide an MSO_2F formula which asserts that a hyperedge e has rank two with exactly one ingoing and one outgoing tentacle.

$$\begin{aligned} rank_2(e) \triangleq \exists^{\mathrm{v}}\, u\, \exists^{\mathrm{v}}\, v[& \\ & u \circ\!\!\rightarrow e \wedge e \rightarrow\!\!\circ\, v \\ & \wedge \forall x (x \circ\!\!\rightarrow e \rightarrow x = u \wedge e \rightarrow\!\!\circ\, x \rightarrow x = v) \\ &] \end{aligned}$$

Furthermore, we use a second auxiliary formula to ensure that a vertex or hyperedge is labeled only with the symbol a of a given alphabet $A = N \dot{\cup} T$.

$$label_a(x) \triangleq lab_a(x) \wedge \bigwedge_{b \in A \setminus \{a\}} \neg lab_b(x)$$

Since the labeling function dir is used to enumerate the ingoing and outgoing edges of any vertex inside the hypergraph $\mathfrak{a}$, we need a sanity condition asserting that every terminal hyperedge is labeled with exactly one positive direction, i.e. there is a label for every outgoing edge, and with exactly one negative label. Furthermore, we add the requirement that terminal hyperedges have rank two.

$$\begin{aligned} & edgeOrder(\overline{D}) \triangleq \forall^{\mathrm{e}}\, e[&& \text{(I.1)} \\ & \quad \wedge\, rank_2(e) && \text{(I.2)} \\ & \quad \wedge \bigvee_{\mathfrak{a} \in \mathfrak{P}, v_i, v_j \in V_{\mathfrak{a}} \setminus free(\mathfrak{a})} (&& \text{(I.3)} \\ & \quad \bigvee_{0 \le k \le width_{\mathfrak{G}}} e \in D_{(k, \mathfrak{a}, v_i, v_j)} \wedge \bigwedge_{(i, \mathfrak{b}, v_x, v_y) \ne (k, \mathfrak{a}, v_i, v_j), i > 0} e \notin D_{(i, \mathfrak{b}, v_x, v_y)} && \text{(I.4)} \\ & \quad \wedge \bigvee_{0 \le k \le width_{\mathfrak{G}}} e \in D_{(-k, \mathfrak{a}, v_i, v_j)} \wedge \bigwedge_{(-i, \mathfrak{b}, v_x, v_y) \ne (-k, \mathfrak{a}, v_i, v_j), i > 0} e \notin D_{(i, \mathfrak{b}, v_x, v_y)} && \text{(I.5)} \\ & \quad)] && \text{(I.6)} \end{aligned}$$

In order to ensure that no additional edges can occur in models of our formula, the following auxiliary formula is used to assert that every outgoing terminal hyperedge of a vertex v corresponds to one of the hyperedges $e_1, ..., e_m$.

$$outgoingEdges(v, e_1, ..., e_m) \triangleq \forall e (v \circ\!\!\rightarrow e \rightarrow \bigvee_{1 \le i \le k} e = e_i)$$

Analogously, we define a formula $ingoingEdges(v, e_1, ..., e_m)$ which is the same except that $e \rightarrow\!\!\circ\, v$ is used instead of $v \circ\!\!\rightarrow e$. With the help of these auxiliary formulae, we can now express the existence of a subgraph corresponding to $\mathfrak{a}$ with root r in the overall graph $\mathcal{H}$ as follows.

$$graph_{\mathfrak{a}}(\overline{D}, r) \triangleq \exists^{\mathrm{v}}\, v_1 ... \exists^{\mathrm{v}}\, v_n\, \exists^{\mathrm{e}}\, e_1 ... \exists^{\mathrm{e}}\, e_m [\tag{I.7}$$

(all non-free vertices and edges in $\mathfrak{a}$ exist)

$$r = v_1 \wedge \bigwedge_{i,j \in [1,n], i \neq j} v_i \neq v_j \wedge \bigwedge_{i,j \in [1,m], i \neq j} e_i \neq e_j \tag{I.8}$$

(vertices and edges are labeled as in $\mathfrak{a}$)

$$\wedge \bigwedge_{lab(v_i) = a, i \in [1,n]} label_a(v_i) \wedge \bigwedge_{lab(e_i) = a, i \in [1,m]} label_a(e_i) \tag{I.9}$$

($e_1, ..., e_m$ connect the same vertices as in $\mathfrak{a}$)

$$\wedge \bigwedge_{1 \leq k \leq m, e_k = \overline{v_i Out}[d] = \overline{v_j In}[d']} v_i \xrightarrow{(d, \mathfrak{a}, v_i, v_j)} e_k \wedge e_k \xrightarrow{(d', \mathfrak{a}, v_i, v_j)} v_j \tag{I.10}$$

(internal vertices of $\mathfrak{a}$ have no other edges)

$$\wedge \bigwedge_{v_i \in V_{\mathfrak{a}} \setminus (attached_{\mathfrak{a}} \cup [\overline{ext_{\mathfrak{a}}}])} (outgoingEdges(v_i, e_1, ..., e_m) \wedge \tag{I.11}$$

$$ingoingEdges(v_i, e_1, ..., e_m))$$

$$]$$

Although this formula looks quite complicated, all it does is to ensure that all vertices of $\mathfrak{a}$ with root r exist in $\mathcal{H}$, are connected properly and are ordered in the same way as $\mathfrak{a}$ is. We go through the formula step by step to clarify this. The first components I.7, I.8, and I.9 assert that all non-free vertices and all terminal hyperedges of hypergraph $\mathfrak{a}$ exist, are different and are labeled according to the labeling function of $\mathfrak{a}$. We omit free vertices of $\mathfrak{a}$, because their position is unclear at this stage and we will see later that it is sufficient to detect external vertices replacing free vertices. Next, I.10 requires that quantified vertices are at least attached to all edges defined in $\mathfrak{a}$ and that these edges are ordered as in $\mathfrak{a}$ by the labeling function *dir*. Finally, we ensure in I.11 that all internal vertices, i.e. neither external nor attached to nonterminals, are only attached to hyperedges labeled with terminal symbols that are introduced in the current production rule. Since these edges have rank two and their source as well as there target is specified in I.10, it follows that internal vertices are exactly connected to the edges specified by the production rule $\mathfrak{a} \in \mathfrak{P}$. Note that this is required for internal vertices only, because vertices attached to nonterminals and external vertices may be attached to hyperedges introduced by different production rules.

Now that we have a fixed ordering defined by $edgeOrder(\overline{D})$ and $graph_{\mathfrak{a}}(\overline{D}, r)$, finite words $\overline{w}$ over the alphabet $[\overline{D}]$ can be used to describe unique paths from $root_{\mathfrak{a}}$ to any non-free vertex in $\mathfrak{a}$. For each finite word $\overline{w} \in [\overline{D}]^{\star}$, the following formula asserts that y

is the vertex obtained from the path induced by $\overline{w}$ starting in vertex x.

$$path_{\overline{w}}(\overline{D}, x, y) \triangleq \begin{cases} x = y & \text{if } \overline{w} = \varepsilon \\ x \overset{\overline{w}}{\circ\!\!-\!\!\circ}\, y & \text{if } \overline{w} = v \in [\overline{D}] \\ \exists z (path_{\overline{u}}(x, z) \wedge z \overset{v}{\circ\!\!-\!\!\circ}\, y) & \text{if } \overline{w} = \overline{u}v, v \in [\overline{D}] \end{cases}$$

This formula can be extended to finite sets of words $W \subseteq [\overline{D}]^{\star}$ as follows.

$$path_{W}(\overline{D}, x, y) \triangleq \bigvee_{\overline{w} \in W} path_{\overline{w}}(x, y)$$

We stress that only a finite number of different paths can be used in our construction, because the recursive definition of $path_{w}(\overline{D}, x, y)$ requires the introduction of a new formula for each word $\overline{w} \in [\overline{D}]^{\star}$. For each production rule $\mathfrak{a} = \alpha \to \mathcal{A} \in \mathfrak{P}$, the following (not necessarily disjoint) finite sets of words will be used:

1. $W_{att_{\mathfrak{a}}(i)[j]}$ denotes the set of all words leading from $root_{\mathfrak{a}}$ to the j-th attached vertex of the i-th nonterminal hyperedge.
2. $W_{\overline{ext_{\mathfrak{a}}[i]}}$ denotes all words leading from $root_{\mathfrak{a}}$ to the i-th external vertex of $\mathfrak{a}$.
3. $W_{\mathfrak{a}[i]}$ denotes all words leading from $root_{\mathfrak{a}}$ to $\overline{V_{\mathfrak{a}}}[i]$.
4. $W_{fixed_{\mathfrak{a}}}$ denotes all words leading from $root_{\mathfrak{a}}$ to some vertex in $V_{\mathfrak{a}} \setminus [\overline{ext_{\mathfrak{a}}}]$.

Obviously, the union of the previously introduced finite sets is finite again and since the number of production rules is finite, the overall set of words is finite, too. We occasionally use a modified version of $path_{W}$ in which it is required that all vertices and hyperedges on a path belong to a given set variable X. This version is denoted by $path_{W}(\overline{D}, X, x, y)$. Since this adaption of $path_{W}$ is straightforward, it is left to the reader. Finally, we note that usually the first component of every symbol in the alphabet $[\overline{D}]$, i.e. a direction $d \in Direction$, is sufficient to navigate through production rules using finite words. The other components are mostly needed to deal with the case that different production rules add (outgoing) vertices to an external vertex.

5.2.2. Path-Connected Derivation Trees

Previously, we showed in Section 5.2.1 how the hypergraph induced by a production rule $\mathfrak{a} \in \mathfrak{P}$ can be specified in MSO_2F if the root vertex r and the production rule $\mathfrak{a}$ are known. Furthermore, we have seen how to navigate through the subgraph induced by r and a production rule $\mathfrak{a} \in \mathfrak{P}$ using sequences of directions. In this section, we clarify how these pairs of root vertices and production rules can be identified in the hypergraph $\mathcal{H}$ resulting from an HRG derivation. Intuitively, we want to identify the derivation tree (see Definition 3.2.7) belonging to $\mathcal{H}$. This corresponds to constructing the inverse image h^{-1} of Courcelle's function mapping trees to hypergraphs in the equationality theorem (see Proposition 5.1.1).

In general, a root vertex of a production rule $\mathfrak{a} = \alpha \to \mathcal{A}$ is not directly connected to the context vertices of the same production rule. Since the context vertices correspond to the root vertices of the next production rule in a derivation, this means that a derivation tree is not directly contained in a tree-like hypergraph. However, Definition 5.2.1 ensures for every context vertex $v \in [\overline{context_{\mathfrak{a}}}]$ of a production rule $\mathfrak{a} = \alpha \to \mathcal{A}$ that there exists a path inside of $\mathfrak{a}$ connecting v with $root_{\mathfrak{a}}$. Thus, we extend the definition of derivation trees such that intermediate vertices between two root vertices are allowed. These additional vertices are not labeled with a production rule and must belong to the same production rule as the closest labeled predecessor in the tree.

Definition 5.2.4 (Rooted Path-Connected Derivation Tree)**.** Let $\mathfrak{G} = (N, T, \mathfrak{P})$ be a tree-like HRG. For a given tree T and a labeling function $\ell : V_T \rightharpoonup \mathfrak{P}$, the set of *rooted path-connected derivation trees* with initial handle $\xi^\bullet$, written $\mathfrak{T}_{\mathfrak{G}}(\xi)$, is defined inductively as follows:

1. If a production rule $\mathfrak{a} = \alpha \to \mathcal{A}$ in $\mathfrak{P}$ contains no nonterminal hyperedges, the labeled tree $(root_{\mathfrak{a}}, \{\varepsilon \mapsto \mathfrak{a}\})$ is in $\mathfrak{T}_{\mathfrak{G}}(\xi)$.

2. Let $\mathfrak{a} = \alpha \to \mathcal{A}$ be a production rule with nonterminal hyperedges $\overline{\xi_{\mathfrak{a}}} = \xi_1 ... \xi_k$ such that (T_1, ℓ_1),...,(T_k, ℓ_k) are path-connected derivation trees in $\mathfrak{T}_{\mathfrak{G}}(\xi_1)$,...,$\mathfrak{T}_{\mathfrak{G}}(\xi_k)$. Furthermore, let T' be a minimal tree (w.r.t. inclusion) with $root_{\mathfrak{a}}$ as root and $\overline{context_{\mathfrak{a}}}$ as leaves. Then, for $T = T'[T_1/\overline{context_{\mathfrak{a}}}[1], ..., T_k/\overline{context_{\mathfrak{a}}}[k]]$ and

$$\ell : V_T \to \mathfrak{P}, v \mapsto \begin{cases} \mathfrak{a} & \text{if } v = root(T') \\ \ell_i(v) & \text{if } v \in T_i, 1 \leq i \leq k \\ \perp & \text{otherwise} \end{cases}$$

the pair (T, ℓ) is a path-connected derivation tree in $\mathfrak{T}_{\mathfrak{G}}(\xi)$.

■

Obviously, we obtain a derivation tree from a path-connected derivation tree by contracting all unlabeled vertices with the closest predecessor labeled with a production rule. Thus, a path-connected derivation tree of $\mathcal{H}$ can be used to reconstruct the HRG derivation of $\mathcal{H}$. The advantage of using path-connected derivation trees is that their existence in hypergraphs directly follows from the conditions of tree-like HRGs provided in Definition 5.2.2. Note that a path-connected derivation tree coincides with a derivation tree if the root vertex is directly connected with all context vertices for each production rule. For instance, this is the case for the (tree-like) HRG introduced in Example 3.2.2.

Our goal is to provide MSO_2F formulae such that a set variable T together with a sequence of set variables $\overline{N}$ encoding the labeling function ℓ (see Chapter 2) characterizes a path-connected derivation tree $(T, \ell_{\overline{N}})$. At first, we ensure that T is indeed a tree with a vertex r as root.

$$\begin{aligned}
\mathit{tree}(T,r) \triangleq & \neg \exists^{\mathrm{v}}\, x.x \circ\!\!-\!\!\circ_T r && \text{(II.1)}\\
& \wedge \forall^{\mathrm{v}}\, x.x \in T \rightarrow \mathit{closure}_{x \circ\!-\!\circ_T y}(r,x) && \text{(II.2)}\\
& \wedge \forall^{\mathrm{v}}\, x.x \neq r \rightarrow [\exists^{\mathrm{v}}\, y(y \circ\!\!-\!\!\circ_T x \wedge \forall^{\mathrm{v}}\, z.z \circ\!\!-\!\!\circ_T x \rightarrow y = z)] && \text{(II.3)}\\
& \wedge \forall^{\mathrm{v}}\, x \forall^{\mathrm{v}}\, y \forall^{\mathrm{v}}\, z \forall^{\mathrm{e}}\, e_1 \forall^{\mathrm{e}}\, e_2 [e_1 \neq e_2 \wedge x \overset{e_1}{\circ\!\!-\!\!\circ}_T y \wedge x \overset{e_2}{\circ\!\!-\!\!\circ}_T z] \rightarrow y \neq z && \text{(II.4)}
\end{aligned}$$

For an edge formula $\varphi(x,y)$ as defined in Table 5.1, let $\varphi_X(x,y)$ denote a modified formula in which it is additionally required that x, y and the intermediate hyperedge e are in X. The first line of the formula from above asserts that r is indeed the root of T, the second line ensures that T is connected by undirected edges, and the third component asserts that there exists a unique predecessor in T for each vertex in T except the root. Finally, we require that there is at most one edge in T connecting two vertices x, y in T. Analogously to label_a in the previous section, the auxiliary formula

$$\mathit{rootLabel}_\alpha(\overline{N},x) \triangleq \bigvee_{\mathfrak{b}\in\mathfrak{P}(\alpha)} x \in N_\mathfrak{b} \wedge \bigwedge_{\mathfrak{b}\neq\mathfrak{c}} x \notin N_\mathfrak{c}$$

is used to assign exactly one label to a vertex x. It remains to ensure that T is really a path-connected derivation tree and exactly the root vertices are labeled with production rules. We need two auxiliary formulae to specify this properly.

$$\begin{aligned}
\mathit{isAttached}_{\mathfrak{a},i,j}(\overline{D},\overline{N},r,x) &\triangleq \begin{cases} r \in N_\mathfrak{a} \wedge \mathit{path}_{W_{att_\mathfrak{a}(i)[j]}}(\overline{D},r,x) & \text{if } att_{\mathfrak{a}(i)[j]} \notin \mathit{free}(\mathfrak{a})\\ \mathit{false} & \text{otherwise}\end{cases}\\
\mathit{isRootAttached}_\alpha(\overline{D},\overline{N},x) &\triangleq \exists^{\mathrm{v}}\, r. \bigvee_{\mathfrak{b}\in\mathfrak{P},\alpha=\overline{\xi_\mathfrak{b}[i]}} \mathit{isAttached}_{\mathfrak{b},i,1}(\overline{D},\overline{N},r,x)
\end{aligned}$$

The first formula states that r is labeled with the production rule $\mathfrak{a}$ and x is the j-th vertex attached to the i-th nonterminal hyperedge of $\mathfrak{a}$. This formula is then used to state that x represents a context vertex of $\mathfrak{a}$ with root r and that the hyperedge x is attached to is labeled with the nonterminal α. Obviously, such a vertex x should be labeled with some production rule $\mathfrak{a} = \alpha \rightarrow \mathcal{A}$ in $\mathfrak{P}$. In fact every vertex except the root vertex of the first production rule must satisfy this requirement in order to be a root vertex of some production rule and therefore should be labeled. This characterization of labeled vertices is formalized in the following formula for the initial nonterminal symbol $\xi \in N$.

$$\begin{aligned}
\mathit{roots}_\xi(\overline{D},\overline{N},r) \triangleq & \bigwedge_{\alpha\in N} \forall^{\mathrm{v}}\, x\\
& \bigvee_{\mathfrak{a}\in\mathfrak{P}(\alpha)} x \in N_\mathfrak{a} \leftrightarrow \begin{cases} x = r \vee \mathit{isRootAttached}_\alpha(\overline{D},\overline{N},x) & \text{if } \alpha = \xi\\ \mathit{isRootAttached}_\alpha(\overline{D},\overline{N},x) & \text{if } \alpha \neq \xi\end{cases}
\end{aligned}$$

Furthermore, we make sure that a vertex is labeled with at most one production rule. Note that this is valid for tree-like HRGs, because context vertices are required to be different from the root vertex.

$$\mathit{uniqueRoots}(\overline{N}) \triangleq \bigwedge_{\mathfrak{a}\in\mathfrak{P}} \forall x.x \in N_{\mathfrak{a}} \rightarrow \bigwedge_{\mathfrak{a}\neq\mathfrak{b}} x \notin N_{\mathfrak{b}}$$

Finally, Definition 5.2.4 requires a path-connected derivation tree to be minimal (w.r.t. inclusion) and that all leaves are labeled with exactly one production rule. This is formalized by two formulae. The first formula requires that every vertex is either labeled with a production rule or is able to reach another labeled vertex in T (see II.6). The second formula makes sure that for each labeled vertex a path to all context vertices is included in T (see II.8). This results in the following formula to characterize path-connected derivation trees.

$$\mathit{derivationTree}(\overline{D}, \overline{N}, T, r) \triangleq \mathit{tree}(T, r) \tag{II.5}$$

$$\wedge \forall x.x \in T \rightarrow [\bigvee_{\mathfrak{a}\in\mathfrak{P}} x \in N_{\mathfrak{a}} \vee \exists y(\bigvee_{\mathfrak{a}\in\mathfrak{P}} y \in N_{\mathfrak{a}} \wedge \mathit{closure}_{x \circ\!\!-\!\!\circ_T y}(x, y))] \tag{II.6}$$

$$\wedge \bigwedge_{\mathfrak{a}\in\mathfrak{P}} \forall x \forall y.[x \in N_{\mathfrak{a}} \wedge \bigvee_{i\leq|\overline{\xi_{\mathfrak{a}}}|} \mathit{isAttached}_{\mathfrak{a},i,1}(\overline{D}, \overline{N}, x, y)] \rightarrow \tag{II.7}$$

$$\bigvee_{w\in W_{att_{\mathfrak{a}}(i)[1]}} \mathit{path}_w(\overline{D}, T, x, y) \tag{II.8}$$

We can combine the formulae introduced so far to an MSO_2F sentence Φ that asserts that a hypergraph $\mathcal{H}$ contains a path-connected derivation tree (T, l) and then apply the formula $\mathit{graph}_{\mathfrak{a}}(\overline{D}, r)$ to every root vertex r in T labeled with a production rule $\mathfrak{a} \in \mathfrak{P}$ to ensure that every vertex and every edge defined by this production rule exists. Every hypergraph $\mathcal{H} \in L(\mathfrak{G}, \xi)$ is also a model of Φ. However, the converse direction is not satisfied yet, because it is not guaranteed that every model of Φ is also in the language $L(\mathfrak{G}, \xi)$. The problem lies in an underspecification for external and attached vertices. While we assert for other vertices in $\mathit{graph}_{\mathfrak{a}}(\overline{D}, r)$ that no edges that are not defined by the production rule $\mathfrak{a}$ exist, we cannot do this locally for external and attached vertices, because arbitrarily many derivation steps may add edges to these vertices. Furthermore, by the definition of hyperedge replacement (see Definition 3.2.1), every external vertex must be merged with an attached vertex which we have not formalized in an MSO_2F sentence yet. Thus, some hypergraphs satisfying the formulae we defined so far may not merge the right external and attached vertices and therefore represent hypergraphs not in $L(\mathfrak{G}, \xi)$.

5.2.3. Verifying Hyperedge Replacement

Intuitively, every external vertex is a reference to some "real" vertex introduced in a previous derivation step. During an HRG derivation, these references are merged with attached vertices as explained in Section 3.2. The merging of attached and external vertices is not specified by the formulae introduced in the previous sections. Thus, some models of our formulae may have too many or improperly connected vertices. In this section, we show how we can verify (in MSO_2F) that a hypergraph is the result of an HRG derivation, i.e. that external and attached vertices are merged according to the rules of hyperedge replacement. The following notion is essential for this.

Definition 5.2.5 (Vertex Replacement Path)**.** For a given HRG derivation

$$\xi^\bullet = \mathcal{H}_0 \xRightarrow{\mathfrak{a}_1}_{\mathfrak{G}} \dots \xRightarrow{\mathfrak{a}_n}_{\mathfrak{G}} \mathcal{H}_n = \mathcal{H} \in L(\mathfrak{G}, \xi),$$

a *vertex replacement path* is a sequence $\overline{x} = x_1 \dots x_k$ of vertices such that $x_1 \in V_{\mathcal{H}_i}$ is an attached, non-external vertex for some $0 \leq i < n$, x_k is an external, non-attached vertex in $V_{\mathcal{H}_n}$ and $x_j \in V_{\mathcal{H}_{i+j}}$ is an attached and external vertex for all $0 \leq j < k$ such that x_j is replaced by x_{j+1} in the derivation step $\mathfrak{a}_{j+1}$.

We say that a vertex v *replaces* another vertex u in a derivation if there exists a vertex replacement path $\overline{x} = x_1 \dots . x_k$ containing both vertices such that v occurs after u in $\overline{x}$. ■

Intuitively, all vertices on a vertex replacement path denote the same vertex in a hypergraph $\mathcal{H}$, but at different stages of its derivation. Since an external vertex never adds a new vertex in a derivation step, but refers to an already existing vertex, we call the first vertex of a vertex replacement path the *real* vertex. Obviously, any external non-attached vertex in a derivation $\xi^\bullet \overset{\star}{\Longrightarrow}_{\mathfrak{G}} \mathcal{H} \in L(\mathfrak{G}, \xi)$ lies on exactly one vertex replacement path (see Definition 3.2.1). The key idea to check whether all external vertices are on a proper vertex replacement path is the following. Given a production rule $\mathfrak{a} = \alpha \to \mathcal{A}$ with some attached vertex $v_\mathfrak{a}$ in $\mathfrak{a}$ and a production rule $\mathfrak{b} = \beta \to \mathcal{B}$ with an external vertex $v_\mathfrak{b}$, we can easily decide whether it is possible that a derivation exists in which $v_\mathfrak{b}$ replaces $v_\mathfrak{a}$. More precisely, this is the case if and only if $v_\mathfrak{b} = att_\mathfrak{a}(i)[j]$ with $\overline{\xi_\mathfrak{a}}[i] = \beta$ and $v_\mathfrak{b} = \overline{ext_\mathfrak{b}}[j]$, i.e. $v_\mathfrak{a}$ is attached to a nonterminal hyperedge labeled with β and $v_\mathfrak{b}$ is the external vertex replacing this attached vertex. Note that only the two production rules $\mathfrak{a}$ and $\mathfrak{b}$ and the indices i and j are needed to verify this. Since all production rules of a derivation are collected in a path-connected derivation tree, all external vertices are on a proper vertex replacement path if and only if no path in a (path-connected) derivation tree contains a "bad tuple" $(\mathfrak{a}, \mathfrak{b}, i, j)$ of production rules which do not allow proper hyperedge replacement. We show that this property can be checked by a tree automaton running on path-connected derivation trees, which means the language of all trees without a bad tuple is a regular tree language (see Definition 3.2.6). As mentioned in Example 4.1.2, there exists an MSO_2F sentence φ for every regular tree language T such that $L(\varphi) = T$. Thus, we can provide a tree automaton running on path-connected derivation trees instead of an MSO_2F sentence.

For our purpose, a special tree automaton called a *tree-walking automaton* is well-suited, because it can traverse a tree step-by-step looking for bad pairs. This automaton model was already introduced in [AU71] and its expressive power was clarified in [NS00]: Every tree language realized by a tree-walking automata is regular. The converse does not hold.

Definition 5.2.6 (Tree-walking Automaton)**.** Let A be a finite alphabet ranked by a function $rk : A \rightharpoonup \mathbb{N}$. Furthermore, let $Types = \{root\} \dot\cup \{1, ..., max_{a \in A} rk(a)\}$ be the set of all possible positions of a vertex in a tree relative to its parent, i.e. the type of a vertex v is k if it is the k-th child of its parent vertex and $root$ otherwise. For a symbol $a \in A$, let $Dir_a = \{\uparrow, 0, 1, 2, ..., rk(a)\}$ be the set of possible directions. A *tree-walking automaton* (TWA) is a tuple $\mathfrak{A} = (Q, A, q_0, \Delta, F)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final states and $\Delta \subseteq Q \times A \times Types \times Q \times Dir_A$ is a transition relation. ■

Given a tree t, a TWA configuration is a pair $(v, p) \in dom(t) \times Q$. A *run* $\overline{\rho}$ of a TWA $\mathfrak{A}$ on a tree t is a sequence of configurations $\overline{\rho} = (v_0, p_0)(v_1, p_1)...$, where $v_0 = \varepsilon$ is the root of t, $p_0 = q_0$ is the initial state such that successive configurations satisfy the following conditions:

- If the vertex v_{i+1} is the j-th child of v_i, i.e. $v_{i+1} = v_{i.j}$, there exists a transition $(q_i, type(v_i), t(v_i), q_{i+1}, j) \in \Delta$,
- if $v_i = v_{(i+1).j}$ for some j, i.e. the vertex v_{i+1} is the parent of v_i, there exists a transition $(q_i, type(v_i), t(v_i), q_{i+1}, \uparrow) \in \Delta$, and
- if $v_{i+1} = v_i$, there exists a transition $(q_i, type(v_i), t(v_i), q_{i+1},) \in \Delta$.

We call a run $\overline{\rho}$ *accepting* if it is finite and the state component of the last configuration is in the set of final states F. The language accepted by a TWA $\mathfrak{A}$ is the set of all trees t such that $\mathfrak{A}$ has an accepting run on t.

Example 5.2.1. *A useful building block to construct more complex tree languages with tree-walking automata is a TWA which traverses all vertices of a tree in a depth-first search and terminates at the root again. Let A be a ranked alphabet and n the maximal rank of all symbols in A. A TWA, which starts the depth-first search in a state p and terminates in a state q, can be defined as $\mathfrak{D}_{p,q,A,n} = (\{p\} \times \{0, 1, ..., n\} \cup \{q\}, A, (p, 0), \Delta)$, where the transition relation Δ is the union of the following sets. Intuitively, the second component of the state space counts which subtrees (from left to right) have already been traversed.*

- *If no subtree has been traversed yet, go into the leftmost one, i.e. $((p, 0), type, a, (p, 0), 1) \in \Delta$ for $type \in Types, a \in A$.*
- *In the i-th leaf v, the i-th subtree of the parent of v is traversed, i.e. $((p, 0), i, a, (p, i), \uparrow) \in \Delta$ for $rk(a) = 0$.*
- *If some subtree has not been traversed yet, go into the next subtree, i.e. $((p, i), type, a, (p, 0), i + 1) \in \Delta$ for $type \in Types, i < rk(a)$.*
- *If all subtrees have been traversed, switch to the parent and mark this subtree as traversed, i.e. $((p, i), j, a, (p, j), \uparrow) \in \Delta$ for $i = rk(a)$.*

- *If all subtrees of the root have been traversed, terminate, i.e.* $((p,i), root, a, q, 0)$ *for* $i = rk(a)$.

∎

In order to verify that external and attached vertices are merged correctly, we construct a TWA $\mathfrak{A}_\ell$ that runs on a path-connected derivation tree and checks whether the ℓ-th external vertex occurring in a production rule $\mathfrak{a} = \alpha \to \mathcal{A}$ applied in some derivation step lies on a vertex replacement path. Furthermore, the real vertex of this vertex replacement path is required to be a non-free vertex. The case of free real vertices is treated later. Intuitively, $\mathfrak{A}_\ell$ runs on a path-connected derivation tree in two phases.

In the first phase, the automaton traverses the tree in a depth-first search as described in Example 5.2.1 and nondeterministically selects a vertex labeled with a production rule that has at least ℓ external vertices. Then, the automaton enters the second phase to check whether the ℓ-th external vertex, which belongs to the production rule and derivation step of the selected vertex, lies on a vertex replacement path.

In the second phase, $\mathfrak{A}_\ell$ walks along the unique path from the selected vertex to the root of the tree. Whenever a vertex labeled with a production rule $\mathfrak{b} \in \mathfrak{P}$ is visited, the automaton checks whether the last visited production rule $\mathfrak{a}$ can replace a nonterminal hyperedge in $\mathfrak{b}$. Therefore, the last visited production rule and the index of the external vertex to be checked are saved in the state space. Since a production rule may contain multiple nonterminal hyperedges labeled with the same nonterminal, the automaton needs to identify the nonterminal hyperedge that should be replaced using the saved production rule. It also saves the path from the last labeled vertex to the current vertex in its state space. This path between two labeled vertices corresponds to the path of the root vertex to one of its context vertices for the production rule the current vertex is labeled with. Note that there are only finitely many of such paths without a cycle from the root of a hypergraph to one of its context vertices in a production rule $\mathfrak{a} = \alpha \to \mathcal{A}$ of a tree-like HRG $\mathfrak{G} = (N, T, \mathfrak{P})$, i.e. the maximal length $\Bbbk$ of such paths for all production rules is bounded. Thus, the nonterminal hyperedge to be replaced can be detected using this path. Formally, let $W_\mathfrak{G} = \{1, ..., width_\mathfrak{G}\}$, $-W_\mathfrak{G} = \{-i \mid i \in W_\mathfrak{G}\}$ and $W = W_\mathfrak{G} \cup -W_\mathfrak{G}$ denote sets of directions as defined in Section 5.2.1. The index of the nonterminal hyperedge of a production rule $\mathfrak{a} = \alpha \to \mathcal{A}$ to be replaced for a given path of directions $\overline{w} \in W^{\leq \Bbbk}$ is given by the mapping

$$\begin{aligned} nt_\mathfrak{a} : W^{\leq \Bbbk} &\rightharpoonup \max_{\mathfrak{b} \in \mathfrak{P}} \{|\overline{context_\mathfrak{b}}|\}, \\ \overline{w} &\mapsto k \ \text{ iff } \ \overline{w} \text{ defines a path from } root_\mathfrak{a} \text{ to } \overline{context_\mathfrak{a}}[k] \end{aligned}$$

It is straightforward to see that for each production rule $\mathfrak{a} \in \mathfrak{P}$, the partial function $nt_\mathfrak{a}$ is finite.

Given two production rules $\mathfrak{a}, \mathfrak{b} \in \mathfrak{P}$, the index i of an external vertex and a path $\overline{w}$ between the vertices labeled with $\mathfrak{a}$ and $\mathfrak{b}$, the automaton $\mathfrak{A}_\ell$ checks whether the i-th external vertex of $\mathfrak{a}$ replaces the i-th attached vertex of a nonterminal hyperedge with a context vertex that can be reached by following the path $\overline{w}$ starting in the root vertex of $\mathfrak{b}$. If this is the case and the attached vertex is external again, the automaton updates its saved production rule and the index of the external vertex to be checked and proceeds as before. If the attached vertex is not external and not free, the real vertex of a vertex replacement path has been reached, i.e. $\mathfrak{A}_\ell$ can accept. Otherwise, the automaton rejects.

In addition to that, $\mathfrak{A}_\ell$ deals with the corner case that all vertex replacement paths containing a vertex v consist only of vertices without outgoing edges. Similar to the outgoing edges of external vertices, this cannot be checked locally by the formula $graph_\mathfrak{a}$ for a production rule $\mathfrak{a} \in \mathfrak{P}$. Thus, we add one more component b to the states of the TWA $\mathfrak{A}_\ell$ which is 1 if and only if some vertex on a vertex replacement path visited by our automaton has outgoing edges. The following predicate will be used to formalize this in the transition relation of $\mathfrak{A}_\ell$.

$$d : \bigcup_{0 \le i \le n} V_{\mathcal{H}_i} \times \{0, ..., \Bbbk\} \to \{0,1\}, (v,b) \mapsto \begin{cases} 1 & |vOut| > 0 \text{ or } b = 1 \\ 0 & \text{otherwise} \end{cases}$$

Thus, no vertex on a vertex replacement path has outgoing edges if $\mathfrak{A}_\ell$ accepts in state with a component $b = 0$. We now formally define the automaton $\mathfrak{A}_\ell$.

Definition 5.2.7 (TWA for non-free attached vertices)**.** Let $\mathfrak{R}$ be the maximal rank of all nonterminals in a tree-like HRG $\mathfrak{G} = (N, T, \mathfrak{P})$ and $\ell \le \mathfrak{R}$. The tree-walking automaton $\mathfrak{A}_\ell = (Q, \mathfrak{P} \cup \{\tau\}, (q_{sel}, 0), \Delta, F)$ is defined as follows. The state set Q is given by the union of the following sets:

- All states of the TWA $\mathfrak{D}_{q_{sel}, q_{reject}, \Gamma, 2\cdot|width_\mathfrak{G}|}$ (see Example 5.2.1) are in Q and form the first phase in which the automaton traverses a tree in a depth-first search:

$$Q(\mathfrak{D}_{q_{sel}, q_{reject}, \Gamma, 2\cdot|width_\mathfrak{G}|})$$

- For each production rule $\mathfrak{a} \in \mathfrak{P}$, a state $(q_{ext}, \mathfrak{a})$ denotes that a vertex to check whether the ℓ-th external vertex lies on a vertex replacement path has been selected:

$$\{(q_{ext}, \mathfrak{a}) \mid \mathfrak{a} \in \mathfrak{P}\}$$

- While verifying the selected path, the automaton saves in a component b whether some vertex on the vertex replacement path visited so far has outgoing edges. Furthermore, the last visited production rule $\mathfrak{a} \in \mathfrak{P}$, the index i of the external vertex to be checked and the path of directions $\overline{w}$ taken since a labeled vertex has been visited are saved in the state:

$$\{(q_{ext}, b, \mathfrak{a}, i, \overline{w}) \mid b \in \{0,1\}, \mathfrak{a} \in \mathfrak{P}, i \in [1, rk(\mathfrak{a})], \overline{w} \in W^{\le \Bbbk}\}$$

- If the saved external vertex is replaced by a non-free vertex, the automaton marks this by entering a special state with q_{in} as first component and the index k of the hyperedge that is replaced. This is a technical trick to simplify the construction of MSO_2F formulae verifying the merging of external and attached vertices:

$$\{(q_{in}, b, \mathfrak{a}, i, k) \mid b \in \{0,1\}, \mathfrak{a} \in \mathfrak{P}, i \in [1, rk(\mathfrak{a})], k \le \mathfrak{R}\}$$

- The set of final states consists of the same components (except the first component) as the previous set:

$$\{(q_{att}, b, \mathfrak{a}, i, k) \mid b \in \{0,1\}, \mathfrak{a} \in \mathfrak{P}, i \in [1, rk(\mathfrak{a})], k \leq \mathfrak{R}\}$$

We assume w.l.o.g. that $Types = \{root\} \cup W$ and use $type \in Types$ and $\mathfrak{a} \in \mathfrak{P}$ as placeholders. The transition relation Δ of $\mathfrak{A}_\ell$ is defined as the union of the following sets:

- All transitions of the automaton $\mathfrak{D}_{q_{sel}, q_{reject}, \Gamma, 2 \cdot |width_{\mathfrak{G}}|}$ (see Example 5.2.1) realizing a depth-first search are in Δ.

- Nondeterministically select a labeled vertex while traversing a tree to go into the second phase:
$$((q_{sel}, i), type, \mathfrak{a}, (q_{ext}, \mathfrak{a}), 0) \in \Delta \text{ for } \ell \leq rk(\mathfrak{a})$$

- Start searching for the next labeled vertex in the path-connected derivation tree:

$$((q_{ext}, \mathfrak{a}), type, \mathfrak{a}, (q_{ext}, d(v, 0), \mathfrak{a}, \ell, \varepsilon), 0) \in \Delta \text{ for } v = \overline{ext}_{\mathfrak{a}}[\ell]$$

- Collect the directions taken while moving to the next labeled vertex:

$$((q_{ext}, b, \mathfrak{a}, i, \overline{w}), type, \tau, (q_{ext}, b, \mathfrak{a}, i, type \cdot \overline{w}), \uparrow) \in \Delta \text{ for } \overline{w} \in W^{<\Bbbk}, type \leq rk(\mathfrak{a})$$

- The pair $(\mathfrak{a}, \mathfrak{c})$ is compatible for the i-th attached non-free vertex of the nonterminal determined by the path $\overline{w}$ in the path-connected derivation tree. Since the attached vertex is non-free, it is marked by entering a state with q_{in} as first component:

$$\begin{aligned}&((q_{ext}, b, \mathfrak{a}, i, \overline{w}), type, \mathfrak{c}, (q_{in}, d(v, b), \mathfrak{c}, j, k), 0) \in \Delta \text{ for}\\ &\overline{w} \in W^{\leq\Bbbk}, nt_{\mathfrak{c}}(type \cdot \overline{w}) = k, v = \overline{ext_{\mathfrak{c}}}[j] = att_{\mathfrak{c}}(\overline{\xi_{\mathfrak{c}}}[k])[i] \notin free(\mathfrak{c})\end{aligned}$$

- The pair $(\mathfrak{a}, \mathfrak{c})$ is compatible for the i-th attached free vertex of the nonterminal determined by the path $\overline{w}$ in the path-connected derivation tree:

$$\begin{aligned}&((q_{ext}, b, \mathfrak{a}, i, \overline{w}), type, \mathfrak{c}, (q_{ext}, d(v, b), \mathfrak{c}, j, \varepsilon), \uparrow) \in \Delta \text{ for}\\ &\overline{w} \in W^{\leq\Bbbk}, nt_{\mathfrak{c}}(type \cdot \overline{w}) = k, v = \overline{ext_{\mathfrak{c}}}[j] = att_{\mathfrak{c}}(\overline{\xi_{\mathfrak{c}}}[k])[i] \notin free(\mathfrak{c})\end{aligned}$$

- Proceed for non-free vertices as for free vertices:

$$((q_{in}, b, \mathfrak{a}, i, k), type, \mathfrak{c}, (q_{ext}, d(v, b), \mathfrak{c}, j, \varepsilon), \uparrow) \in \Delta \text{ for } v = \overline{ext_{\mathfrak{c}}}[j] = att_{\mathfrak{c}}(\overline{\xi_{\mathfrak{c}}}[k])[i]$$

- Accept if the beginning of a vertex replacement path is reached:

$$\begin{aligned}&((q_{ext}, b, \mathfrak{a}, i, \overline{w}), type, \mathfrak{c}), (q_{att}, d(v, b), \mathfrak{c}, i, k) \in \Delta \text{ for}\\ &\overline{w} \in W^{\leq\Bbbk}, nt_{\mathfrak{c}}(type \cdot \overline{w}) = k, v = att_{\mathfrak{c}}(\overline{\xi_{\mathfrak{c}}}[k])[i] \text{ and}\\ &v \notin [\overline{ext_{\mathfrak{c}}}] \cup free(\mathfrak{c}) \text{ or } type = root\end{aligned}$$

■

A sketch of an accepting run of the TWA $\mathfrak{A}_\ell$ on a path-connected derivation tree is shown in Figure 5.10. Here, blue vertices denote root vertices labeled with some production rule. For convenience, we assume that all production rules applied are different such that we do not have to distinguish between the label of a root vertex and the vertex itself. Squirreled arrows indicate that two vertices are connected via a path instead of a single edge. The traversal of the tree in the first phase of $\mathfrak{A}_\ell$ is colored in gray. The automaton

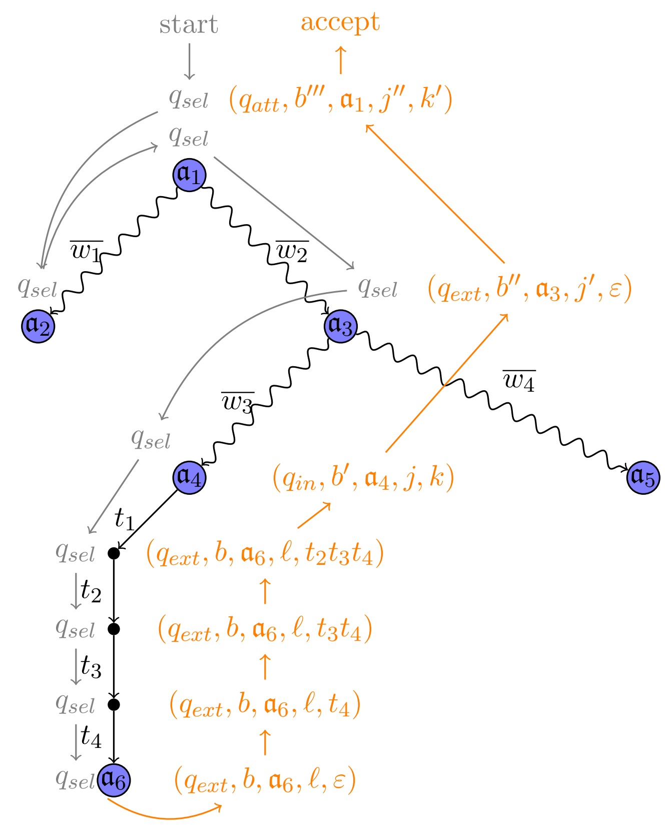

start
accept
q_{sel}
$(q_{att}, b''', \mathfrak{a}_1, j'', k')$
$\mathfrak{a}_1$
$\overline{w_1}$
$\overline{w_2}$
$\mathfrak{a}_2$
$(q_{ext}, b'', \mathfrak{a}_3, j', \varepsilon)$
$\mathfrak{a}_3$
$\overline{w_3}$
$\overline{w_4}$
$\mathfrak{a}_4$
$\mathfrak{a}_5$
$(q_{in}, b', \mathfrak{a}_4, j, k)$
t_1
$(q_{ext}, b, \mathfrak{a}_6, \ell, t_2t_3t_4)$
t_2
$(q_{ext}, b, \mathfrak{a}_6, \ell, t_3t_4)$
t_3
$(q_{ext}, b, \mathfrak{a}_6, \ell, t_4)$
t_4
$\mathfrak{a}_6$
$(q_{ext}, b, \mathfrak{a}_6, \ell, \varepsilon)$

Figure 5.10.: Sketch of a run of $\mathfrak{A}_\ell$ on a path-connected derivation tree verifying the vertex replacement path $att_{\mathfrak{a}_1}[k'](j'') \cdot \overline{ext_{\mathfrak{a}_3}}[j'] \cdot \overline{ext_{\mathfrak{a}_4}}[j] \cdot \overline{ext_{\mathfrak{a}_6}}[\ell]$

selects the vertex labeled with $\mathfrak{a}_6$ and switches to the second phase - colored in orange - to find a vertex replacement path for the ℓ-th external vertex of $\mathfrak{a}_6$. In the vertex labeled with $\mathfrak{a}_4$, a non-free attached vertex is found that lies on the same vertex replacement path. The next labeled vertex is labeled with $\mathfrak{a}_3$ and contains a free attached vertex on the vertex replacement path, i.e. no state with a component q_{in} is entered. Finally, the automaton finds a non-external attached vertex in the vertex labeled $\mathfrak{a}_1$ and accepts.

We still have to show that we constructed a correct TWA, i.e. $\mathfrak{A}_\ell$ should accept if and only if the path visited after reaching a state $(q_{ext}, \mathfrak{a})$, $\mathfrak{a} \in \mathfrak{P}$ corresponds to a vertex replacement path. Let $(T, l) \in \mathfrak{T}_{\mathfrak{G}}(\xi)$ be a path-connected derivation tree of $\mathfrak{G}$ (see Definition 5.2.4) and $\Theta = \mathcal{H}_0 \overset{r_1,\mathfrak{a}_1}{\Longrightarrow}_{\mathfrak{G}} \mathcal{H}_1 \overset{r_2,\mathfrak{a}_2}{\Longrightarrow}_{\mathfrak{G}} ... \overset{r_n,\mathfrak{a}_n}{\Longrightarrow}_{\mathfrak{G}} \mathcal{H}_1$ be a fragment of a corresponding derivation where $r_i = root_{\mathfrak{a}_i}$, $i \in [1, n]$, is a vertex which uniquely determines the nonterminal hyperedge to be replaced. In addition to that, we write $\mathcal{H}(r_i)$ to denote the restriction of $\mathcal{H}_i$ to vertices and edges defined in the production rule $\mathfrak{a}_i$. Since all of these vertices and edges are in $\mathcal{H}_i$, terms like $next(r_i, j, k) = att_{\mathcal{H}(r_i)}(\overline{\alpha}_{\mathcal{H}(r_i)}[k])[j]$ for $k \leq |\overline{\alpha}_{\mathcal{H}(r_i)}|$ and $j \leq rk(\overline{\alpha}_{\mathcal{H}(r_i)}[k])$ to denote the successor vertex of r_i on a vertex replacement path are well-defined. For a run fragment $\overline{\rho}$ of a TWA, let $\widetilde{\rho}$ denote the subsequence of $\overline{\rho}$ in which consecutive configurations with equal vertices are merged and all configurations containing unlabeled vertices are removed. This transformation of a run is required, because some transitions of the automaton $\mathfrak{A}_\ell$ stay in the current vertex. The correctness of $\mathfrak{A}_\ell$ is is formalized by the following lemma.

Lemma 5.2.1. *Let $(T, l) \in \mathfrak{T}_{\mathfrak{G}}(\xi)$ be a path-connected derivation tree of an HRG $\mathfrak{G} = (N, T, \mathfrak{P})$ and $\Theta = \mathcal{H}_0 \overset{r_1,\mathfrak{a}_1}{\Longrightarrow}_{\mathfrak{G}} \mathcal{H}_1 \overset{r_2,\mathfrak{a}_2}{\Longrightarrow}_{\mathfrak{G}} ... \overset{r_n,\mathfrak{a}_n}{\Longrightarrow}_{\mathfrak{G}} \mathcal{H}_1$ be a derivation of $\mathfrak{G}$ as defined above that corresponds to (T, l). Furthermore, let $\overline{\rho} = \rho_1...\rho_n$ be a run fragment of $\mathfrak{A}_\ell$ on (T, l) with vertices $\overline{v} = v_1....v_n$ as first component and states $\overline{p} = p_1...p_n$ as second component such that $p_1 = (q_{ext}, b, \mathfrak{a}, j, \varepsilon)$ for some $b \in \{0, 1\}$, $\mathfrak{a} \in \mathfrak{P}$, and $j \leq rk(\mathfrak{a})$. Then, for mapping*

$$vr : V_T^\star \rightharpoonup (\bigcup_{1 \leq i \leq n} V_{\mathcal{H}_i})^\star \textit{ with}$$

$$vr(v \cdot \overline{w}) = \begin{cases} \overline{ext_{\mathcal{H}}}[j] & \mathfrak{A}_\ell \textit{ visits } v \textit{ in a state } (q_{ext}, b, \mathfrak{a}, j, \varepsilon) \in \Delta \\ & \textit{and } \overline{w} = \varepsilon \\ vr(\overline{w}) \cdot next(v, j, nt_{\mathfrak{a}}(v \cdot \overline{w})) & \mathfrak{A}_\ell \textit{ visits } v \textit{ in a state } (q_{ext}, b, \mathfrak{a}, j, \overline{w}) \in \Delta \\ & \textit{and } \overline{w} \neq \varepsilon, \end{cases}$$

the run $\overline{\rho}$ is accepting if and only if $vr(\widetilde{v \cdot \overline{w}})$ is a vertex replacement path in Θ with a non-free real vertex. ■

Intuitively, the lemma from above states that every accepting run of $\mathfrak{A}_\ell$ can be associated with a vertex replacement path defined by the mapping vr.

Proof (of Lemma 5.2.1). We show the claim by complete induction over the length of $\widetilde{v}$. For $|\widetilde{v}| < 2$ there is nothing to show. Thus, assume the claim holds for all $\widetilde{v}$ with $|\widetilde{v}| = n \in \mathbb{N}$ and consider a fragment $\widetilde{v} = xy\overline{z}$ with $|\widetilde{y\overline{z}}| = n$ and $|\widetilde{x}| = 1 = |\widetilde{y}|$.

"$\Rightarrow$": By assumption, $\mathfrak{A}_\ell$ is in a state $(q_{ext}, \mathfrak{a})$ in a vertex x and the automaton can move to a state $(q_{ext}, b, \mathfrak{a}, \ell, \varepsilon)$. By construction this implies that $\overline{ext_{\mathcal{H}(x)}}[\ell]$ exists. Since $\overline{v}$ corresponds to a run fragment $\overline{\rho}$, the automaton eventually reaches a state $(q_{ext}, b', \mathfrak{a}, \ell, \overline{u})$ without visiting any vertices labeled with a production rule. Hence, a transition to a state $(q_{ext}, b'', \mathfrak{b}, j, \varepsilon)$ (possibly via a state $(q_{in}, b'', \mathfrak{b}, j, k)$) is enabled for the automaton such that a vertex y labeled with $\mathfrak{b}$ is reached. By construction, the i-th attached vertex of the k-th nonterminal in $\mathcal{H}(y)$ replaces the ℓ-th external vertex in $\mathcal{H}(x)$. Hence, $vr(xy)$ describes a valid vertex replacement path (up to now, this is equal to the case $|\widetilde{v}|= 2$). By assumption, $\overline{z}$ corresponds to an accepting run fragment of $\mathfrak{A}_\ell$ and by induction hypothesis $vr(\widetilde{yz})$ is a vertex replacement path in Θ. Since, every vertex can only reach one state of the form $(q_{ext}, b, \mathfrak{a}, j, \varepsilon)$ (the automaton moves upwards only), and the i-th attached vertex of the k-th nonterminal in $\mathcal{H}(y)$ is equal to the external vertex $vr(y)$, it follows that $vr(xy\overline{z})$ is a valid vertex replacement path in Θ.

"$\Leftarrow$": Assume $vr(\widetilde{v})$ is a vertex replacement path in Θ. By definition, $vr(\widetilde{v}) = vr(\widetilde{y}) \cdot next(y, i, nt_\mathfrak{a}(y \cdot \overline{z})) \cdot \overline{ext_{\mathcal{H}(x)}}[j]$. Since $next(y, i, nt_\mathfrak{a}(y \cdot \overline{z})) \cdot \overline{ext_{\mathcal{H}(x)}}[i]$ is a vertex replacement, $\overline{ext_{\mathcal{H}(x)}}[i]$ must exist in $\mathcal{H}(x)$ and y is the closest predecessor of x in T which is labeled with a production rule. Hence, there is a unique path $\overline{\pi}$ from x to y in T such that the automaton only moves upwards as in the definition of $\mathfrak{A}_\ell$. Thus, the automaton can reach y in a state $(q_{ext}, b, \mathfrak{a}, j, \overline{u})$ from vertex x in a state $(q_{ext}, b, \mathfrak{a}, j, \varepsilon)$. Then, a transition (or two for non-free vertices) is enabled such that $\mathfrak{A}_\ell$ reaches a state $s = (q_{ext}, b', \mathfrak{b}, i, \varepsilon)$ in vertex x. Now, by induction hypothesis, there must be an accepting run fragment $\overline{\rho}$ of $\mathfrak{A}_\ell$ starting in vertex y and state s. Hence, the concatenation of the run fragment $\overline{\pi}$ to reach y from x and $\overline{\rho}$ yields an accepting run starting in vertex x. □

Note that $\mathfrak{A}_\ell$ traverses a tree in depth-first search first and can nondeterministically switch to a state $(q_{ext}, \mathfrak{a}) \in Q$ for any visited vertex labeled $\mathfrak{a}$. From this state, the automaton can only move to a state $(q_{ext}, b, \mathfrak{a}, \ell, \varepsilon)$ such that it follows from Lemma 5.2.1 that $\mathfrak{A}_\ell$ realizes the language of trees in which the ℓ-th external vertex of some labeled vertex in the tree lies on a vertex replacement path with a real vertex u. By construction, this vertex u is required to be non-free. We make two further observations on the construction of $\mathfrak{A}_\ell$ which are easy to see and therefore not proven formally.

1. $\mathfrak{A}$ accepts in a final state with a component $b = 0$ if and only if no vertex on the corresponding vertex replacement path has outgoing edges.

2. In a vertex replacement path corresponding to an accepting run, exactly the attached and external vertices belonging to labeled vertices which are visited in a state with component q_{in} are non-free.

It remains to provide MSO_2F formulae to assert that external vertices are on proper vertex replacement paths. Recall the formula $isAttached_{\mathfrak{a},i,j}(\overline{D}, \overline{N}, r, x)$ defined in Section 5.2.2 and consider the analogous formula for external vertices

$$isExternal_{\mathfrak{a},i}(\overline{D}, \overline{N}, r, x) \triangleq \begin{cases} r \in N_\mathfrak{a} \land path_{W_{\overline{ext_\mathfrak{a}}[i]}}(\overline{D}, r, x) & \text{if } \overline{ext_\mathfrak{a}}[i] \notin free(\mathfrak{a}) \\ false & \text{otherwise.} \end{cases}$$

By Büchi's theorem, we know that there exists an MSO_2F formula characterizing the tree language realized by $\mathfrak{A}_\ell$ (see [Don70]). Let $\Phi_{\mathfrak{A}_\ell}(\overline{D}, \overline{N}, T, \overline{Q})$ be this MSO_2F formula where $\overline{Q}$ denotes a sequence of set variables representing the states of $\mathfrak{A}_\ell$. These set variables $\overline{Q}$ are used in Büchi's construction to encode an accepting run. Note that we marked all non-free vertices on a vertex replacement path by going into states with q_{in} as first component. This construction can be exploited to find these vertices in MSO_2F and to assert that all non-free vertices on a vertex replacement path are equal. Since variables representing free vertices are never introduced in any MSO_2F formula, we do not have to consider free vertices on a vertex replacement path. The following formula asserts that a variable u representing an external vertex introduced by a production rule $\mathfrak{a} \in \mathfrak{P}$ replaces an attached vertex represented by a variable a, i.e. u and a are equal. Furthermore, we use the parameter b to specify whether outgoing edges exist on the vertex replacement path containing u and a.

$$\begin{aligned}
boundExternal_{\mathfrak{a},b,\ell}(\overline{D},\overline{N},T,u,a) \triangleq{} & \exists \overline{Q}\, \exists^{\mathrm{v}}\, r_u\, \exists^{\mathrm{v}}\, r_a [\\
& \Phi_{\mathfrak{A}_\ell}(\overline{D},\overline{N},T,\overline{Q}) \\
& \wedge isExternal_{\mathfrak{a},\ell}(r_u,u) \wedge r_u \in Q_{\mathfrak{a},\ell} \\
& \wedge \bigvee_{(q_{att},b,\mathfrak{a},i,k)\in Q_{\mathfrak{A}_\ell}} (r_a \in Q_{(q_{att},b,\mathfrak{a},i,k)} \wedge isAttached_{\mathfrak{a},k,i}(r_a,a)) \\
& \wedge a = u \\
&]
\end{aligned}$$

For convenience, we define

$$boundExternal_{\mathfrak{a},\ell}(\overline{D},\overline{N},T,u,a) = \bigvee_{b\in\{0,1\}} boundExternal_{\mathfrak{a},b,\ell}(\overline{D},\overline{N},T,u,a).$$

In order to capture all external vertices, we also have to take vertex replacement paths with a free vertex as real vertex into account. This is achieved by a second tree-walking automaton $\mathfrak{A}_{\ell,\jmath}$ which checks whether a ℓ-th external vertex lies on a vertex replacement path where the real vertex is some $\jmath$-th free non-external vertex. Intuitively, $\mathfrak{A}_{\ell,\jmath}$ works analogously to the automaton $\mathfrak{A}_\ell$, but has a simpler acceptance condition, because it suffices to check that the first non-external vertex reached on a vertex replacement path is the $\jmath$-th free vertex.

Definition 5.2.8 (TWA for free attached vertices)**.** Let $\mathfrak{R}$ be the maximal rank of all nonterminals of a tree-like HRG $\mathfrak{G}$, $\ell \leq \mathfrak{R}$ and $\jmath \in \mathbb{N}$. The tree-walking automaton $\mathfrak{A}_{\ell,\jmath} = (Q, \mathfrak{P}\,\dot{\cup}\{\bot\}, \Delta, F)$ is a variant of the TWA $\mathfrak{A}_\ell$ (see Definition 5.2.7) where the set of final states is replaced by $F = \{(q_{final},0),(q_{final},1)\}$ and the transitions

$$\begin{aligned}
&((q_{ext},b,\mathfrak{a},i,\overline{w}),type,\mathfrak{b}),(q_{att},d(v,b),\mathfrak{b},i,k) \text{ for} \\
&\overline{w}\in W^{\leq \Bbbk}, nt_{\mathfrak{b}}(type\cdot\overline{w}) = k, v = att_{\mathfrak{b}}(\overline{\xi_{\mathfrak{b}}}[k])[i] \text{ and } v \notin \overline{[ext_{\mathfrak{b}}]} \cup free(\mathfrak{b}) \text{ or } type = root
\end{aligned}$$

are replaced by transitions

$$\begin{aligned}
&((q_{ext},b,\mathfrak{a},i,\overline{w}),type,\mathfrak{b},(q_{final},d(v,b)),0) \text{ for} \\
&\overline{w}\in W^{\leq \Bbbk}, nt_{\mathfrak{b}}(type\cdot\overline{w}) = k, v = att_{\mathfrak{b}}(\overline{\xi_{\mathfrak{b}}}[k])[i] = \overline{free(\mathfrak{b})}[\jmath] \text{ and } type = root \text{ or } v \notin \overline{[ext_{\mathfrak{b}}]}\,.
\end{aligned}$$

■

The correctness proof is analogous to the proof of Lemma 5.2.1 for the TWA $\mathfrak{A}_\ell$. Let $\Phi_{\ell,\jmath}(\overline{D},\overline{N},T,\overline{Q})$ be the MSO_2F formula corresponding to the TWA $\mathfrak{A}_{\ell,\jmath}$. Again, we define two auxiliary formulae to assert that the variable u corresponds to the ℓ-th external vertex of a production rule $\mathfrak{a}$ in some derivation step and replaces the $\jmath$-th free vertex introduced in some derivation step corresponding to the root vertex represented by a variable r_a. In addition to that, we check whether outgoing edges exist on the corresponding vertex replacement path, too.

$$\begin{aligned} freeExternal_{\mathfrak{a},b,\ell,\jmath}(T,\overline{D},\overline{N},u,r_a) \triangleq \exists\overline{Q}\,\exists^{\mathrm{v}}\, r_u[\\ &\Phi_{\mathfrak{A}_{\ell,\jmath}}(\overline{D},\overline{N},T,\overline{Q}) \\ &\wedge isExternal_{\mathfrak{a},\ell}(r_u,u)\wedge r_u\in Q_{\mathfrak{a},\ell} \\ &\wedge r_a\in Q_{q_{final},b} \\ &] \end{aligned}$$

$$freeExternal_{\mathfrak{a},\ell,\jmath}(T,\overline{D},\overline{N},u,r_a) \triangleq \bigvee_{b\in\{0,1\}} freeExternal_{\mathfrak{a},b,\ell,\jmath}(T,\overline{D},\overline{N},u,r_a)$$

In order to ensure the correct treatment of external vertices, we use these auxiliary formulae to express four essential properties. To keep the formulae small, let $\mathfrak{R}$ denote the maximal rank of all production rules as in Definition 5.2.6 and $\sharp(\mathfrak{a}) =\mid \boxed{ext_{\mathfrak{a}}} \setminus free(\mathfrak{a})|$. The first formula asserts that all external vertices lie on at least one vertex replacement path. Note that free external vertices are not considered, because they are never referenced by a variable.

$$\begin{aligned} externals(\overline{D},\overline{N},T,r) \triangleq \bigwedge_{\mathfrak{a}\in\mathfrak{P}} \forall^{\mathrm{v}}\, x\,\forall^{\mathrm{v}}\, r_x[\\ &r\neq r_x \wedge \bigvee_{\ell\leq\sharp\mathfrak{a}} isExternal_{\mathfrak{a},\ell}(r_x,x) \\ &]\rightarrow[\exists z. \\ &boundExternal_{\mathfrak{a},\ell}(\overline{D},\overline{N},T,x,z) \\ &\vee \bigvee_{\jmath\leq\mathfrak{R}} freeExternal_{\mathfrak{a},\ell,\jmath}(\overline{D},\overline{N},T,x,z) \\ &] \end{aligned}$$

The second property is based on the fact that external vertices in HRGs only refer to a vertex which already has been introduced in some previous derivation step. This, together with the fact that HRG derivations are monotone, which means the size of hypergraphs in a derivation never decreases for well-formed HRGs (see [Hab92]), implies that two external vertices with non-disjunct vertex replacement paths correspond to the same vertex in the final hypergraph of the derivation. Note that an external and attached vertex v may belong to several vertex replacement paths, because it may be attached to multiple nonterminal hyperedges. However, considering the last vertex of such a path yields a unique path.

$$\begin{aligned} &\mathit{pathEquality}(\overline{D}, \overline{N}, T) \triangleq \bigwedge_{\mathfrak{a}_1,\mathfrak{a}_2,\ell_1 \leq \sharp(\mathfrak{a}_1),\ell_2 \leq \sharp(\mathfrak{a}_2)} \forall^{\mathrm{v}}\, x\, \forall^{\mathrm{v}}\, y [\exists^{\mathrm{v}}\, z. \\ &\quad (\mathit{boundExternal}_{\mathfrak{a}_1,\ell_1}(\overline{D}, \overline{N}, T, x, z) \wedge \mathit{boundExternal}_{\mathfrak{a}_2,\ell_2}(\overline{D}, \overline{N}, T, y, z) \\ &\quad \vee \bigvee_{\jmath \leq \Re} (\mathit{freeExternal}_{\mathfrak{a}_1,\ell_1,\jmath}(\overline{D}, \overline{N}, T, x, z) \wedge \mathit{freeExternal}_{\mathfrak{a}_2,\ell_2,\jmath}(\overline{D}, \overline{N}, T, y, z) \\ &\quad] \rightarrow x = y \end{aligned}$$

By the same reasoning, we assert that every vertex corresponds to some vertex introduced as a non-external vertex. Let $\mathit{isFixed}_{\mathfrak{a}}(\overline{D}, \overline{N}, r, x) \triangleq r \in N_{\mathfrak{a}} \wedge \mathit{path}_{W_{\mathit{fixed}_{\mathfrak{a}}}}(\overline{D}, r, x)$ be an auxiliary formula to check whether a vertex represented by x belongs a non-external non-attached vertex introduced by a production rule $\mathfrak{a} \in \mathfrak{P}$ with root r. Then, every vertex is either such a non-external non-attached vertex or lies on a vertex replacement path and is therefore merged with a non-external vertex. This is formalized as follows.

$$\begin{aligned} &\mathit{allVertices}(\overline{D}, \overline{N}, T, r) \triangleq \forall^{\mathrm{v}}\, x\, \exists^{\mathrm{v}}\, y. \bigvee_{\mathfrak{a} \in \mathfrak{P}} [\\ &\quad \mathit{isFixed}_{\mathfrak{a}}(\overline{D}, \overline{N}, y, x) \\ &\quad \vee \bigvee_{\ell \leq \sharp \mathfrak{a}, \mathit{free}(\mathfrak{a})[\jmath] \notin [\mathit{ext}_{\mathfrak{a}}]} \mathit{freeExternal}_{\mathfrak{a},\ell,\jmath}(\overline{D}, \overline{N}, T, x, y) \\ &\quad \vee (y = r \wedge \bigvee_{v_i \in [\mathit{ext}_{\mathfrak{a}}] \setminus \mathit{free}(\mathfrak{a})} \mathit{isExternal}_{\mathfrak{a},i}(\overline{D}, \overline{N}, y, x)) \\ &\quad] \end{aligned}$$

Note that the last line treats the first applied production rule differently, because we do not explicitly represent the initial handle in our formula.

It remains to ensure that all edges attached to a vertex have been introduced by some production rule. For internal vertices, this has already been formulated in Section 5.2.1. For all other vertices, we require that every edge labeled with a tuple $(d, \mathfrak{a}, v_i, v_j)$ (see Section 5.2.1) must be defined by the corresponding production rule $\mathfrak{a}$ to connect the vertices v_i and v_j inside of $\mathfrak{a}$. Formally, this means that v_i must be reachable from the root of $\mathfrak{a}$ by a word $\overline{w} \in W_{\mathfrak{a}[i]}$ inside of $\mathfrak{a}$ and that extending this word $\overline{w}$ by the label $(d, \mathfrak{a}, v_i, v_j)$ also yields to a valid path inside of $\mathfrak{a}$ that leads to v_j, i.e. $\overline{w} \cdot (d, \mathfrak{a}, v_i, v_j) \in W_{\mathfrak{a}[j]}$. This is expressed by the following formula.

$$\begin{aligned} &\mathit{allOutgoingEdges}(\overline{D}, \overline{N}) \triangleq \bigwedge_{D_{d,\mathfrak{a},v_i,v_j} \in [\overline{D}], d > 0} \forall^{\mathrm{e}}\, e. e \in D_{d,\mathfrak{a},v_i,v_j} \rightarrow \exists^{\mathrm{v}}\, r\, \exists^{\mathrm{v}}\, u\, \exists^{\mathrm{v}}\, v [\\ &\quad r \in N_{\mathfrak{a}} \wedge \mathit{source}(u, e) \wedge \mathit{target}(e, v) \\ &\quad \wedge \mathit{path}_{W_{\mathfrak{a}[i]}}(\overline{D}, r, u) \end{aligned}$$

$$\wedge \bigvee_{\overline{w}\cdot(d,\mathfrak{a},v_i,v_j)\in W_{\mathfrak{a}[j]},\overline{w}\in W_{\mathfrak{a}[i]}} path_{\overline{w}\cdot(d,\mathfrak{a},v_i,v_j)}(\overline{D},r,v)$$
$$]$$

Furthermore, the same production rule may be applied multiple time to the same external vertex if it occurs as the same external vertex attached to the same nonterminal again. In this case, a vertex can be attached to multiple edges with the same labels in $\overline{D}$. In order to ensure that the number of such edges is correct, we introduce another formula which requires that different edges with the same labeling belong to different root vertices.

$$\begin{aligned}
multipleApplications(\overline{D},\overline{N}) \triangleq & \bigwedge_{D_{d,\mathfrak{a},v_i,v_j}\in[\overline{D}],v_i=\overline{ext_{\mathfrak{a}}}[k],d>0} \forall^{\mathrm{e}}\, e\, \forall^{\mathrm{e}}\, e'\, \forall^{\mathrm{v}}\, u[\\
& e \neq e' \wedge source(u,e) \wedge source(u,e') \wedge e \in D_{d,\mathfrak{a},v_i,v_j} \wedge e' \in D_{d,\mathfrak{a},v_i,v_j}\\
&] \rightarrow \exists^{\mathrm{v}}\, r\, \exists^{\mathrm{v}}\, r'[\\
& r \neq r' \wedge isExternal_{\mathfrak{a},k}(\overline{D},\overline{N},r,u) \wedge isExternal_{\mathfrak{a},k}(\overline{D},\overline{N},r',u))\\
&]
\end{aligned}$$

Finally, the corner case that all vertex replacement paths containing a vertex v have no outgoing edges needs special treatment. Otherwise, there could be more terminal hyperedges than in a properly derived hypergraph.

$$\begin{aligned}
unallocatedAttachments(\overline{D},\bar{N},T) \triangleq & \forall^{\mathrm{v}}\, x\, \forall^{\mathrm{v}}\, y\langle\\
& [\\
& \bigvee_{\mathfrak{a}\in\mathfrak{P},\ell\leq\sharp(\mathfrak{a})} boundExternal_{\mathfrak{a},\ell}(\bar{D},\bar{N},T,x,y)\\
& \wedge \bigwedge_{\mathfrak{a}\in\mathfrak{P},\ell\leq\sharp(\mathfrak{a})} \neg\exists z.boundExternal_{1,\mathfrak{a},\ell}(\bar{D},\bar{N},T,z,y)\\
&] \vee [\\
& \bigvee_{\mathfrak{a}\in\mathfrak{P},\ell\leq\sharp(\mathfrak{a}),\jmath\leq\mathfrak{R}} freeExternal_{\mathfrak{a},\ell,\jmath}(\bar{D},\bar{N},T,x,y)\\
& \wedge \bigwedge_{\mathfrak{a}\in\mathfrak{P},\ell\leq\sharp(\mathfrak{a}),\jmath\leq\mathfrak{R}} \neg\exists z.freeExternal_{1,\mathfrak{a},\ell,\jmath}(\bar{D},\bar{N},T,z,y)\\
&]\\
& \rangle \rightarrow \neg\, \exists^{\mathrm{e}}\, z.x \circ\!\!\rightarrow z
\end{aligned}$$

We have constructed MSO_2F formulae to verify for a given path-connected derivation tree and the hypergraph resulting from the derivation that external and attached vertices are merged correctly. Furthermore, it is ensured that every variable representing a vertex is introduced because of a production rule and every vertex corresponds to a non-external vertex in the derivation.

5.2.4. MSO_2L Definability of Tree-Like HRGs

The previous sections introduced MSO_2F formulae to identify path-connected derivation trees, to navigate through graphs using finite words over an alphabet of directions and to verify that a graph is the result of proper hyperedge replacement. To complete the proof of Theorem 5.2.1, it remains to provide the overall MSO_2F sentence $\Psi^{\xi}_{\mathfrak{G}}$ and show that every hypergraph is a model of $\Psi^{\xi}_{\mathfrak{G}}$ if and only if it can be generated by the HRG $\mathfrak{G} = (N, T, \mathfrak{P})$ with initial handle $\xi^{\bullet}$, i.e. $\mathcal{H} \in L(\mathfrak{G}, \xi) \Leftrightarrow \underline{\mathcal{H}} \models \Psi^{\xi}_{\mathfrak{G}}$ holds for every hypergraph $\mathcal{H} \in HG_T$. The MSO_2F sentence $\Psi^{\xi}_{\mathfrak{G}}$ is essentially the conjunction of all formulae presented in the previous three sections where the $graph_{\mathfrak{a}}$ formula is applied to every vertex labeled with a production rule $\mathfrak{a} \in \mathfrak{P}$.

$$\Psi^{\xi}_{\mathfrak{G}} \triangleq \exists \overline{D} \exists \overline{N} \exists T \exists r.[\tag{IV.1}$$

(edges are ordered by $\overline{D}$ to navigate inside of production rules)

$$edgeOrder(\overline{D}) \tag{IV.2}$$

(characterization of root vertices)

$$\wedge roots_{\xi}(\overline{D}, \overline{N}, r) \tag{IV.3}$$

(every root vertex is labeled with one production rule)

$$\wedge uniqueRoots(\overline{N}) \tag{IV.4}$$

(T is a path-connected derivation tree with root r and labeling $\overline{N}$)

$$\wedge derivationTree(\overline{D}, \overline{N}, T, r) \tag{IV.5}$$

(for every root vertex, the corresponding production rule is applied)

$$\wedge \bigwedge_{\mathfrak{a} \in \mathfrak{P}} \forall^{\mathrm{v}}\, x.x \in N_{\mathfrak{a}} \rightarrow (x \in T \wedge graph_{\mathfrak{a}}(\overline{D}, r)) \tag{IV.6}$$

(every external vertex lies on a vertex replacement path)

$$\wedge externals(\overline{D}, \overline{N}, T, r) \tag{IV.7}$$

(vertices with intersecting vertex replacement paths are equal)

$$\wedge pathEquality(\overline{D}, \overline{N}, T) \tag{IV.8}$$

(all vertices are equal to a non-external vertex)

$$\wedge allVertices(\overline{D}, \overline{N}, T, r) \tag{IV.9}$$

(all edges are defined by production rules)

$$\wedge allOutgoingEdges(\overline{D}, \overline{N}) \tag{IV.10}$$

(equal edges are defined by different applications of a production rule)

$$\wedge multipleApplications(\overline{D}, \overline{N}) \tag{IV.11}$$

(treatment of vertex replacement paths without outgoing edges)

$$\wedge unallocatedAttachments(\overline{D}, \overline{N}, T) \tag{IV.12}$$

$$] \tag{IV.13}$$

We start with the “only if” direction of the proof, i.e. we show that every model of $\Psi^{\xi}_{\mathfrak{G}}$ is indeed a hypergraph that can be generated by $\mathfrak{G}$ with initial handle $\xi^{\bullet}$, i.e.

$\mathcal{H} \in L(\mathfrak{G}, \xi) \Leftarrow \underline{\mathcal{H}} \models \Psi^{\xi}_{\mathfrak{G}}$

Assume $\underline{\mathcal{H}} \models \Psi^{\xi}_{\mathfrak{G}}$. By IV.1, there must be variables $T, \bar{N}, \bar{D}$ and r satisfying all subformulae of $\Psi^{\xi}_{\mathfrak{G}}$. Furthermore, IV.2 implies that all hyperedges of $\mathcal{H}$ have rank two and are ordered with one positive and one negative direction. Thus, for each vertex v there is a sequence $\overline{e}(v) = e_1...e_n$ such that e_i is labeled with some direction and $e_i \in [\overline{vOut}]$ or $e_i \in [\overline{vIn}]$.

The main steps of the proof can be summarized as follows. At first, we show that $T, \bar{N}$ and r describe a path-connected derivation tree $(T, \ell_{\overline{N}})$ with root r, i.e. $\underline{\mathcal{H}} \models \Psi^{\xi}_{\mathfrak{G}}$ implies $(T, \ell_{\overline{N}}) \in \mathfrak{T}_{\mathfrak{G}}(\xi)$. Since there is a unique derivation tree corresponding to a rooted path-connected derivation tree, there is a unique hypergraph $\mathcal{K}$ corresponding to $(T, \ell_{\overline{N}})$. To complete the proof, we show that $\mathcal{K} \cong \mathcal{H}$ (see Section 3.1). This obviously implies $\mathcal{H} \in L(\mathfrak{G}, \xi)$.

$(T, \ell_{\overline{N}})$ **is a path-connected derivation tree with root** r Clearly, IV.4 asserts that no vertex is labeled with more than one production rule, i.e. $N_{\mathfrak{a}} \cap N_{\mathfrak{b}} = \emptyset$. By IV.5, T is a tree rooted at r. In addition to that, every vertex labeled with some production rule $\mathfrak{a} \in \mathfrak{P}$ is in T due to IV.3 and IV.5. Note that these two formulae also imply that the leaves of T are labeled with production rules. We show by complete induction over the maximal height h of labeled inner vertices of T that $\underline{\mathcal{H}} \models \Psi^{\xi}_{\mathfrak{G}}$ implies $(T, \ell_{\overline{N}}) \in \mathfrak{T}_{\mathfrak{G}}(\xi)$.

Induction Base $h = 0$: By assumption, only the root r of T is labeled with a production rule and by IV.3, r is labeled with some production rule $\mathfrak{x} = \xi \to \mathcal{X}$ in $\mathfrak{P}$. Furthermore, IV.5 (more precisely II.6) asserts that every vertex in T is either labeled with a production rule or on a path in T to such a vertex. Thus, since only r is labeled, T consists exactly of the singular vertex r and the production rule $\mathfrak{x} = \xi \to \mathcal{X}$ is terminal. By Definition 5.2.4, $(T, \ell_{\bar{N}})$ is a path-connected derivation tree.

Induction Step $h \mapsto h + 1$: Again, r must be labeled with a production rule $\mathfrak{x} = \xi \to \mathcal{X}$ in $\mathfrak{P}$. Then, IV.6 ensures the existence of all (non-free) vertices and hyperedges of the graph defined by the production rule $\mathfrak{x}$. By assumption there exists at least one other vertex labeled with a production rule. Furthermore, IV.6 and IV.2 enforce the edge ordering $\overline{D}$ to correspond to the sequence $\overline{e}(v)$ for each vertex v in the hypergraph $\mathcal{H}(root_{\mathfrak{a}})$. Thus, any path following the directions defined by $\overline{D}$ from $r = root_{\mathfrak{x}}$ to a non-external vertex in $\mathfrak{x}$ also describes a unique path in $\mathcal{H}$. Now, IV.3 implies that all context vertices $a_1, ..., a_k$ of $\mathfrak{x}$ are labeled with production rules and together with IV.5, it follows that $\mathfrak{x}$ is production rule containing no nonterminal hyperedges. In addition to that, IV.5 ensures that a path from r to a context vertex a_i is in T and every vertex in T is labeled or on a path to a vertex labeled with a production rule. Let $(T', \ell'_{\overline{N}})$ be the restriction of T and $\overline{N}$ to these paths and $\xi_1, ..., \xi_k$ be the nonterminal symbols corresponding to the context vertices $a_1, ..., a_k$. Then, for each $i \in [1, k]$, there exists a subgraph $\mathcal{H}_i$ of $\mathcal{H}$ such that $\underline{\mathcal{H}_i}$ is a model of $\Psi^{\xi_i}_{\mathfrak{G}}$ with $r = a_i$. Intuitively, $\mathcal{H}_i$ is obtained by selecting the nonterminal symbol ξ_i as initial nonterminal instead of ξ and choosing the same production rule for each nonterminal hyperedge as in a derivation of $\mathcal{H}$. Since the (labeled) root vertex of $\mathcal{H}$ is not in $\mathcal{H}_i$, the height of labeled inner nodes is h. By induction hypothesis, this implies the existence of path-connected derivation trees $(T_i, \ell_i) \in \mathfrak{T}_{\mathfrak{G}}(\xi_i)$ for each $i \in [1, k]$. Hence, by Definition 5.2.4, $T = T'[T_1/a_1, ..., T_k/a_k]$ together with the respective labeling describes a path-connected derivation tree in $\mathfrak{T}_{\mathfrak{G}}(\xi)$.

Graph Equality For the rest of this section, let $\mathcal{K}$ be the hypergraph induced by the path-connected derivation tree $(T, \ell_{\overline{N}})$. In order to show that $\mathcal{K} \cong \mathcal{H}$ holds, we first show that both hypergraphs have the same number of vertices. Clearly, all vertices of T are in $\mathcal{K}$ as well as in $\mathcal{H}$.

$|V_{\mathcal{H}}| \leq |V_{\mathcal{K}}|$: Assume there exists a vertex $v \in V_{\mathcal{H}} \setminus V_{\mathcal{K}}$. By IV.9, three cases are possible.

1. There exists a vertex y labeled with $\mathfrak{a} \in \mathfrak{P}$ such that $isFixed_{\mathfrak{a}}(y, x)$ is satisfied. Since y is labeled, it is in T (by IV.5 and IV.3) and x is a non-external, non-free vertex in the graph described by $\mathfrak{a}$. Then, x must also be in $V_{\mathcal{K}}$, because HRGs are monotone and at some point in the derivation of $\mathcal{K}$ all non-external, non-free vertices described by $\mathfrak{a}$ are added.

2. There exists a vertex y labeled with a production rule $\mathfrak{a} \in \mathfrak{P}$ and integers $\ell, \jmath$ such that $freeExternal_{\mathfrak{a},\ell,\jmath}(T, \bar{D}, \bar{N}, x, y)$ is satisfied. Due to Lemma 5.2.1, this means that there is a vertex replacement path containing x and the $\jmath$-th free non-external vertex introduced by $\mathfrak{a}$ with $y = root_{\mathfrak{a}}$. Obviously, this vertex also exists in $\mathcal{K}$. Furthermore, IV.8 asserts that all vertices with such a vertex replacement path are equal, i.e. there is exactly one vertex corresponding to this free vertex in $\mathcal{H}$.

3. y is an external, non-free vertex in $\mathcal{H}(\mathfrak{a})$ with the root r of T as $root_{\mathfrak{a}}$. Since all derivations of hypergraphs in $L(\mathfrak{G}, \xi)$ start with the initial handle $\xi^{\bullet}$ where all vertices are free and non-external, it is clear that such a vertex also exists in $\mathcal{K}$.

Since all three cases lead to a contradiction, it follows that $|V_{\mathcal{H}}| \leq |V_{\mathcal{K}}|$.

$|V_{\mathcal{H}}| \geq |V_{\mathcal{K}}|$: Since HRGs are monotone, a non-external vertex added in some derivation step is never removed. Furthermore, an external vertex always refers to a non-external vertex added in some derivation step before. For the initial derivation step, external vertices correspond to free vertices of the initial handle. Thus, $V_{\mathcal{K}}$ is the union of the following sets of vertices:

1. All vertices introduced in some derivation step which are neither free nor external nor attached,

2. all vertices introduced in some derivation step which are free and not external in some derivation step, and

3. all vertices introduced in some derivation step which are attached but neither free nor external in some derivation step.

In the first case, it is easy to see that IV.6 ensures that all of these vertices are also in $V_{\mathcal{H}}$, because their existence is ensured and they may have no other incoming edges, i.e. they cannot be equal to some other vertex introduced in another production rule.

Now, let VA denote the set of all vertices in the second and the third set and let $\Bbbk$ be the number of vertices that have been introduced as external vertices which are not attached to any nonterminal hyperedge in some derivation step. Since every vertex replacement path ends in an external, non-attached vertex and there is a unique vertex replacement path for each of these vertices, $\Bbbk$ coincides with the number of vertex replacement paths in the derivation of $\mathcal{K}$. Furthermore, every vertex $v \in VA$ lies on as many vertex replacement paths as it is attached to nonterminal hyperedges during the derivation of $\mathcal{K}$. Let k_v denote this number of vertex replacement paths for each vertex $v \in VA$. Obviously, it follows that $\Bbbk = \sum_{k \in VA} k_v$. Since $\mathcal{H}$ contains the path-connected derivation tree (T, l), it follows from IV.6 that there are (at least) $\Bbbk$ variables referring to external, non-attached vertices in $\Psi^{\xi}_{\mathfrak{G}}$. By IV.7 and Lemma 5.2.1, there exists a vertex replacement path for each of these variables and by IV.8, the existence of two vertex replacement paths leading to the same non-attached vertex already implies that both are equal. Since there are (at most) k_v different vertex replacement paths starting in a vertex v and leading to a non-attached external vertex, $\mathcal{H}$ must contain at least all attached and non-external vertices in $\mathcal{K}$ which covers the second and the third set.

To complete the proof, it remains to show that $\mathcal{H}$ and $\mathcal{K}$ are really isomorphic. For the labeling this is clear by IV.6 (see I.9).

$\mathcal{K}$ is isomorphic to a subgraph of $\mathcal{H}$: Towards a contradiction, assume $\mathcal{K}$ is not isomorphic to a subgraph of $\mathcal{H}$. Since $|V_{\mathcal{H}}|=|V_{\mathcal{K}}|$, there must be an edge e with source x and target y in $\mathcal{K}$ which is not in $\mathcal{H}$. Then, in the derivation of $\mathcal{K}$, there is a nonterminal hyperedge and some production rule $\mathfrak{a} \in \mathfrak{P}$ such that e is added to $\mathcal{K}$. Since $\mathfrak{G}$ is tree-like, there is a vertex z labeled $\mathfrak{a}$ in the path-connected derivation tree T corresponding to the context vertex of this hyperedge. Obviously, x and y are not free in $\mathcal{H}_{\mathfrak{a}}$, i.e. there exist words $\overline{w_x}$ and $\overline{w_y}$ leading from z to x and y, respectively. We already know that T is also in $\mathcal{H}$, i.e. there is a vertex corresponding to z in $\mathcal{H}$. Since this vertex is labeled $\mathfrak{a}$, it must satisfy IV.6 for this production rule. Thus, since IV.2 is also satisfied, the paths starting in z and following words $\overline{w_x}$ and $\overline{w_y}$ also lead to vertices x and y in $\mathcal{H}$, respectively. Furthermore, x and y must correspond to variables v_i and v_j for some i and j in IV.6 by IV.10 and IV.11. However, this implies that all edges between vertices in the graph defined by $\mathfrak{a}$ are also in $\mathcal{H}$ which contradicts the initial assumption.

$\mathcal{H}$ is isomorphic to a subgraph of $\mathcal{K}$: Towards a contradiction, assume $\mathcal{H}$ is not isomorphic to a subgraph of $\mathcal{K}$. Since $|V_{\mathcal{H}}|=|V_{\mathcal{K}}|$, there must be an edge e with source x and target y in $\mathcal{H}$ which is not in $\mathcal{K}$. By IV.9, there exists a vertex r_x labeled with a production rule $\mathfrak{a} = \alpha \to \mathcal{A}$ such that only three cases are possible.

1. $isFixed_{\mathfrak{a}}(\overline{D}, \overline{N}, y, x)$ holds. By definition, this means that x corresponds to some variable v_i in IV.6 representing a vertex in $\mathfrak{a}$ which is neither free nor external. If v_i is also not attached, it is clear by IV.6 that e must be an edge in $\mathcal{K}$. Otherwise, IV.7 and Lemma 5.2.1 ensure that x belongs to some vertex replacement path. By IV.8, x is equal to all vertices on such a path. Since x has at least one outgoing edge, the automaton checking the vertex replacement path does not terminate with a component $b = 0$. Otherwise IV.12 would ensure that x has no outgoing edges at all. Hence, there exists some vertex z on the vertex replacement path with at least one outgoing edge e. By the same argumentation as for x, there must be a vertex r_z labeled $\mathfrak{b} = \beta \to \mathcal{B}$ and a word $\overline{w_z}$ such that Then, by IV.10, every such edge e corresponds to some edge defined in IV.6, i.e. the edge is introduced by a production rule $\mathfrak{b} \in \mathfrak{P}$ and its application is defined by the root vertex r_z. For the corner case that multiple edges e have the same labeling in $\overline{D}$, IV.11 ensures that each of these edges is introduced by another application of the same production rule. Since r_z is a labeled vertex in the derivation tree, the same derivation step is part of the derivation of $\mathcal{K}$, i.e. e is an edge in $\mathcal{K}$.

2. x is the ℓ-th external vertex of some labeled vertex r_z in T and there is a vertex replacement path containing the $\jmath$-th free vertex belonging to r_x. Then, again x belongs to some vertex replacement path and by the same argument as in the first case, e is an edge in $\mathcal{K}$.

3. x corresponds to an external, non-free vertex in the graph defined by IV.6 for the root vertex of T. Since these vertices correspond to the vertices of the initial handle, this is analogous to the first case.

All three cases lead to a contradiction of the assumption that e is not in $\mathcal{K}$ which implies that the claim holds. Hence, all models of $\Psi^{\xi}_{\mathfrak{G}}$ are also in the language generated by $\mathfrak{G}$. It remains to prove the converse direction, i.e.

$\mathcal{H} \in L(\mathfrak{G}, \xi) \Rightarrow \underline{\mathcal{H}} \models \Psi^{\xi}_{\mathfrak{G}}$

Let $\mathcal{H} \in L(\mathfrak{G}, \xi)$ and (T, l) be a path-connected derivation tree corresponding to $\mathcal{H}$ with root r and $\overline{N}$ be the encoding of the labeling function l (see Chapter 2). Furthermore, let $\overline{D}$ be a sequence of sets such that for every hyperedge e in $\mathcal{H}$, which is introduced in some derivation step by a production rule $\mathfrak{a} = \alpha \to \mathcal{A}$, e is in $D_{d,\mathfrak{a},v_i,v_j}$ if and only if $src_{\mathfrak{a}}(e) = v_i$, $tgt_{\mathfrak{a}}(e) = v_j$ and either $e = \overline{v_i Out_{\mathfrak{a}}}[d]$ and $d > 0$ or $e = \overline{v_j In_{\mathfrak{a}}}[d]$ and $d < 0$. We show that all conjuncts IV.2 to IV.13 of $\Psi^{\xi}_{\mathfrak{G}}$ are satisfied for this choice of $\overline{D}, \overline{N}, T$ and r.

$edgeOrder(\overline{D})$ **(IV.2)** By Definition 5.2.1, every terminal hyperedge is of rank two. Furthermore, $width_{\mathfrak{G}}$ denotes the maximal number of ingoing and outgoing edges of any vertex defined in a production rule of $\mathfrak{G}$. Since every edge must be added by a derivation step, it is clear that all outgoing edges of any vertex in $\mathcal{H}$ can be ordered as done in $\overline{D}$.

The same holds for all ingoing edges of a vertex inside a single production rule. Since our choice of $\overline{D}$ never assigns two positive or two negative directions to a hyperedge, IV.2 is satisfied.

$roots_{\xi}(\overline{D}, \overline{N}, r)$ **(IV.3)** Obviously, the root r of a path-connected derivation tree (T, l) is labeled with some production rule corresponding to nonterminal ξ. Furthermore, by definition of path-connected derivation trees, exactly the context vertices in each graph $\mathcal{A}$ corresponding to a production rule $\mathfrak{a} = \alpha \rightarrow \mathcal{A}$ are labeled with production rules. Since $\mathfrak{G}$ is tree-like, there exists a path from $root_{\mathcal{A}}$ to every non-free vertex x such that no vertex on the path except $root_{\mathcal{A}}$ and x is external. Hence, $isRootAttached(\bar{D}, \bar{N}, x)$ is satisfied for our choice of $\bar{D}, \bar{N}$ and r exactly for the labeled vertices in T, i.e. for each context vertex. Thus, IV.3 is satisfied.

$uniqueRoots(\overline{N})$ **(IV.4)** This is clear by Definition 5.2.4 and Definition 5.2.1. Note that every vertex in a path-connected derivation tree can be labeled at most once, because root vertices are not allowed to be context vertices.

$derivationTree(\overline{D}, \overline{N}, T, r)$ **(IV.5)** Obviously, T is a tree rooted at r. By Definition 5.2.4, T is minimal and contains all context vertices. Thus, for each vertex x in T, either x is labeled or lies on a path in T to another labeled vertex and all paths in T starting in a labeled vertex lead to a context vertex. Hence, IV.5 is satisfied.

$graph_{\mathfrak{a}}(\overline{D}, r))$ **(IV.6)** Let x be a vertex in T labeled with a production rule $\mathfrak{a} \in \mathfrak{P}$. Since HRGs are monotone, all vertices $v_1, ..., v_n$ defined in IV.6 are different and exist in $\mathcal{H}$. Furthermore, edges between these vertices as well as the edge ordering correspond to the edges defined in $\mathfrak{a}$. Since it is impossible for an HRG to add an edge between vertices x and y where neither x nor y are attached or external and both vertices are added in different different derivation steps, IV.6 is satisfied.

$externals(\overline{D}, \overline{N}, T, r)$ **(IV.7)** Let x and $r_x \neq r$ be vertices satisfying the formula $isExternal_{\mathfrak{a},\ell}(r_x, x)$. By our choice of $\overline{D}$, this means that x is the ℓ-th external vertex of a graph $\mathfrak{a}$ corresponding to a derivation step with root r_x. Since an external vertex never introduces a new vertex in an HRG derivation, there must exist a vertex replacement path such that x corresponds to a previously introduced attached vertex z. Then, Lemma 5.2.1 implies that IV.7 is true.

$pathEquality(\overline{D}, \overline{N}, T)$ **(IV.8)** For HRG derivations it is clear that vertices on the same vertex replacement path are equal in the final graph $\mathcal{H}$ and that a non-empty intersection of two vertex replacement paths already implies that all vertices on both paths are equal. By Lemma 5.2.1, it follows that the precondition of IV.8 is satisfied only if two external vertices belong to the same attached non-external vertex.

$allVertices(\overline{D}, \overline{N}, T, r)$ **(IV.9)** Since $\mathcal{H}$ is the result of a derivation in $\mathfrak{G}$, every vertex x in $\mathcal{H}$ has been added in some derivation step as a non-external (free or non-free) vertex. Thus, either there exists a labeled vertex y in T such that x is a non-external, non-free vertex belonging to the graph induced by the label of y or x is equal to a free, non-external vertex introduced again by a labeled vertex y in T. Due to Lemma 5.2.1, all vertices not satisfying the first or third case are replacements of such free, non-external vertices. Hence, IV.9 holds.

$allOutgoingEdges(\overline{D}, \overline{N})$ **(IV.10)** By our choice of $\overline{D}$, every edge is labeled with a tuple $(d, \mathfrak{a}, v_i, v_j)$ if and only if it has been introduced by an application of the production rule $\mathfrak{a}$ to connect the vertices v_i and v_j inside of $\mathfrak{a}$. Then, a root vertex r of this application must exist in the derivation tree of $\mathcal{H}$ such that v_i is reachable by a path inside of $\mathfrak{a}$ from r. Since there is an edge between v_i and v_j in $\mathfrak{a}$, this path can be extended by $(d, \mathfrak{a}, v_i, v_j)$ to reach v_j from r via v_i which satisfies (IV.10).

$multipleApplications(\overline{D}, \overline{N})$ **(IV.11)** This formula is obviously satisfied of no vertex u is the source of two edges with exactly the same labeling. By our choice of $\overline{D}$, two edges can get the same labeling only if they have been introduced by two applications of the same production rule. Furthermore, u must be the same external vertex in both applications to be the source of two such edges. Since every root vertex corresponds to exactly one application of a production rule, there must be two root vertices representing the two applications of a production rule with u as external vertex. Then, (IV.11) is satisfied.

$unallocatedAttachments(\overline{D}, \overline{N}, T)$ **(IV.12)** Note that the left-hand side of the implication is satisfied only if x refers to an external vertex at some point in the derivation of $\mathcal{H}$ and y refers to an attached non-external vertex such that x and y are on the same vertex replacement path due to Lemma 5.2.1. Furthermore, if there is any vertex replacement path including y and containing a vertex with outgoing edges in the derivation of $\mathcal{H}$, the left-hand side of the implication is violated, too. Hence, the right-hand side of the implication is only relevant if no vertex in any vertex replacement path leading to y has outgoing edges. In this case, it is clear that x has no outgoing edges.

Since all subformulae are satisfied for our choice of $T, \bar{N}, \bar{D}$ and r, it follows that $\underline{\mathcal{H}} \models \Psi^{\xi}_{\mathfrak{G}}$ which concludes the proof of Theorem 5.2.1. $\square$

A discussion of the expressiveness of tree-like HRGs in comparison to general HR-languages and separation logic is given in Chapter 6.

5.3. Some Algorithmic Properties

In the previous sections, we provided a proof of Theorem 5.2.1 showing that every tree-like HRG can be translated into an equivalent MSO_2F formula. Since the satisfiability problem for MSO_2F formulae over graphs of bounded tree width is decidable, it follows that the emptiness problem is decidable. Furthermore, we can first translate two tree-like HRGs $\mathfrak{G}$ and $\mathfrak{G}'$ with initial nonterminals α and β into two equivalent MSO_2F formulae φ, φ' and check whether the formula $\varphi \rightarrow \varphi'$ is satisfiable in order to check the inclusion problem, i.e. whether $L(\mathfrak{G}, \alpha) \subseteq L(\mathfrak{G}', \beta)$ holds. Unfortunately, we already mentioned in Proposition 4.1.1 that any algorithm solving the satisfiability problem for MSO_2F formulae over strings or trees cannot be bounded by a function of the form $2^{2^{\cdots}}\}k-\text{times}$ for any $k \in \mathbb{N}$. The same result can be lifted to MSO_2F formulae over hypergraphs of bounded tree width. Thus, checking the emptiness or the language inclusion problem for (tree-like) HRGs by translating them into MSO_2 formulae first is not tractable for practical applications.

However, there are more efficient algorithms to solve the emptiness and inclusion problem for MSO_2F definable languages of strings or trees if a language is not given as an MSO_2F formula, but as a grammar or a finite automaton. This is a consequence of the equivalence between regular languages and MSO_2F definable languages of strings or trees (see Example 4.1.1 and Example 4.1.2). We collect the corresponding complexity results in the following proposition.

Proposition 5.3.1. *Let L be a regular language of strings or trees which is given as a finite (tree) automaton or an appropriate regular (tree) grammar. Then, the emptiness problem is solvable in polynomial time. Furthermore, the language inclusion problem is* PSPACE*-complete if L is a regular string language and* EXPTIME*-complete if L is a regular tree language (cf. [Sei90]).* ■

In this section, we study whether similar complexity results can be obtained for MSO_2F definable HR-languages that are given as a tree-like HRG. Note that this does not capture all MSO_2F definable HR-languages as it is the case for string and tree languages. We start with the emptiness problem for tree-like HRGs. Habel already notes in [Hab92] that the emptiness problem for HRGs is decidable. However, results on the complexity of the emptiness problem are sparse for general HRGs. For tree-like HRGs, we can provide a polynomial time algorithm to solve the emptiness problem. Our algorithm is based on the following observation: The language of a tree-like HRG $\mathfrak{G}$ with initial nonterminal ξ is empty if and only if the corresponding set of all derivation trees $\mathfrak{T}_{\mathfrak{G}}(\xi)$ is empty, i.e. $L(\mathfrak{G}, \xi) = \emptyset$ iff $\mathfrak{T}_{\mathfrak{G}}(\xi) = \emptyset$. Since tree-like HRGs are constructed in compliance with the intuition based on the results of Engelfriet and Courcelle (see Section 5.1), we can reconstruct a derivation tree from every hypergraph $\mathcal{H} \in L(\mathfrak{G}, \xi)$. This motivates the construction of an algorithm as sketched in Algorithm 1. Intuitively, this algorithm first constructs a regular tree grammar $\mathfrak{G}_T$ realizing undirected tree languages from a normalized tree-like HRG $\mathfrak{G}$. This grammar is then transformed into an equivalent regular tree automaton. By Proposition 5.3.1, the emptiness problem for regular tree automata can be solved in polynomial time. Furthermore, it is well-known that spanning trees with a given root vertex can be computed in polynomial time. Since $\mathfrak{G}$ is normalized and tree-like, such a spanning tree can be efficiently transformed into a component of a path-connected derivation tree by keeping only paths from the context

Algorithm 1 Checking emptiness for normalized tree-like HRGs

Input: normalized tree-like HRG $\mathfrak{G} = (N, T, \mathfrak{P})$, a nonterminal $\xi \in N$

Output: true if $L(\mathfrak{G}, \xi) = \emptyset$ and false otherwise

```
 1: function EMPTY($\mathfrak{G}, \xi$)
 2:     $\mathfrak{P}_T \leftarrow \emptyset$        ▷ production rules of regular tree grammar $(N, T, \mathfrak{P}_T)$
 3:     for $\mathfrak{a} = \alpha \rightarrow \mathcal{A} \in \mathfrak{P}$ do
 4:         $T = SpanningTree(\mathcal{A}, root_\mathfrak{a}) \cup \{e \in E_\mathfrak{a} \mid lab(e) \in N\}$
 5:         $T_\mathfrak{a} = T \setminus \{v \in V_\mathfrak{a} \mid \forall c \in \overline{context_\mathfrak{a}}.v \notin path_T(root_\mathfrak{a}, c)\}$
 6:         $\mathfrak{P}_T.add(\alpha \rightarrow T_\mathfrak{a})$
 7:     end for
 8:     $\mathfrak{A}_T = treeAutomaton(N, T, \mathfrak{P}_T)$        ▷ tree automaton equivalent to
 9:                                                               grammar $(N, T, \mathfrak{P}_T)$
10:     return $L(\mathfrak{A}_T) = \emptyset$?
11: end function
```

vertices to the root vertex. Thus, the grammar $\mathfrak{G}_T$ can be computed in polynomial time. Finally, regular tree grammars can be efficiently translated into regular tree automata (cf. [CDG+07]). Hence, we obtain the following theorem.

Theorem 5.3.1. *The emptiness problem of normalized tree-like HRGs can be solved in polynomial time.* ■

Note that this algorithm works for every class of HRGs for which an effective procedure to compute the set of derivation trees from the set of production rules exists. Furthermore, the theorem also applies to non-normalized tree-like HRGs, because a tree-like HRG can be normalized in polynomial time as described in Section 5.2.

Finally, we sketch an algorithm to check for an arbitrary HRG $\mathfrak{G} = (N, T, \mathfrak{P})$ whether it is tree-like. Note that there is only a finite number of external vertices in each right-hand side of a production rule and therefore there is a maximal number of external vertices for each nonterminal symbol α. Thus, there are only finitely many predicates of the form $roots : N \rightharpoonup [1, max\{rk(\alpha) \mid \alpha \in N\}]$ assigning an index of an external vertex to each nonterminal symbol. Intuitively, these indices mark the root vertex of each production rule with α as left-hand side. Now, given a hypergraph $\mathcal{H}$ and such a predicate *roots*, it is straightforward to check whether $\mathcal{H}$ is tree-like where *roots* defines the selection of the root and context vertices which correspond to root vertices of other nonterminals. This is done by Algorithm 2. If all production rules map to tree-like hypergraphs with respect to a predicate *roots*, we can extract a regular tree-grammar from these production rules and check whether this grammar can be normalized as described in Section 5.2. Regarding the complexity, note that all computations in Algorithm 2 can be performed in polynomial time. Thus, as an upper bound, we obtain that checking tree-likeness is in NP, because we still have to guess the right assignment of root and context vertices.

Algorithm 2 Checking whether a given hypergraph is tree-like

Input: Hypergraph $\mathcal{H}$, predicate $roots : N \rightharpoonup [1, max\{rk(\alpha) \mid \alpha \in N\}]$

Output: true if $\mathcal{H}$ with root and context defined by $roots$ is tree-like and false otherwise

```
 1: function TreeLike($\mathcal{H}$,$roots$)
 2:     $E_N = \{e \in E_\mathcal{H} \mid lab(e) \in N\}$
 3:     $V_N = \{v \in V_\mathcal{H} \mid \exists e \in E_N . v = att_\mathcal{H}(e)[roots(lab(e))]\}$
 4:     for $e \in E_\mathcal{H} \setminus E_N$ do
 5:         if $rk(e) \neq 2$ then
 6:             return false
 7:         end if
 8:     end for
 9:     $\mathcal{H}' = UndirectedGraph(\mathcal{H} \setminus (free(\mathcal{H}) \cup [\overline{ext_\mathcal{H}}] \cup V_N))$
10:     if $UndirectedGraph(\mathcal{H} \setminus free(\mathcal{H}))$ connected and $\mathcal{H}'$ connected then
11:         return $|V_N| = |E_N|$?
12:     else
13:         return false
14:     end if
15: end function
```

CHAPTER 6

A Landscape of Hypergraph Languages

The previous chapters introduced and discussed graphical and logical formalisms to realize various classes of hypergraph languages. In this chapter, these formalisms are compared with respect to their expressiveness and decidability of the language inclusion and entailment problem. In addition to formalisms presented in previous chapters, we additionally consider two separation logic fragments introduced by Iosif et al. [IRS13] [IRV14].

6.1. MSO_2L, $\mathcal{HRL}$ and $\texttt{SL}_{RD}$

We begin with some results from the literature regarding the most general formalisms considered in this thesis: monadic second-order logic over graphs, HR-languages and separation logic with inductive predicates without further restrictions. For each of these formalisms, the entailment problem (or language inclusion problem) is undecidable, as discussed in Section 4.1 for MSO_2L, in Example 3.2.1 for $\mathcal{HRL}$ and Section 4.3 for $\texttt{SL}_{RD}$ (see also [AGH$^+$14]), respectively. Regarding the expressiveness, the following lemma states that for each formalism a hypergraph language exists that cannot be realized by the other two formalisms.

Lemma 6.1.1. *MSO_2L, $\mathcal{HRL}$ and $\texttt{SL}_{RD}$ are incomparable.* ■

Proof (of Lemma 6.1.1). We begin with the relationship between MSO_2L and $\mathcal{HRL}$. As already shown in Example 3.2.1, every context-free string language can be encoded in $\mathcal{HRL}$, but not all context-free string languages are definable in MSO_2L. Intuitively, the context-free string language $L = \{a^n b^n \mid n \geq 0\}$ is not MSO_2L definable, because counting is impossible in MSO_2L. By the same argument, the language of even stars - graphs where a single vertex is connected to $2n + 1$ vertices for every $n \in \mathbb{N}$ - is not MSO_2L definable, but realizable by an HRG as shown in Figure 5.7.

For the converse direction, we note that the set of all Eulerian graphs, i.e. graphs where every vertex is attached to exactly two edges, is not an HR-language (cf. [Hab92]). However, the formula

$$\begin{aligned}
\varphi \triangleq\ & \forall^{\mathrm{v}}\, x\, \exists^{\mathrm{e}}\, y\, \exists^{\mathrm{e}}\, z[\\
& \quad y \neq z\\
& \wedge (x \circ\!\!\rightarrow y \vee x \rightarrow\!\!\circ\ y)\\
& \wedge (x \circ\!\!\rightarrow z \vee x \rightarrow\!\!\circ\ z)\\
& \wedge \forall z'(x \circ\!\!\rightarrow z' \vee x \rightarrow\!\!\circ\ z') \rightarrow (z' = y \vee z' = z)\\
& \quad]
\end{aligned}$$

characterizes exactly the set of all Eulerian graphs, i.e. MSO_2L is not included in $\mathcal{HRL}$.

We now turn to the relationship between $\mathtt{SL}_{RD}$ and the other two formalisms. By Proposition 4.3.1, a fragment of $\mathcal{HRL}$ that covers context-free string languages is included in $\mathtt{SL}_{RD}$. Thus, it directly follows that $\mathtt{SL}_{RD}$ is not included in MSO_2L. Furthermore, the MSO_2F formula φ from above can be written as a $\mathtt{SL}_{RD}$ formula using pointer assertions instead of the binary relations $\circ\!\!\rightarrow$ and $\rightarrow\!\!\circ$. Hence, the set of Eulerian graphs, which are also heaps, can be defined in $\mathtt{SL}_{RD}$ which implies $\mathtt{SL}_{RD} \not\subseteq \mathcal{HRL}$. Since $\mathtt{SL}_{RD}$ is defined for sets of heaps only (see Definition 4.2.1), neither MSO_2L nor $\mathcal{HRL}$ is included in $\mathtt{SL}_{RD}$. For example, the MSO_2 formula

$$\psi \triangleq \exists^{\mathrm{v}}\, x\, \exists^{\mathrm{v}}\, y\, \exists^{\mathrm{v}}\, z(x \neq y \wedge y \neq z \wedge x \overset{a}{\circ\!\!\rightarrow\!\!\circ}\ y \wedge x \overset{a}{\circ\!\!\rightarrow\!\!\circ}\ z)$$

specifies a language of graphs that contains no proper heap. □

If $\mathcal{HRL}$ is restricted to languages of heaps, we obtain the class $\mathcal{DSL}$ of hypergraph languages realizable by data structure grammars. By Proposition 4.3.1, this class is a fragment of $\mathtt{SL}_{RD}$. We remark that our examples of graph languages that are not definable in $\mathtt{SLF}_{RD}$ exploit the fact that every pointer of an object in a heap must be labeled with a different selector (see Section 2.2). Since the set of all selectors is finite, this means that the language of all singly-linked lists in which an additional vertex is connected by an outgoing edge to every list element is not $\mathtt{SL}_{RD}$ definable. A tree-like HRG generating this language is shown in Figure 6.1.

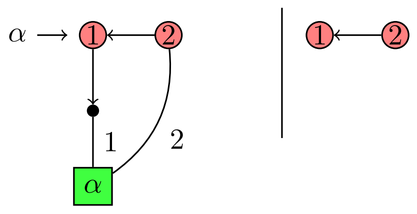


Figure 6.1.: An HRG generating singly-linked lists with an additional vertex pointing to all vertices

Furthermore, Antonopoulos and Dawar showed that simple separation logic (see Section 4.2) defined over arbitrary graphs is a fragment of MSO_2L [AD09]. It is still an open question whether MSO_2L is included in a variant of $\mathtt{SL}_{RD}$ that is defined for arbitrary graphs.

6.2. Separation Logic with MSO Definable Predicates

Recently, Iosif, Rogalewicz and Simacek presented three syntactic conditions for $\mathtt{SLF}_{RD}$ in order to obtain a fragment of separation logic with a decidable entailment problem [IRS13]. We denote this fragment by $\mathtt{SLF}_{MSO}$, because decidability is shown by a translation of separation logic formulae to MSO_2F. To our best knowledge, $\mathtt{SLF}_{MSO}$ has been developed without knowledge of the close relationship between separation logic with recursive predicates and data structure grammars discussed in Section 4.3. In fact, the decidability results for $\mathtt{SLF}_{MSO}$ motivated the further study of decidability problems for separation logic and data structure grammars in this thesis. We briefly report on the fragment $\mathtt{SLF}_{MSO}$ introduced by Iosif et al. [IRS13] and present their main results. The relationship between tree-like hyperedge replacement grammars and $\mathtt{SLF}_{MSO}$ is discussed in Section 6.3. All separation logic formulae that belong to $\mathtt{SLF}_{MSO}$ are required to be in a normal form known as *symbolic heaps* in the literature [O'H12].

Definition 6.2.1 (Symbolic Heap)**.** An $\mathtt{SLF}_{RD}$ formula φ is called a *symbolic heap* if it is of the form $\varphi(\overline{x}) \triangleq \exists \overline{r}.head_{\varphi}(\overline{x},\overline{r}) * \rho_1(\overline{y_1}) * ... * \rho_n(\overline{y_n}) \wedge pure_{\varphi}(\overline{x},\overline{r})$ where $\rho_1, ..., \rho_n$ are predicate names and $[\overline{y_i}] \subseteq [\overline{x}] \cup [\overline{r}]$ for all $i \in [1,n]$. Here, $head_{\varphi}(\overline{x},\overline{r})$ is a spatial formula, i.e. it consists only of separating conjunctions of pointer assertions and empty heaps, and $pure_{\varphi}(\overline{x},\overline{r})$ is a pure formulae, i.e. it consists only of Boolean assertions. We denote the set of existentially quantified variables by $BV(\varphi)$ and the sequence of predicate calls $\langle \rho_1(\overline{y_1}), ..., \rho_n(\overline{y_n}) \rangle$ by $tail(\varphi)$. A symbolic heap $\psi(\overline{x})$ is called *progressing* if exactly one variable occurs as the left-hand side of a pointer assertion in it, which means $head_{\psi}(\overline{x},\overline{r}) \triangleq z \mapsto (z_1, ..., z_k)$ for some $k > 0$ and $\{z, z_1, ..., z_k\} \subseteq [\overline{x}] \cup [\overline{r}]$. ■

For a progressing symbolic heap $\psi(\overline{x})$, we call a variable $x \in [\overline{x}]$ *allocated* if x occurs on the left-hand side of a pointer assertion in $head_{\psi}(\overline{x},\overline{r})$ or if x occurs as the k-th parameter of some predicate call $\rho_j(\overline{y}_j) \in tail(\psi)$, i.e. $\overline{y}_j[k] = x$ and the k-th parameter of ρ_j is allocated in the respective predicate definition. Thus, a variable is allocated if it is eventually required that it points to other locations. With these notions, we can formally introduce the fragment $\mathtt{SLF}_{MSO}$.

Definition 6.2.2 (The Fragment $\mathtt{SLF}_{MSO}$)**.** Let $\Gamma_{\mathtt{Pred}}$ be an environment over a set of predicate names `Pred`. Then, $\Gamma_{\mathtt{Pred}}$ belongs to the fragment of separation logic with MSO definable predicates, denoted by $\mathtt{SLF}_{MSO}$, if and only if the following conditions are satisfied.

1. *Progress Condition:* Every disjunction $\varphi(\overline{x})$ in every predicate definition of $\Gamma_{\mathtt{Pred}}$ is a progressing symbolic heap with additional existentially quantified variables $BV(\varphi) = \overline{r}$, pointer assertions $head_{\varphi}(\overline{x},\overline{r}) = z \mapsto (z_1, ..., z_k)$ and predicate calls

$tail(\varphi) = \langle \rho_1(\overline{y_1}), ..., \rho_n(\overline{y_n}) \rangle$ for some $k, n > 0$, $z, z_1, ..., z_k \in [\overline{x}] \cup [\overline{r}]$ and $[\overline{y_i}] \subseteq [\overline{x}] \cup [\overline{r}]$, $i \in [1, n]$.

2. *Strictness Condition:* Every pointer assertion specifies all selectors of a location, i.e. for every interpretation $\jmath$ and every location ℓ there is at most one pointer assertion $z \mapsto (z_1, ..., z_k)$ with $\jmath(z) = \ell$.

3. *Connectivity Condition:* For each predicate call $\rho_i(\overline{y_i}) \in tail(\varphi)$, there is a variable z_ℓ occurring on one of the right-hand sides of $head_\varphi(\overline{x}, \overline{r})$ and some $s \geq 0$ such that $\overline{y_i}[s] = z_\ell$ and $\overline{y_i}[s]$ is allocated in the head of each disjunction of the predicate definition of $\rho_i(\overline{y_i})$ in $\Gamma_{\texttt{Pred}}$.

4. *Establishment Condition:* Every existentially quantified variable is eventually allocated.

■

The class of hypergraph languages realizable by $\texttt{SLF}_{MSO}$ formulae is denoted by $\texttt{SL}_{MSO}$. As for tree-like HRGs, the intution behind the conditions of $\texttt{SLF}_{MSO}$ can be explained using the results by Engelfriet and Courcelle introduced in Section 5.1. The progress and connectivity condition ensure that a tree can be reconstructed from every model in MSO_2F and every vertex in this tree corresponds to one application of a recursive predicate call. Then, this tree can be used to verify the correct stepwise application of recursive predicate calls and pointer assertions. Note that our recurring example of binary trees with linked leaves can also be defined in $\texttt{SLF}_{MSO}$ as already shown in Example 4.2.3. For completeness, we report on the main theorem presented in [IRS13].

Proposition 6.2.1 (Iosif, Rogalewicz and Simacek)**.** *For every closed separation logic formula φ with an environment Γ in $\texttt{SLF}_{MSO}$, all models have bounded tree width and there exists an MSO_2F sentence ψ such that for all $h \in \texttt{Heaps}$, we have $h, \eta_\Gamma \models \varphi$ iff $h \models \psi$.* ■

For a detailed proof, we refer to [IRS13] or, in a slightly more general context, Section 6.3. We remark that the strictness condition is not explicitly stated in the original paper. However, it is required that only pointer assertions of the form $z \mapsto (z_1, ..., z_k)$ are used and [IRS13] contains an assertion in the proof of Proposition 6.2.1 which implicitly assumes that the strictness condition holds. The decidability of the satisfiability and entailment problem for $\texttt{SLF}_{MSO}$ follows directly from Proposition 6.2.1 and Proposition 4.1.2. As stated by the following lemma, $\texttt{SL}_{MSO}$ is the less expressive than other separation logic fragments considered in previous chapters.

Lemma 6.2.1. *$SL_{MSO} \subseteq SL_{HR} \subseteq SL_{RD}$.* ■

Proof (of Lemma 6.2.1). Every $\texttt{SL}_{MSO}$ formula is, syntacticly, also an $\texttt{SL}_{HR}$ formula. Thus, it suffices to show that every $\texttt{SL}_{MSO}$ formula φ with a corresponding environment Γ satisfies the dangling pointer assumption specified in Definition 4.3.1. Assume this is not the case, i.e. there are two existentially quantified variables $x \neq y$ occurring in Γ such that for some interpretation $\jmath$ and a heap h with $h, \eta_\Gamma \models^\jmath \varphi$, we have $\jmath(x) = \jmath(y)$. Then, by the establishment condition of Definition 6.2.2, there are pointer assertions $x \mapsto (u_1, ..., u_n)$ and $y \mapsto (v_1, ..., v_m)$ in Γ such that x and y are allocated. Furthermore, the strictness condition implies that there is at most one pointer assertion for each location which contradicts $\jmath(x) = \jmath(y)$. Thus, $\texttt{SL}_{MSO} \subseteq \texttt{SL}_{HR}$. By Proposition 6.2.1, every $\texttt{SL}_{MSO}$ formula can be translated into an equivalent MSO_2F formula. Thus, a context-free string language like $L = \{a^n b^n \mid n \geq 0\}$ (encoded as a graph language) is not $\texttt{SL}_{MSO}$ definable. Since every context-free string grammar can be transformed into a DSG (see Example 3.2.1) and for every DSG there exists an equivalent $\texttt{SL}_{HR}$ environment because of Proposition 4.3.1, it follows that $\texttt{SL}_{MSO}$ is strictly included in $\texttt{SL}_{MSO}$.

The inclusion $\texttt{SL}_{HR} \subseteq \texttt{SL}_{RD}$ is trivial by Definition 4.3.2. In addition to that, Lemma 6.1.1 and Proposition 4.3.1 imply that $\texttt{SL}_{RD} \not\subseteq \texttt{SL}_{HR}$. □

We now turn to the relationship between $\texttt{SL}_{MSO}$ and data structure grammars.

6.3. Directed Tree-Like Hypergraph Languages

As a consequence of Lemma 6.2.1, every $\texttt{SLF}_{MSO}$ environment can be translated into an equivalent data structure grammar (see Section 4.3). Thus, we can obtain an alternative proof of the decidability results in Proposition 6.2.1 by applying Theorem 5.2.1. In addition to that, the conditions of Iosif et al. (see Definition 6.2.2) for separation logic are interpretable in terms of data structure grammars.

In this section, we compare $\texttt{SL}_{MSO}$ to hypergraph languages realizable by tree-like data structure grammars and develop an alternative characterization of $\texttt{SL}_{MSO}$. The intuition on HR-languages based on the work of Engelfriet and Courcelle (see Section 5.1) tells us that an HR-language can be characterized by a regular tree language and a function which transforms every tree by adding and removing vertices and hyperedges. Furthermore, we obtain a decidable language inclusion problem if for each hypergraph in the language, the original tree can be reconstructed. Both formalisms introduce restrictions on possible transformations to allow such a reconstruction.

For instance, the progressing condition of $\texttt{SLF}_{MSO}$ ensures that every predicate call contains exactly one pointer assertion forming a directed tree. The root vertices of these trees must again form a tree due to the connectivity condition. In the original paper, this tree is called the "backbone" and it is shown that a backbone can be defined in MSO_2F for every $\texttt{SLF}_{MSO}$ environment [IRS13]. Intuitively, $\texttt{SLF}_{MSO}$ restricts transformations on the original tree to adding additional edges, because every existentially quantified variable must be allocated. For tree-like HRGs, removal of vertices is forbidden and edges may only be removed if a path between the vertices attached to the removed edge is added. Thus, a path-connected derivation tree is contained in every graph in the language. A detailed discussion of the properties of tree-like HRGs can be found in Section 5.2.

We observe that tree-like HRGs impose fewer restrictions on possible tree transformations. Consider, for instance, the tree-like DSG in Figure 6.2 realizing "reversed" binary trees, i.e. binary trees in which sources and targets of terminal hyperedges are switched. This hypergraph language cannot be defined in $\mathtt{SLF}_{MSO}$. To see this, note

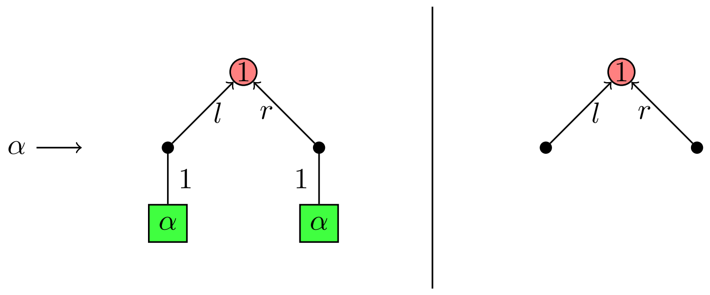


Figure 6.2.: Tree-like DSG realizing "reversed" binary trees

that every vertex - with the exception of the root - in a reversed binary tree has exactly one outgoing edge. By the progress condition, only one variable can be allocated per predicate call. Thus, one predicate call per vertex is needed to realize a reversed binary tree in $\mathtt{SLF}_{MSO}$. Moreover, the connectivity condition requires the existence of a pointer assertion from the single variable allocated in the head of a formula φ to the head of every predicate call in $tail(\varphi)$. Since every leaf of a reversed binary tree has no incoming edges, a predicate call allocating a variable that represents a leaf can only occur in the top-level formula. Then, a top-level formula ψ needs 2^n predicate calls in $tail(\psi)$ to specify a reversed binary tree of height n. Since every $\mathtt{SLF}_{MSO}$ formula is of finite length, it follows that the language of all reversed binary trees is not $\mathtt{SLF}_{MSO}$ definable. Thus, there are hypergraph languages realizable by tree-like data structure grammars that are not $\mathtt{SLF}_{MSO}$ definable. Towards an exact characterization of $\mathtt{SL}_{MSO}$ in terms of data structure grammars, we remove the ability of tree-like DSGs to reverse directions of edges. Formally, all production rules are required to map nonterminals to *directed* tree-like hypergraphs which are defined as follows.

Definition 6.3.1 (Directed tree-like hypergraph)**.** A tree-like hypergraph $\mathcal{H}$ is called *directed* if there exists a directed path from $root_{\mathcal{H}}$ to every other non-free vertex in $\mathcal{H}$. ■

For the corresponding definition of directed tree-like HRGs, we need an additional condition. Intuitively, the establishment condition for $\mathtt{SLF}_{MSO}$ formulae is stronger than the dangling pointer assumption (see Definition 4.3.1), because it prevents locations from having no outgoing selector edges. The goal of both conditions is to prevent variables from pointing to arbitrary locations if not explicitly allocated, which may lead to unwanted graph structures like square grids. In contrast to the dangling pointer assumption, however, a consequence of the establishment condition is that only the special location `null` and parameters of top-level formulae, which never correspond to an existentially quantified variable, are allowed to have no outgoing edges.

Definition 6.3.2 (Directed tree-like HRG)**.** A *directed tree-like HRG* is a tree-like HRG $\mathfrak{G}$ in which

1. the right-hand side of every production rule is a directed tree-like hypergraph,
2. there are at most k vertices $u_1, ..., u_k$ without outgoing terminal edges in every hypergraph generated by $\mathfrak{G}$, and
3. these vertices $u_1, ..., u_k$ are required to occur in every production rule as external vertices.

■

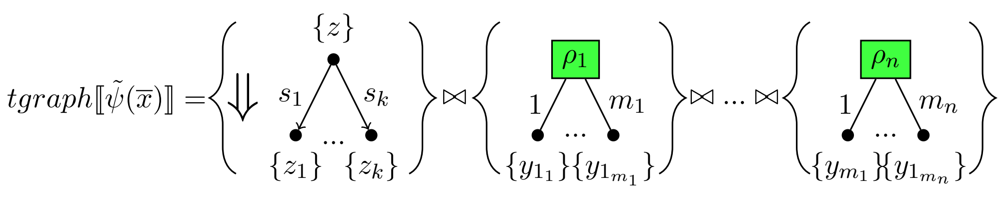
Since the vertices $u_1, ..., u_k$ correspond to `null` and parameters of a top-level formula only, it is straightforward to see that each $\texttt{SLF}_{MSO}$ environment can be extended such that variables corresponding to $u_1, ..., u_k$ occur as parameters in every predicate definition. Analogously, these vertices can be added as external (potentially free) vertices to every production rule. We denote the set of all of these HRGs by $\mathcal{DTHRG}$ and the corresponding class of hypergraphs by $\mathcal{DTHRL}$. Analogously, $\mathcal{DTDSG}$ and $\mathcal{DTDSL}$ describe the corresponding restriction to data structure grammars. Obviously, we have $\mathcal{DTHRL} \not\subseteq \mathcal{THRL}$ by definition and Figure 6.2. We now consider the relation between $\texttt{SL}_{MSO}$ and this new fragment of DSGs.

Lemma 6.3.1. *$SL_{MSO} \subseteq \mathcal{DTDSL}$.* ■

Proof (of Lemma 6.3.1). By Lemma 6.2.1, it is clear that for each $\texttt{SLF}_{MSO}$ formula φ with a corresponding environment Γ, an equivalent DSG can by obtained using the translation procedure described in Section 4.3.. Without loss of generality, we assume that φ and all formulae in Γ are extended such that all parameters of φ are also parameters of every predicate definition. We show that this procedure yields a directed tree-like DSG. At first, we show that any formula ψ occurring in a predicate definition in Γ is translated into a directed tree-like hypergraph. By Definition 6.2.2, every such formula ψ is of the form

$$\psi(\overline{x}) \triangleq \exists \overline{r}.head_\psi(\overline{x}, \overline{r}) * \rho_1(\overline{y_1}) * ... * \rho_n(\overline{y_n}) \wedge pure_\psi(\overline{x}, \overline{r})$$

where $head_\psi(\overline{x}, \overline{r}) = z \mapsto (z_1, ..., z_k)$ with $\{z, z_1, ..., z_k\} \subseteq [\overline{x}] \cup [\overline{r}]$ and $[\overline{y_i}] \subseteq [\overline{x}] \cup [\overline{r}]$ for each $i \in [1, n]$. For simplicity, let $\tilde{\psi}(\overline{x})$ be the formula $\psi(\overline{x})$ without the pure part. Then, by Definition 4.3.4,



and $tgraph[\![\psi(\overline{x})]\!] = \{\Downarrow \mathcal{H} \mid \mathcal{H} \in tgraph[\![\tilde{\psi}(\overline{x})]\!]\}$. Clearly, all terminal edges have rank two. Furthermore, the vertices $z, z_1, ..., z_k$ with different tags form a tree of height one with root z and all other vertices occur free, i.e. are only connected to nonterminal hyperedges. Since $\psi(\overline{x})$ belongs to some predicate definition and therefore satisfies the connectivity condition of Definition 6.2.2, we have $z \in [\overline{x}]$, which means z is an external vertex. By the same reasoning, the connectivity condition implies that there exists a sequence of vertices $\overline{context} = v_1...v_\ell$ with $v_i \neq v_j$ for $i \neq j$ such that v_i is attached to the i-th nonterminal hyperedge. Due to the connectivity and the strictness condition, these vertices can always be chosen such that the corresponding variables in ψ are allocated in the head of the next predicate call of ρ_i. Thus, $expose(\mathcal{H}, \overline{x})$ is a directed tree-like hypergraph for $\mathcal{H} \in tgraph[\![\psi(\overline{x})]\!]$ and the DSG $\mathfrak{G}_\Gamma$ obtained from Γ contains only directed tree-like production rules.

Now, let $T_\mathfrak{a}$ be the tree with $root_\mathfrak{a}$ as root and exactly the vertices in $\overline{context_\mathfrak{a}}$ as leaves for every production rule $\mathfrak{a} = \alpha \to \mathcal{A}$ in $\mathfrak{P}_\Gamma$. To complete the proof, we have to show that replacing every production rule $\mathfrak{a} = \alpha \to \mathcal{A}$ of $\mathfrak{G}$ by a rule $\alpha \to T_\mathfrak{a}$ induces a regular tree grammar. Assume this is not the case and let $\xi^\bullet = \mathcal{H}_0 \overset{\mathfrak{a}_1}{\Longrightarrow}_\mathfrak{G} ... \overset{\mathfrak{a}_n}{\Longrightarrow}_\mathfrak{G} \mathcal{H}_n$ be a derivation such that the composition of trees $T_{\mathfrak{a}_i}$, $i \in [1, n]$ is not a tree again. Then, for some $1 \leq k < n$, T_k is a tree and T_{k+1} is not. Let e_{k+1} be the nonterminal hyperedge replaced in the $k+1$-th step of the derivation in question. By construction of T_k, there exists a vertex v in $\overline{context_{\mathfrak{a}_i}}$ for some $i \in [1, k]$ which belongs to this hyperedge. Since v is chosen in compliance with the connectivity condition, we have $v = root_{\mathfrak{a}_{k+1}}$. Furthermore, the strictness condition asserts that v is not equal to some other vertex in T_k. By the same argumentation, $\overline{context_{\mathfrak{a}_{k+1}}}$ is either empty or consists only of vertices which do not correspond to vertices already in T_k. Obviously $v(\overline{context_{\mathfrak{a}_{k+1}}})$ is a tree, because every production rule is directed tree-like and by construction of $\texttt{SLF}_{MSO}$ formulae non-free vertices even form a tree of height one. Thus, T_{k+1} is a tree again, which contradicts the assumption.

Finally, only vertices corresponding to `null` and parameters of the top-level formula φ may not have outgoing vertices, because the establishment condition enforces the existence of a pointer assertion of the form $x \mapsto (y_1, ..., y_k)$ for every other variable x. By assumption, all predicate definitions contain these variables as parameters. Thus, by definition of the *expose* function, each vertex without outgoing edges occurs as an external vertex in every production rule.

Since all conditions of Definition 6.3.1 and Definition 5.2.2 are satisfied, $\mathfrak{G}_\Gamma$ is a directed tree-like HRG. Then, the claim follows directly from Proposition 4.3.1. □

Before we look at the converse statement, we make two observations about directed tree-like DSGs. The first observation allows us to assume that only one vertex in every hypergraph that occurs on the right-hand side of some production rule has outgoing edges.

Lemma 6.3.2. *Every directed tree-like DSG $\mathfrak{G} = (N, T, \mathfrak{P})$ can be transformed into an equivalent one where every production rule $\mathfrak{a} = \alpha \to \mathcal{A} \in \mathfrak{P}$ maps to a hypergraph in which the non-free vertices form a tree of height one with $root_\mathfrak{a}$ as root.* ■

Proof (of Lemma 6.3.2). At first, we construct a directed tree-like DSG $\mathfrak{G}_\mathfrak{a}$ realizing $\{\mathcal{A}\}$ for every production rule $\mathfrak{a} = \alpha \to \mathcal{A}$. Intuitively, all vertices of $\mathcal{A}$ are introduced in the first derivation step, potentially as free vertices. Then, we move along a spanning tree of $\mathcal{A}$ adding exactly the outgoing terminal edges of one single vertex in this tree in every production rule. Thus, every nonterminal hyperedge e is attached to all vertices v of $\mathcal{A}$ such that some subtree of the tree represented by e contains a vertex with an outgoing edge to v. These DSGs are then composed into a new DSG $\mathfrak{G}'$ with $L(\mathfrak{G}, \xi) = L(\mathfrak{G}', \xi)$ for every nonterminal ξ.

Formally, let $\mathfrak{a} = \alpha \to \mathcal{A}$ be a production rule and $T_\mathfrak{a}$ be a directed spanning tree of the non-free vertices of $\mathfrak{a}$ with $root_\mathfrak{a}$ as root. Note that such a tree always exists, because $\mathfrak{G}$ is directed tree-like (see Section 5.2). Furthermore, let $T_\mathfrak{a}(v)$ denote the subtree of $T_\mathfrak{a}$ with root v and $ET_\mathfrak{a}(v) = \{u \in V_\mathfrak{a} \mid \exists w \in T_\mathfrak{a}(v). w \to u\}$ denote the set of vertices with an incoming edge from some vertex in $T_\mathfrak{a}(v)$, respectively. For convenience, we write $Param_\mathfrak{a} = \{w_1, ..., w_k\}$ to denote the set of all vertices without outgoing (terminal) edges in $\mathcal{A}$ and $vE_\mathcal{H} = \{u \in V_\mathcal{H} \mid \exists e \in E_\mathcal{H}. v \xrightarrow{e} u\}$ to denote the set of all vertices reachable from v by an outgoing edge.

We introduce a new nonterminal symbol ν_v for each vertex $v \in V_\mathfrak{a}$ with $vE_\mathfrak{a} \neq \emptyset$ and exactly one production rule $\mathfrak{v} = \nu_v \to \mathcal{V}_v$ for each of these nonterminals. These production rules map to hypergraphs $\mathcal{V}_v = (V_v, E_v, src_v, tgt_v, lab_v, \overline{ext_v})$ where

- V_v consists of all vertices in the subtree $T_\mathfrak{a}(v)$, all vertices pointed to by some vertex in this tree, all free vertices of $\mathfrak{a}$ and $Param_\mathfrak{a}$, i.e. $V_v = V_{T_\mathfrak{a}(v)} \cup ET_\mathfrak{a}(v) \cup Param_\mathfrak{a} \cup free(\mathfrak{a})$,

- E_v consists of all outgoing terminal hyperedges starting in v, a new nonterminal hyperedge e_{ν_u} for each vertex reachable from v by exactly one directed edge and all original nonterminal hyperedges such that the corresponding context vertex is a direct successor of v in $T_\mathfrak{a}(v)$, i.e.

$$E_v = vE_\mathfrak{a} \cup \{e_{\nu_u} \mid u \in vE_\mathfrak{a} \wedge uE_\mathfrak{a} \neq \emptyset\} \cup \{e \in E_\mathfrak{a} \mid att_\mathfrak{a}(e)[1] \in vE_\mathfrak{a} \cap E_{T_\mathfrak{a}(v)}\},$$

- src_v and tgt_v together define all original attachments of edges in $\mathfrak{a}$ and attach every new hyperedge e_{ν_u} to all vertices occurring in the corresponding subtree of $T_\mathfrak{a}$ plus all free vertices, i.e.

$$att_v(e) = \begin{cases} u \cdot ET_\mathfrak{a}(u) \cdot \overline{free(\mathfrak{a})} \cdot \overline{Param_\mathfrak{a}} & \text{if } e = e_{\nu_u} \\ att_\mathfrak{a}(e) & \text{otherwise,} \end{cases}$$

- lab_v labels every new hyperedge with the corresponding new nonterminal and uses the original labeling otherwise, i.e.

$$lab_v(e) = \begin{cases} \nu_u & \text{if } e = e_{\nu_u} \\ lab_\mathfrak{a}(e) & \text{otherwise, and} \end{cases}$$

- the sequence of external vertices corresponds to the external vertices of $\mathfrak{a}$ for the root vertex and consists of all vertices otherwise, i.e.

$$\overline{ext_v} = \begin{cases} \overline{ext_{\mathfrak{a}}} & \text{if } v = root_{\mathfrak{a}} \\ \langle V_v \rangle & \text{otherwise.} \end{cases}$$

Here, we implicitly assume that the sequence $\langle V_v \rangle$ chosen for external vertices is compatible with the order of attachments defined in att_v. Since every new nonterminal ν_u occurs exactly once and there is exactly one production rule for each of these nonterminals, we have that $L(\mathfrak{G}_{\mathfrak{a}}, \nu_{root_{\mathfrak{a}}})$ is a singleton set. Then, the single hypergraph generated by $\mathfrak{G}_{\mathfrak{a}}$ is equal to $\mathcal{A}$. Furthermore, every production rule of $\mathfrak{G}_{\mathfrak{a}}$ is a directed tree-like hypergraph and, by construction and the existence of a tree $T_{\mathfrak{a}}$, it follows that $\mathfrak{G}_{\mathfrak{a}}$ is a directed tree-like DSG.

Now, the new DSG $\mathfrak{G}'$ consists of all production rules of $\mathfrak{G}_{\mathfrak{a}}$ for each $\mathfrak{a} \in \mathfrak{P}$ and additionally production rules $\alpha \to \mathcal{V}_{root_{\mathfrak{a}_i}}$ for each $\mathfrak{a}_i \in \mathfrak{P}(\alpha)$ where $\mathfrak{P}(\alpha) = \{\mathfrak{a}_1, ..., \mathfrak{a}_k\}$ denotes the set of all production rules of the original DSG $\mathfrak{G}$ corresponding to the nonterminal $\alpha \in N$. This DSG is again directed tree-like, note that all vertices without outgoing edges are collected in $Param_{\mathfrak{a}}$ for each DSG $\mathfrak{G}_a$, and by construction and the fact that every derivation respecting the partial order defined by a derivation tree of $\mathfrak{G}$ (and $\mathfrak{G}'$) yields the same hypergraph (see Definition 3.2.7), it follows that $L(\mathfrak{G}, \xi) = L(\mathfrak{G}', \xi)$. □

In addition to that, we observe that all outgoing edges of a vertex can be added in the production rule that introduces a vertex as a non-external vertex. This is formalized in the following lemma.

Lemma 6.3.3. *Every directed tree-like DSG $\mathfrak{G} = (N, T, \mathfrak{P})$ can be transformed into an equivalent one in which for every derivation*

$$\mathcal{H}_0 \xRightarrow{\mathfrak{a}_1}_{\mathfrak{G}} ... \xRightarrow{\mathfrak{a}_n}_{\mathfrak{G}} \mathcal{H}_n \textit{ with}$$

$\mathfrak{a}_i \in \mathfrak{P}$ *for each* $i \in [1, n]$*, and every vertex* $v \in \mathcal{H}_i$*, there is at most one derivation step adding outgoing edges to* v*.* ■

We sketch the main idea of the proof without providing a complete formal construction of the resulting directed tree-like DSG.

Proof (of Lemma 6.3.3). We show the claim by complete induction over the maximal number n of outgoing edges that are added to a vertex v by production rules in which v is represented by an external vertex. For $n = 0$, the claim holds trivially. Thus, assume that the claim holds for $n \geq 0$ and let $\mathfrak{G} = (N, T, \mathfrak{P})$ be a directed tree-like DSG where $n + 1$ outgoing edges are added to a vertex v by production rules in which v is represented by an external vertex. Furthermore, let $\mathfrak{a} \in \mathfrak{P}$ be the production rule which introduces v as a non-external vertex. By Lemma 6.3.2, we can assume without loss of generality that every production rule $\mathfrak{a} = \alpha \to \mathcal{A} \in \mathfrak{P}$ maps to a hypergraph where the non-free vertices form a tree of height one with $root_{\mathfrak{a}}$ as root. For simplicity, we assume that a vertex u exists such that there is an outgoing edge from v to u in every derivation of $\mathfrak{G}$. If this is not the case, every possible choice of outgoing edges can be realized by adding production rules $\mathfrak{a}_1, ..., \mathfrak{a}_k$ representing the available choices of outgoing edges for v. Thus, intuitively, all outgoing edges of v must be guessed when applying the first production rule that contains v.

Now, add a vertex u' to $\mathfrak{a}$ together with an edge from v to u' as in the production rule $\mathfrak{b} \in \mathfrak{P}$ that adds an edge between v and u. Furthermore, attach u' to every nonterminal v is attached to in $\mathfrak{a}$. Then, this edge is removed from $\mathfrak{b}$ and u is replaced by the external vertex corresponding to u'.

The same must be done for every production rule in which v might be represented by an external vertex, i.e. the rank of all nonterminals v is attached to increases by one. The resulting production rules again map to directed tree-like hypergraphs. In order to see that the context attachment and rootedness conditions are satisfied for each production rule, note that the non-external vertices form a tree of height one by assumption. Then, only n outgoing edges are added to v by production rules in which v is an external vertex. Thus, by induction hypothesis, there exists an equivalent directed tree-like DSG where all outgoing edges are added in one derivation step. $\square$

With these two observations, we can show that $\texttt{SLF}_{MSO}$ is as expressive as directed tree-like DSGs.

Lemma 6.3.4. $\mathcal{DTDSL} \subseteq \mathcal{SL}_{MSO}$. $\blacksquare$

Proof (of Lemma 6.3.4). Without loss of generality, we assume that $\mathfrak{G}$ is a directed tree-like DSG in which there is at most one derivation step for every vertex that adds outgoing edges to this vertex (see Lemma 6.3.3). Furthermore, by Lemma 6.3.2, we assume that $\mathfrak{G}$ is a (normalized) directed tree-like DSG where the non-free vertices of every production rule $\mathfrak{a} = \alpha \to \mathcal{A}$ form a tree of height one with $root_{\mathfrak{a}}$ as root. Then, it is straightforward to see that every production rule $\mathfrak{a} = \alpha \to \mathcal{A}$ is translated according to Definition 4.3.7 into a formula of the form

$$\psi(\overline{x}) \triangleq \exists \overline{r}.head_{\psi}(\overline{x}, \overline{r}) * \rho_1(\overline{y_1}) * ... * \rho_n(\overline{y_n})$$

where $head_{\psi}(\overline{x}, \overline{r})$ is a single pointer assertion of the form $z \mapsto (z_1, ..., z_k)$ corresponding to the tree defined in $\mathcal{A}$ and $\overline{y_i}$ corresponds to the sequence of attached vertices for each nonterminal hyperedge. Let $\Gamma_{\mathfrak{G}}$ be the environment obtained by applying the translation procedure defined in Section 4.3 to $\mathfrak{G}$. Since at most one production rule adds outgoing edges to a vertex, $\Gamma_{\mathfrak{G}}$ satisfies the progress and strictness condition of 6.2.2. Furthermore, the connectivity condition directly follows from the fact that every DSG derivation contains a tree which is composed of trees corresponding to each production rule.

Furthermore, only a fixed number of vertices $u_1, ..., u_k$ may not have outgoing terminal edges in every hypergraph realized by $\mathfrak{G}$ and these vertices occur as external vertices in every production rule. Then, these vertices must be introduced as non-external vertices in the initial handle already, i.e. they correspond to parameters of a top-level formula or to `null`. Since all other vertices have outgoing edges, all variables not representing one of the vertices $u_1, ..., u_k$ are allocated, i.e. the establishment condition is also satisfied. Hence, the claim holds by Proposition 4.3.1. □

For reference, we collect the results of Lemma 6.3.4 and Lemma 6.3.1 in the following theorem.

Theorem 6.3.1. *$\mathtt{SL}_{MSO}$=$\mathcal{DTDSL}$.* ■

For the rest of this section we consider some implications of Theorem 6.3.1 when looking at $\mathtt{SL}_{MSO}$ from a language-theoretic perspective. First of all, every regular tree language (see Definition 3.2.6) and every regular string language (by Example 3.2.1 applied to right-linear context-free string grammars) can be generated by directed tree-like DSGs and therefore can be characterized by $\mathtt{SLF}_{MSO}$ formulae. Furthermore, we obtain an alternative proof of Proposition 6.2.1 as a direct consequence of Theorem 6.3.1 and Theorem 5.2.1. The close relationship between $\mathtt{SL}_{MSO}$ and directed tree-like DSGs is also interesting regarding the complexity of the satisfiability/emptiness and entailment/inclusion problem. By the time of writing this thesis, complexity questions for $\mathtt{SL}_{MSO}$ are only addressed by Antonopoulos et al. in [AGH$^+$14]. We remark that other notions of separation logic with inductive predicates have been analyzed in the literature, for instance in [BFPG14]. However, these notions are not equivalent to the fragment $\mathtt{SL}_{MSO}$. Antonopoulos et al. show that entailment for $\mathtt{SL}_{MSO}$ is ExpTime-hard. Since every regular tree language can be generated by a directed tree-like DSG and the language inclusion problem for regular tree-languages is ExpTime-complete (see Proposition 5.3.1), it also directly follows from Theorem 6.3.1 that the entailment problem for $\mathtt{SL}_{MSO}$ is ExpTime-hard. To our best knowledge, an upper bound for the entailment problem of $\mathtt{SL}_{MSO}$ is still unknown. Based on Proposition 5.3.1 and similarity between tree-like hypergraph languages and regular languages (of strings and trees), we conjecture that no elementary upper bound exists.

For the satisfiability problem, we can obtain more precise results. At first, note that the use of negation and conjunction is very restricted in $\mathtt{SLF}_{MSO}$ and therefore an algorithm solving the satisfiability problem does not provide an algorithm for the entailment problem by checking $\varphi \rightarrow \psi$ for two $\mathtt{SLF}_{MSO}$ formulae φ and ψ. In addition to that, a translation of tree-like DSGs or $\mathtt{SLF}_{MSO}$ formulae to MSO_2F formulae as used in the proof of Theorem 5.2.1 or in [IRS13] does not provide further insight regarding the complexity, because no upper bound of the satisfiability problem for MSO_2F formulae is known (see Proposition 4.1.1). However, we can always construct a (normalized) directed tree-like data structure grammar for every $\mathtt{SLF}_{MSO}$ environment and by Theorem 5.3.1, the complexity of the satisfiability problem for $\mathtt{SLF}_{MSO}$ only depends on the complexity of this translation procedure. Hence, it is solvable in polynomial time.

6.4. Local Separation Logic with MSO Definable Predicates

Recently, Iosif et al. proposed another fragment of separation logic with recursive definitions where the entailment problem can be reduced to the language inclusion problem of tree automata [IRV14]. As a consequence, the entailment problem for this fragment is ExpTime-complete. Additionally to the syntactic constraints of $\texttt{SLF}_{MSO}$ (see Definition 6.2.2), this fragment requires a *locality condition* defined as follows.

Definition 6.4.1 (local $\texttt{SLF}_{MSO}$ environment)**.** An $\texttt{SLF}_{MSO}$ environment Γ is called *local* if and only if for each predicate definition $\rho(x_1, ..., x_n) = \bigvee_{j=1}^{m} \rho_j(x_1, ..., x_n)$ each parameter x_i is either

1. allocated in each formula $\rho_j(x_1, ..., x_n)$ and the j-th parameter of $\rho(x_1, ..., x_n)$ is referenced at each occurrence of ρ in other predicate definitions of Γ, or
2. referenced in each formula $\rho_j(x_1, ..., x_n)$ and the j-th parameter of $\rho(x_1, ..., x_n)$ is allocated at each occurrence of ρ in other predicate definitions of Γ.

■

The underlying intuition of this definition can again be explained based on the results of Engelfriet and Courcelle discussed in Section 5.1. The key idea is that it is sufficient to ensure that every heap h satisfying such an environment Γ can be written as a directed graph whose corresponding undirected graph is a tree. Since every HR-language can be decomposed into a regular tree language T and a function $f : \mathit{Trees} \rightharpoonup \mathit{Graphs}$ which is definable by finitely many MSO_2F formulae, this means that f is restricted to just two operations: either use an edge in just one direction or in both directions in the final graph. Especially, other possible transformations of such functions like relabeling, removal and addition of vertices and edges are forbidden. For every edge connected to a vertex, the directions used by f can be encoded in the vertex label, called *tiles* in [IRV14] such that the set of undirected graphs corresponding to $f(T)$ is a regular tree language. Now, in order to solve the entailment problem for local $\texttt{SLF}_{MSO}$ environments, there is only one caveat. Two hypergraphs may be isomorphic although they are represented by different (undirected) trees. For instance, every vertex of a doubly-linked list may be chosen as root node leading to as many different possible (undirected) trees as there are elements in the list (see Figure 6.3). The main result of [IRV14] is that this ambiguity can be lifted, because the set of all of these "rotated" trees is again a regular tree language which can be constructed in polynomial time. Thus, the entailment problem of local $\texttt{SLF}_{MSO}$ environments can be reduced to the language inclusion problem of regular tree automata which is ExpTime-complete by Proposition 5.3.1.

In [IRV14], Iosif et al. remark that it is an open problem whether one can decide for an $\texttt{SLF}_{MSO}$ environment if there exists an equivalent local $\texttt{SLF}_{MSO}$ environment. As a further consequence of Theorem 6.3.1, this question is equivalent to the question whether the language of undirected hypergraphs realized by a directed tree-like data structure grammar is a regular tree language. This problem is in fact decidable. To see this, note that due to Theorem 5.2.1 every directed tree-like data structure grammar $\mathfrak{G} = (N, T, \mathfrak{P})$

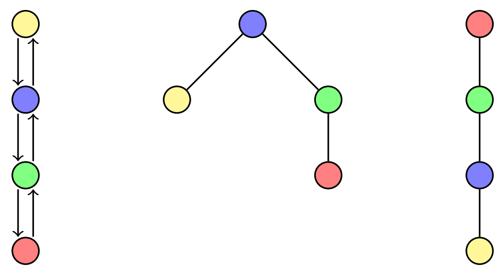


Figure 6.3.: A doubly-linked list with two possible tree-representations

with initial nonterminal $\xi \in N$ generating only trees realizes a regular tree language. It is straightforward to construct an HRG $\mathfrak{G}'$ realizing exactly the language of undirected hypergraphs of $\mathfrak{G}$. Furthermore, the set of all graphs representing undirected trees is obviously MSO_2F definable. Let φ be an MSO_2F sentence defining this set. Then, by Theorem 5.2.1 and Theorem 4.1.2, it is decidable whether $L(\mathfrak{G}', \xi) \cap L(\neg\varphi) = \emptyset$. If this is the case, every hypergraph in $L(\mathfrak{G}', \xi)$ is an undirected tree, i.e. $L(\mathfrak{G}, \xi)$ can be defined by a local $\texttt{SLF}_{MSO}$ environment. Otherwise, there exists a hypergraph in $L(\mathfrak{G}', \xi)$ which is not an undirected tree. Thus, $L(\mathfrak{G}', \xi)$ is not a regular tree language which implies that $L(\mathfrak{G}, \xi)$ is not definable by a local $\texttt{SLF}_{MSO}$ environment. Since this problem is decidable, the question whether an $\texttt{SLF}_{MSO}$ environment can be represented by an equivalent local one is decidable.

6.5. Undirected Tree-Like Hypergraph Languages

The syntactic conditions of $\texttt{SLF}_{MSO}$ do not capture all hypergraph languages that can be translated into tree-like data structure grammars, like the language of reversed binary trees shown in Figure 6.2. Thus, the question arises whether there exists a syntactic fragment of $\texttt{SL}_{RD}$ characterizing exactly the class $\mathcal{TDSL}$. Such a fragment would extend the fragment $\texttt{SL}_{MSO}$ of Iosif et al. (see Section 6.2) and still have a decidable entailment problem. By Figure 6.2 and Theorem 6.3.1 there are two aspects of $\texttt{SLF}_{MSO}$ which are too weak to capture the class $\mathcal{TDSL}$: The direction of pointer assertions is fixed due to the connectivity condition and the establishment condition restricts the number of unallocated variables.

In order to lift the first restriction, we allow any graph structure connected via one designated vertex in the head of a symbolic heap instead of directed trees of height one.

Definition 6.5.1 (Rooted Symbolic Heap)**.** A symbolic heap $\varphi(\overline{x})$ is called *rooted* if there exists a variable $u \in [\overline{x}]$ such that for every pointer assertion in

$$head_{\varphi}(\overline{x}, \overline{r}) = y_1 \mapsto \overline{z_1} * ... * y_n \mapsto \overline{z_n}$$

either $u = y_i$ or $u \in [\overline{z_i}]$ holds for all $i \in [1, n]$. The variable u is also called the root of $\varphi(\overline{x})$. ■

Furthermore, we require that the dangling pointer assumption introduced by Jansen et al. [JGN14] (see Definition 4.3.1) is satisfied instead of the stronger establishment condition of Iosif et al. [IRS13]. The result is the following more expressive version of separation logic with MSO_2F definable predicates.

Definition 6.5.2 (Tree-like Separation Logic with MSO_2F Definable Predicates)**.** Let $\Gamma_{\texttt{Pred}}$ be an environment over a set of predicate names Pred. Then, $\Gamma_{\texttt{Pred}}$ belongs to the fragment of tree-like separation logic with MSO_2F definable predicates, denoted by $\texttt{SLF}_{tree-like}$, if and only if the following conditions are satisfied.

1. *Rootedness:* Every disjunction $\varphi(\overline{x})$ in every predicate definition of $\Gamma_{\texttt{Pred}}$ is a rooted symbolic heap with root u, additional existentially quantified variables $BV(\varphi) = \overline{r}$, and predicate calls $tail(\varphi) = \langle\rho_1(\overline{y_1}), ..., \rho_n(\overline{y_n})\rangle$, respectively, for some variables $\overline{y_i}$ and $k, n > 0$.
2. *Weak Connectivity Condition:* For each predicate call $\rho_i(\overline{y_i}) \in tail(\varphi)$, there is a variable z_ℓ occurring on the right-hand side of $head_\varphi(\overline{x}, \overline{r})$ and some $s \geq 0$ such that $\overline{y_i}[s] = z_\ell$ is the root of each disjunction of the predicate definition of $\rho_i(\overline{y_i})$ in $\Gamma_{\texttt{Pred}}$.
3. *Dangling Pointer Assumption:* As introduced in Definition 4.3.1, two existentially quantified variables, which may not occur in the same formula, are different if not explicitly stated otherwise.

■

The main difference between $\texttt{SL}_{MSO}$ and $\texttt{SL}_{tree-like}$ is that a tree contained in a specified heap is not required to be directed anymore. In addition to that, the number of unallocated variables may be arbitrary, because the establishment condition is replaced by the dangling pointer assumption. The proof to show $\texttt{SL}_{tree-like}$=$\mathcal{TDSL}$ is analogous to the proof of Theorem 6.3.1 and thus omitted.

6.6. Overview

The relationships with respect to language inclusion between MSO_2L, $\texttt{SL}_{RD}$ and $\mathcal{HRL}$ as well as their respective fragments are illustrated in Figure 6.4. We showed in Lemma 6.1.1 that the classes MSO_2L, $\texttt{SL}_{RD}$ and $\mathcal{HRL}$ are incomparable. If $\texttt{SLF}_{RD}$ is extended to arbitrary hypergraphs, the relationship to MSO_2L is still open. We reported on the equivalence between data structure grammars and $\texttt{SL}_{HR}$ in Section 4.3. Theorem 5.2.1 states that the class of hypergraph languages realizable by tree-like HRGs is MSO_2F definable. The relationship between $\texttt{SL}_{MSO}$ (see Section 6.2) is clarified in Section 6.3. For example, the language of binary trees where all leaves are additionally connected by a singly-linked list (see Figure 3.4) lies in this class. However, $\texttt{SL}_{MSO}$ does not capture all hypergraph languages that can be realized by tree-like data structure grammars. For instance, the language of reversed binary trees (see Figure 6.2) cannot be specified in $\texttt{SLF}_{MSO}$. We presented an extended version of $\texttt{SLF}_{MSO}$ that coincides with the class of

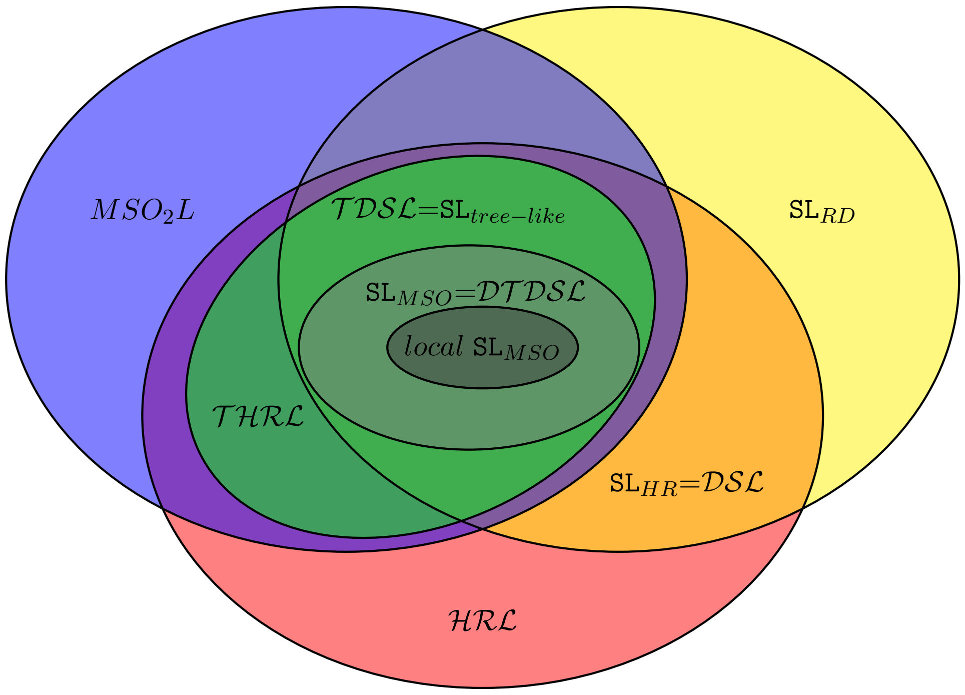

MSO_2L
$\mathcal{TDSL}$=$\mathtt{SL}_{tree-like}$
$\mathtt{SL}_{RD}$
$\mathtt{SL}_{MSO}$=$\mathcal{DTDSL}$
$local$ $\mathtt{SL}_{MSO}$
$\mathcal{THRL}$
$\mathtt{SL}_{HR}$=$\mathcal{DSL}$
$\mathcal{HRL}$

Figure 6.4.: Relationships between formalisms considered in this thesis

tree-like data structure grammars in Section 6.5. The fragment of local $\mathtt{SLF}_{MSO}$ definable hypergraph languages is the smallest class considered in this thesis (see Section 6.4) and has originally been introduced by Iosif et al. [IRV14].

With regards to decidability questions, the entailment / language inclusion problem is undecidable in the most general case for each of the three formalisms $MSO_2L, \mathcal{HRL}$ and $\mathtt{SL}_{RD}$. The same holds true for the intersections $MSO_2L \cap \mathtt{SL}_{RD}$ and $\mathtt{SL}_{RD} \cap \mathcal{HRL}$. The intersection $MSO_2L \cap \mathcal{HRL}$, however, has a decidable entailment problem by Proposition 4.1.2 and all fragments of separation logic and HRGs considered in this thesis lie in this intersection. Table 6.1 summarizes all decidability results considered in this thesis.

Note that no syntactic characterization of the complete fragment $MSO_2L \cap \mathcal{HRL}$ is known and probably does not exist (see Section 4.1). This is unfortunate, because this fragment is a natural candidate for a proper notion of regular hypergraph languages. Analogously, there is no syntactic characterization for the intersection of MSO_2L and graph languages realizable by data structures grammars. Tree-like HRGs and $\mathtt{SLF}_{MSO}$ formulae capture fragments of this intersection, but only for connected graphs. It is an open question whether a hypergraph language $L \in MSO_2L \cap \mathcal{HRL}$ consisting only of connected graphs exist which is not realizable by a tree-like HRG. In Section 6.3, it is shown that the language inclusion / entailment problem for each fragment in MSO_2L $\cap$ $\mathcal{HRL}$ considered in this thesis is ExpTime-hard. An upper bound, however, is only

class of languages	entailment / inclusion decidable	satisfiability / emptiness decidable	reference
$\mathcal{HRL}$	$\times$	✓	Section 3.2
$\mathcal{DSL}$	$\times$	✓	Section 3.3
MSO_2L	$\times$	$\times$	Section 4.1
$\texttt{SL}_{RD}$	$\times$	$\times$	Section 4.2
$\texttt{SL}_{HR}$	$\times$	✓	Section 4.3
$\mathcal{THRL}$	✓	✓	Section 5.2
$\mathcal{DTDSL}$	✓	✓	Section 6.3
local $\texttt{SL}_{MSO}$	✓	✓	Section 6.4
$\texttt{SL}_{tree-like}$	✓	✓	Section 6.5
$\mathcal{TDSL}$	✓	✓	Section 6.5

Table 6.1.: decidability results for classes of hypergraph languages

known for the smallest fragment, local $\texttt{SL}_{MSO}$, where the entailment problem can be reduced to the language inclusion problem for tree automata, i.e. the entailment problem for local $\texttt{SLF}_{MSO}$ formulae is EXPTIME-complete (see Section 6.4).

CHAPTER 7

Conclusion

This thesis discusses the relationship between various fragments of separation logic with recursive definitions and graph languages realizable by hyperedge replacement grammars with special attention to their entailment and language inclusion problem, respectively. In this chapter, we recapitulate our main results and provide an outlook for future research.

7.1. Summary

Dealing with dynamic data structures for formal verification poses a complex challenge, because it usually requires working with infinite state spaces. Thus, formalisms to represent infinite languages of memory states - heaps - by finite structures are needed. An especially interesting problem for such structures is the language inclusion problem, i.e. the question whether all heaps specified by one model are also specified by another model. For example, a decision procedure solving this problem can be applied to check properties of heaps which are expressed in the same formalism as the language under consideration. In the related domain of model-checking, there is a close relationship between graphical structures, like transition systems, modeling a system and logical formalisms, like linear temporal logic, which specify properties a model should have. For these graphical and logical formalisms, the language inclusion problem for graphical structures coincides with the entailment problem for logical specification languages.

In this thesis, we studied a similar relationship for two formalisms modeling abstract dynamic data structures. On the one hand, we considered hyperedge replacement grammars as a graphical modeling language introduced by Habel [Hab92]. This language is an extension of classical context-free string grammars to hypergraphs and based on the idea of hyperedge replacement in which a hyperedge is removed and a hypergraph of appropriate type is inserted instead. Furthermore, we reported on a restriction of hyperedge replacement called data structure grammars by Jansen et al. [JGN14] such that only valid heaps can be modeled. On the other hand, separation logic is a logical formalism to specify sets of heaps. Separation logic has been introduced by Reynolds [Rey00] and extends Hoare logic by the separating conjunction to allow local reasoning about memory states.

We considered a variant of separation logic called separation logic with recursive definitions. Both formalisms have an undecidable language inclusion and entailment problem, respectively. For HRGs, this can be shown by a straightforward reduction to the language inclusion problem for context-free string languages. For separation logic, this was first shown by Antonopoulos et al. [AGH$^+$14]. However, Iosif et al. provide a fragment of $\texttt{SLF}_{RD}$, denoted by $\texttt{SLF}_{MSO}$, with a decidable entailment problem by a reduction to monadic second-order logic over graphs with bounded tree-width [IRS13]. Inspired by their work and the relationship between $\texttt{SLF}_{RD}$ and HRGs studied by Dodds [Dod08] and Jansen et al. [JGN14] for data structure grammars, we studied different fragments of HRGs and separation logic with decidable language inclusion and entailment problems. The intuition underlying these fragments was based on the extensive work of Engelfriet and Courcelle [Cou90] [Cou91] [SR97] [CE12] which can be summarized as follows: Every HR-language L can be decomposed into a regular tree language T and a function $h : Trees \to HRA^\star$ definable by finitely many MSO_2F formulae such that $L = h(T)$. Thus, intuitively, fragments of HRGs and $\texttt{SLF}_{RD}$ with decidable language inclusion and entailment problems can be obtained if they contain "enough structure" to effectively reconstruct the tree language T from the actual language L. We showed that the conditions of Iosif et al. can be explained in terms of this intuition. Furthermore, we introduced the notion of *tree-like* HRGs in which a local fragment of a derivation tree can be reconstructed from every production rule. These fragments are additionally required to contain a regular tree grammar which is again compliant with the intuition based on Courcelle's results. One of our main results is that these conditions are sufficient to formally specify an MSO_2F sentence capturing exactly the hypergraphs realized by a tree-like HRG. Thus, the language inclusion problem is decidable for tree-like HRGs. Due to the close relationship between data structure grammars and $\texttt{SLF}_{RD}$, we discussed fragments of separation logic corresponding to tree-like HRGs. By restricting the direction and the number of vertices without outgoing edges, we showed that the fragment $\texttt{SLF}_{MSO}$ of Iosif et al. [IRS13] is a proper fragment of hypergraph languages realized by tree-like data structure grammars. Thus, we also obtained an alternative proof of their results and additionally an algorithm to check the satisfiability problem of $\texttt{SLF}_{MSO}$ formulae in polynomial time which - to our best knowledge - has been unknown without the link between separation logic and DSGs. Moreover, we looked at the fragment of local $\texttt{SLF}_{MSO}$ formulae introduced by Iosif et al. [IRV14]. For this fragment, the entailment problem can be solved by a reduction to the language inclusion problem Although, some complexity results results for $\texttt{SLF}_{RD}$ can be derived using the link between separation logic and DSGs, local $\texttt{SLF}_{MSO}$ is the only fragment with a known upper bound: The entailment problem for local $\texttt{SLF}_{MSO}$ environments is ExpTime-complete. We applied our results to show that the question whether a hypergraph language is definable by a local $\texttt{SLF}_{MSO}$ environment is decidable (see Section 6.4). According to [IRV14], this has been an open problem before. Furthermore, we provided an alternative syntactic fragment of $\texttt{SLF}_{RD}$ which captures exactly the hypergraph languages realizable by tree-like data structure grammars. We completed our results with an analysis of the relationships between MSO_2L, $\texttt{SL}_{RD}$, $\mathcal{HRL}$ and their subclasses with respect to their expressiveness.

7.2. Future Work

Unfortunately, tree-like HRGs do not capture the full intersection of monadic second-order logic and hypergraph languages $MSO_2L \cap \mathcal{HRL}$ and research by Engelfriet and Courcelle [CE12] indicates that no proper syntactic restriction of HRGs exists to capture this. Thus, no definition of regular hypergraph languages is currently known although it is known that the intersection $MSO_2L \cap \mathcal{HRL}$ has similar properties as regular string and tree languages. It is an open question, whether this also holds for the subclass of connected hypergraph languages. Furthermore, it is unknown whether an HR-language in $MSO_2L \cap \mathcal{HRL}$ consisting only of connected graphs exists which is not realizable by a tree-like HRG. Due to the similarity between linear context-free string grammars and tree-like HRGs, we conjecture that this is not the case. One of the difficulties in showing this is the lack of a construction to compute the intersection of two HRGs. By results of Habel [Hab92] and Courcelle [Cou91], there exists an HRG realizing the intersection of two tree-like HRGs, because every tree-like HRG can be translated into an MSO_2F formula and $\mathcal{HRL}$ is closed under intersection with MSO_2F definable languages. However, no effective algorithm to compute this HRG is known given an arbitrary HRG and an arbitrary MSO_2F sentence. In addition to that, there remain some open questions regarding the complexity of the entailment and inclusion problem for $\texttt{SLF}_{MSO}$ and tree-like HRGs, respectively. For instance, no upper bound to check the entailment problem of $\texttt{SLF}_{MSO}$ or $\texttt{SLF}_{tree-like}$ formulae is known. A reduction of $\texttt{SLF}_{MSO}$ to MSO_2F as done to prove decidability of the entailment problem gives no further insight here, because checking satisfiability for MSO_2F is non-elementary. Finally, fragments of HRGs which can also realize disconnected graphs might be studied with respect to decidability of the inclusion problem. One approach might be based on the following conjecture by Courcelle [CE12]:

> Every hypergraph language of bounded tree width is recognizable if and only if it is definable in counting monadic second-order logic.

Counting monadic second-order logic extends MSO_2F by (infinitely many) new operators $card_{p,q}(X)$ which assert that the size of a set X equals p modulo q. Lapoire proposes in [Lap98] that this conjecture is true, but the full proof is missing and the conjecture is still considered open in [CE12].

Bibliography

[AD09] Timos Antonopoulos and Anuj Dawar. Separating Graph Logic from MSO. In *Foundations of Software Science and Computational Structures*, pages 63–77. Springer, 2009.

[AGH$^+$14] Timos Antonopoulos, Nikos Gorogiannis, Christoph Haase, Max Kanovich, and Joël Ouaknine. Foundations for Decision Problems in Separation Logic with General Inductive Predicates. In *Foundations of Software Science and Computation Structures*, pages 411–425. Springer, 2014.

[AU71] Alfred V. Aho and Jeffrey D. Ullman. Translations on a Context Free Grammar. *Information and Control*, 19(5):439–475, 1971.

[BCC$^+$07] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape Analysis for Composite Data Structures. In *Computer Aided Verification*, pages 178–192. Springer, 2007.

[BCO06] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Formal Methods for Components and Objects*, pages 115–137. Springer, 2006.

[BFPG14] James Brotherston, Carsten Fuhs, Juan A. Navarro Pérez, and Nikos Gorogiannis. A Decision Procedure for Satisfiability in Separation Logic with Inductive Predicates. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, pages 25:1–25:10, New York, NY, USA, 2014. ACM.

[BK$^+$08] Christel Baier, Joost-Pieter Katoen, et al. *Principles of Model Checking*, volume 26202649. MIT press Cambridge, 2008.

[Büc60] J. Richard Büchi. Weak Second-Order Arithmetic and Finite Automata. *Mathematical Logic Quarterly*, 6(1-6):66–92, 1960.

[CD11] Cristiano Calcagno and Dino Distefano. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods*, pages 459–465. Springer, 2011.

[CDG$^+$07] Hubert Comon, Max Dauchet, Rémi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree Automata Techniques and Applications. 2007.

[CE12] Bruno Courcelle and Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach.* Number 138. Cambridge University Press, 2012.

[Cou90] Bruno Courcelle. The Monadic Second-Order Logic of Graphs I: Recognizable Sets of Finite Graphs. *Information and computation*, 85(1):12–75, 1990.

[Cou91] Bruno Courcelle. The Monadic Second-Order Logic of Graphs V: On Closing the Gap between Definability and Recognizability. *Theoretical Computer Science*, 80(2):153–202, 1991.

[CR08] Bor-Yuh Evan Chang and Xavier Rival. Relational Inductive Shape Analysis. In *Symposium on Principles of Programming Languages*, volume 43, pages 247–260. ACM, 2008.

[Dod08] Mike Dodds. From Separation Logic to Hyperedge Replacement and Back. In *Graph Transformations*, pages 484–486. Springer, 2008.

[Don70] John Doner. Tree Acceptors and some of their Applications. *Journal of Computer and System Sciences*, 4(5):406–451, 1970.

[DP08] Dino Distefano and Matthew J. Parkinson. jStar: Towards Practical Verification for Java. In *OOPSLA*, pages 213–226. ACM, 2008.

[DPV11] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures using Separation Logic. In *Computer Aided Verification*, pages 372–378. Springer, 2011.

[EFE95] Heinz-Dieter Ebbinghaus, Jörg Flum, and Hans-Dieter Ebbinghaus. *Finite Model Theory*, volume 2. Springer, 1995.

[Ehr79] Hartmut Ehrig. Introduction to the Algebraic Theory of Graph Grammars (A Survey). In *Graph-Grammars and Their Application to Computer Science and Biology*, pages 1–69. Springer, 1979.

[Eng91] Joost Engelfriet. A Regular Characterization of Graph Languages Definable in Monadic Second-Order Logic. *Theoretical Computer Science*, 88(1):139–150, 1991.

[Hab92] Annegret Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643. Springer, 1992.

[HJJ$^+$95] Jesper G Henriksen, Jakob Jensen, Michael Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. *Mona: Monadic Second-Order Logic in Practice.* Springer, 1995.

[Hoa69] Charles Antony Richard Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[IRS13] Radu Iosif, Adam Rogalewicz, and Jiri Simacek. The Tree Width of Separation Logic with Recursive Definitions. In *Automated Deduction–CADE-24*, pages 21–38. Springer, 2013.

[IRV14] Radu Iosif, Adam Rogalewicz, and Tomas Vojnar. Deciding Entailments in Inductive Separation Logic with Tree Automata. *arXiv preprint arXiv:1402.2127*, 2014.

[ISO12] Working Draft, Standard for Programming Language C++, 2012.

[JGN14] Christina Jansen, Florian Göbe, and Thomas Noll. Generating Inductive Predicates for Symbolic Execution of Pointer-Manipulating Programs. Technical Report AIB-2014-08, RWTH Aachen University, may 2014.

[KS94] Nils Klarlund and Michael I. Schwartzbach. Graphs and Decidable Transductions based on Edge Constraints. *DAIMI Report Series*, 23(469), 1994.

[Lap98] Denis Lapoire. Recognizability Equals Monadic Second-Order Definability for Sets of Graphs of Bounded Tree-Width. In *STACS 98*, pages 618–628. Springer, 1998.

[Lau91] Clemens Lautemann. Tree Automata, Tree Decomposition and Hyperedge Replacement. In *Graph Grammars and Their Application to Computer Science*, pages 520–537. Springer, 1991.

[Mey74] Albert R. Meyer. The Inherent Computational Complexity of Theories of Ordered Sets. In *Proceedings of the International Congress of Mathematicians*, volume 2, page 481, 1974.

[MTLT08] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. THOR: A Tool for Reasoning about Shape and Arithmetic. In *Computer Aided Verification*, pages 428–432. Springer, 2008.

[NS00] Frank Neven and Thomas Schwentick. On the Power of Tree-Walking Automata. In *Automata, Languages and Programming*, pages 547–560. Springer, 2000.

[O'H12] Peter W. O'Hearn. A Primer on Separation Logic (and Automatic Program Verification and Analysis). *Software Safety and Security: Tools for Analysis and Verification*, 33:286, 2012.

[Rey00] John C. Reynolds. Intuitionistic Reasoning about Shared Mutable Data Structure. *Millennial perspectives in computer science*, 2(1):303–321, 2000.

[Rey02] John 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. IEEE, 2002.

[RS84] Neil Robertson and Paul D. Seymour. Graph Minors III: Planar Tree-Width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

[Sei90] Helmut Seidl. Deciding Equivalence of Finite Tree Automata. *SIAM Journal on Computing*, 19(3):424–437, 1990.

[SR97] A Salomaa and G Rozenberg. Handbook of Formal Languages. *Vol. III: Beyond Words*, 1997.

[TW68] James W. Thatcher and Jesse B. Wright. Generalized Finite Automata Theory with an Application to a Decision Problem of Second-Order Logic. *Mathematical systems theory*, 2(1):57–81, 1968.

[VLC10] Jules Villard, Étienne Lozes, and Cristiano Calcagno. Tracking Heaps that Hop with Heap-Hop. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 275–279. Springer, 2010.

APPENDIX A

Notation

A.1. Classes of Graphs and Languages

For logical formalisms, we use the suffix F to denote the class of formulae and the suffix L to denote the class of language that can be defined by these formulae.

Table A.1.: classes of hypergraphs and languages

Notation	Explanation	Page
$A^\star$	all finite words over the alphabet A	7
$PG_{A,B}$	all plain graphs over the alphabets A and B	8
$Trees_A$	all trees over the alphabet A	8
HG_A	all hypergraphs over the alphabet A	12
$\lceil \mathcal{H} \rceil$	all hypergraphs isomorphic to $\mathcal{H}$	14
$\mathcal{HRG}$, $\mathcal{HRL}$	all hyperedge replacement grammars and all HR-languages realizable by HRGs	16
$\mathfrak{T}_{\mathfrak{G}}(\xi)$	all (path-connected) derivation trees of the HRG $\mathfrak{G}$ with initial nonterminal ξ	19
HC_A	all heap configurations over the alphabet A	21
$\mathcal{DSG}$, $\mathcal{DSL}$	all data structure grammars and all hypergraph languages realizable by data structure grammars	21

continued on next page

Table A.1 – continued from previous page

Notation	Explanation	Page
$\mathfrak{G}[S]$	all structures over the signature S	24
$MSOF[S]$	all MSO formulae over the signature S	24
$MSOL[S, C]$	all languages $L \subseteq C$ that can be defined in $MSOF[S]$	26
MSO_1F, MSO_1L	all MSO formulae over hypergraph models using the adjacency matrix approach and all hypergraph languages definable in MSO_1F	29
MSO_2F, MSO_2L	all MSO formulae over hypergraph models as defined in Definition 4.1.4 and all hypergraph languages definable in MSO_2F	29
$\mathtt{SSLF}$	all simple separation logic formulae	32
$\mathtt{SLF}_{RD}$, $\mathtt{SL}_{RD}$	all separation logic formulae with recursive definitions and all languages definable in $\mathtt{SLF}_{RD}$	33
$\mathtt{SLF}_{HR}$, $\mathtt{SL}_{HR}$	all separation logic formulae that can be translated to data structure grammars and all HR-languages definable by an $\mathtt{SLF}_{HR}$ formula	36
THC_A	all tagged heap configurations over the alphabet A	37
$HRA^\star$	all hypergraphs that can be generated by an hyperedge replacement algebra	44
$\mathcal{THRG}$, $\mathcal{THRL}$	all tree-like HRGs and all HR-languages realizable by tree-like HRGs	52
$\mathcal{TDSG}$, $\mathcal{TDSL}$	all tree-like data structure grammars and all HR-languages realizable by tree-like DSGs	52
$\mathtt{SLF}_{MSO}$, $\mathtt{SL}_{MSO}$	all formulae of the separation logic fragment of Iosif et al. [IRS13] and all hypergraph languages definable in $\mathtt{SL}_{MSO}$	85
$\mathcal{DTHRG}$, $\mathcal{DTHRL}$	all directed tree-like HRGs and all HR-languages realizable by directed tree-like HRGs	88

continued on next page

Table A.1 – continued from previous page

Notation	**Explanation**	**Page**
$\mathcal{DTDSG}$, $\mathcal{DTDSL}$	all directed tree-like data structure grammars and all HR-languages realizable by directed tree-like DSGs	88
local $\texttt{SLF}_{MSO}$, local $\texttt{SL}_{MSO}$	all formulae of the separation logic fragment of Iosif et al. that can be translated into regular tree automata [IRV14] and all hypergraph languages definable in $\texttt{SLF}_{tree-like}$	93
$\texttt{SLF}_{tree-like}$, $\texttt{SL}_{tree-like}$	all tree-like separation logic formulae and all HR-languages definable in $\texttt{SLF}_{tree-like}$	95

A.2. Symbols

The following table collects the most frequently used symbols in this thesis.

Table A.2.: symbols

Notation	**Explanation**	**Page**
$u, v, w, x, ...$	single objects representing vertices, edges, variables etc.	7
$A, B, C, ...$	(finite) alphabets	7
$\mathcal{A}, \mathcal{B}, \mathcal{H}, \mathcal{K}, ...$	graphs, hypergraphs, structures	7
$\mathfrak{G}, \mathfrak{G}', ...$	context-free string grammars, regular tree grammars, HRGs, DSGs	7
$\mathfrak{A}, \mathfrak{B}, ...$	finite (tree) automata	7
$\mathbb{N}$	the set of natural numbers	7
$f, g, h, \ell, \jmath, ...$	(partial) functions	8
	the null reference	8

continued on next page

Table A.2 – continued from previous page

Notation	Explanation	Page
`Loc`	the set of all heap locations	8
`Val`	the set of all heap values	8
`Heaps`	the set of all heaps	9
Ⓢ	the set of all selectors	9
`Obj`(h)	the set of locations pointing to objects in a heap h	9
$\alpha, \beta, \gamma, ...$	nonterminal symbols	14
N, T	the set of nonterminal and terminal symbols	14
L, $L(\mathfrak{A})$, $L(\mathfrak{G})$, $L(\mathfrak{G}, \alpha)$	languages, languages realized by an automaton $\mathfrak{A}$, a string grammar $\mathfrak{G}$, an HRG with initial nonterminal α	14
$\mathfrak{a}, \mathfrak{b}, \mathfrak{c}, ...$	production rules of an HRG	15
$\xi^\bullet$	a handle corresponding to a nonterminal symbol ξ	16
S	a signature, i.e. a set of relational symbols	24
$\rho, \sigma, ...$	predicate names	34
Γ	an `SLF` environment	34
τ	placeholder to represent “no label”	65