Verifying Java Programs
A Graph Grammar Approach

Jonathan Heinen

Verifying Java Programs
A Graph Grammar Approach

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
RWTH Aachen University
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker

Jonathan Heinen

aus

Eschweiler

Berichter: Universitätsprofessor Dr. Ir. Joost-Pieter Katoen
Universitätsprofessor Dr. Ir. Arend Rensink

Tag der mündlichen Prüfung: 9. März 2015

Abstract

Object-oriented programming languages enforce some new challenges to the automated analysis and verification of programs as objects can be created, manipulated and newly linked at runtime. A main difficulty is enforced by the dynamic creation of objects as this imposes (potentially) infinite many possible program-states. However, classical verification approaches such as model checking depend on finite state spaces. It is common practice to use abstraction in order to transform an infinite state space into a finite abstract one.

In this thesis we introduce an abstraction approach that is based on hypergraphs and hyperedge replacement grammars. We use hypergraphs to naturally model abstracted heap structures. Nonterminals are used to abstractly represent subgraphs of *well-defined structures* (e.g. subtrees) by a single edge. The *well-defined structures* of a subgraph are defined via a hyperedge replacement grammar. Derivations of hyperedge replacement grammars are defined via edge replacements, where single nonterminal edges are replaced by subgraphs. We use those replacements in the classical forward direction to concretise abstracted parts of the heap as well as in a backward manner to partially abstract the heap. It is a characteristic of our approach that abstracted heaps are composed from abstract as well as concrete parts.

Starting at a given (abstract) start state we generate the reachable abstract states of a program by executing the statements of the program on the abstract states. The execution is realised in three steps. In a first step we concretise those parts of the heap that are involved in the execution. In a second step the statement is executed on concrete parts of the heap. In the third and last step the heap is abstracted as far as possible.

In the generated state space each state is represented by an abstract hypergraph which potentially represents infinitely many concrete states of the systems. We aim at describing and verifying properties of those represented concrete states. The *Monadic Second Order logic*(MSO) provides a possibility to describe various properties over graphs. Among others simple properties such as reachability or cycle-freeness as well as counter properties such as minimal and maximal lengths or that a list contains an even number of elements. The power of the logic, however, involves high verification costs. We are not aware of any efficient algorithm from the literature that could be used here. In this thesis we develop a new approach for the verification of MSO properties on abstract graphs and provide some experimental results that hint to its efficiency.

For the analysis of temporal properties classical verification techniques such as model checking can be used as the considered abstract state spaces are finite. However, analysing object oriented languages we are not just interested in properties of separated states but we want to describe the dynamic behaviour of objects over time. Unfortunately, the objects are not visible for classical model checking techniques. We extend the logics typically used for model checking by quantification over objects. Among others we

consider the logic *quantified computational tree logic* for which we develop an on-the-fly model checking algorithm.

The framework presented in this thesis can be used for any object oriented language. In this thesis we focus on Java bytecode for which we present various adaption considering Java-specific properties such as its object model, the method stack or local and global variables.

We implemented a prototype tool called Juggrnaut for the analysis and verification of Java bytecode that is based on the presented techniques and algorithms. Given a Java program, an abstraction grammar as well as a start state that includes the program input the tool generates the abstract state space that then can be used for further analysis. E.g. our tool supports the verification of state properties in *Monadic Second Order logic*. We present various experimental results that we generated by the tool and that are quite encouraging.

Zusammenfassung

Objektorientierte Programmiersprachen bringen einige Herausforderungen für die automatische Analyse und Verifikation mit sich. Diese ergeben sich daraus, dass Objekte zur Laufzeit generiert, manipuliert und neu verknüpft werden können. Insbesondere die Möglichkeit dynamisch unbeschränkt viele neue Objekte zu generieren stellt ein Problem dar, da dies zu einer unendlichen Anzahl von möglichen Programmzuständen führt. Klassische Verifikationsverfahren wie z.B. das Model Checking setzen jedoch einen endlichen Zustandsraum voraus. Die Nutzung von Abstraktion, um unendliche Zustandsräume in endliche, *abstrakte* Zustandsräume zu überführen, ist ein oft genutzter Lösungsansatz um Verifikation in diesen Fällen zu ermöglichen.

In dieser Arbeit stellen wir ein Abstraktionsverfahren vor, das auf der Verwendung von Hypergraphen und Hyperkantenersetzungsgrammatiken basiert. Wir nutzen Hypergraphen um abstrahierte Heapstrukturen auf natürliche Weise darzustellen. Hierbei nutzen wir Nichtterminalkanten um ganze Teilgraphen von *vordefinierter Struktur* (z.B. Teilbäume) durch nur eine Kante abstrakt zu beschreiben. Die *vordefinierte Struktur* eines abstrahierten Teilgraphen beschreiben wir mit Hilfe von Hyperkantenersetzungsgrammatiken. Ableitungsschritte von Hyperkantenersetzungsgrammatiken werden durch Kantenersetzung realisiert, wobei einzelne Nichtterminalkanten durch Teilgraphen ersetzt werden. Diese nutzen wir in klassischer Vorwärtsrichtung um abstrahierte Teile des Heaps zu konkretisieren sowie in rückwärtiger Richtung um Teile des Heaps zu abstrahieren. Eine Besonderheit unseres Abstraktionsverfahrens liegt darin, dass ein abstrakter Heap sowohl aus konkreten als auch abstrakten Teilbereichen besteht.

Wir bestimmen alle erreichbaren, abstrakten Zustände eines Programms indem wir, ausgehend von einer (abstrakten) Startkonfiguration, die Anweisungen des Programms auf den abstrakten Zuständen ausführen. Hierzu konkretisieren wir in einem ersten Schritt all die Bereiche des Heaps, die in die Ausführung der Anweisung involviert sind um dann in einem zweiten Schritt die Anweisung, die nun lediglich auf konkret vorliegende Teilbereiche Einfluss nimmt, auszuführen. In einem letzten, dritten Schritt abstrahieren wir dann den Heap wiederum soweit wie möglich.

In den generierten Zustandsräumen wird jeder mögliche Zustand durch einen abstrakten Hypergraphen dargestellt, der potentiell unendlich viele konkrete Zustände des Systems repräsentiert. Es stellt sich die Frage, wie wir diese Zustände auf bestimmte Eigenschaften überprüfen können. Mit Hilfe der *Monadischen Logik zweiter Stufe über Graphen* lassen sich zahlreiche Eigenschaften beschreiben: von einfachen Eigenschaften wie Erreichbarkeit oder Kreisfreiheit bis hin zu verschiedenen Zähleigenschaften wie minimale oder maximale Längen oder dass eine Liste eine gerade Anzahl an Elementen besitzt. Die Mächtigkeit der Logik bringt aber auch sehr hohe Kosten für die Überprüfung mit sich. Aus der Literatur sind uns keine effizienten Verfahren bekannt, die sich hierfür nutzen lassen. In dieser Arbeit entwickeln wir einen neuen Ansatz zur Überprüfung dieser Eigschenschaften über abstrakten Graphen und zeigen, dass dieser für verschiedenste Eigenschaften effizient arbeitet.

Da die betrachteten abstrakten Zustandsräume endlich sind, können wir für die Analyse temporaler Eigenschaften erprobte Verfahren wie das klassische Model Checking nutzen. Bei der Analyse objektorientierter Sprachen interessieren wir uns jedoch nicht nur für isolierte Eigenschaften der einzelnen Zustände, sondern insbesondere auch für das dynamische Verhalten der einzelnen Objekte des Heaps über die Zeit. Diese jedoch sind für die klassischen Model Checking Verfahren nicht sichtbar. In dieser Arbeit erweitern wir die klassischerweise für Model Checking genutzten Logiken um Quantoren über Objekten. Dies führt unter anderem zur Logik *quantifed computational tree logic* für die wir einen on-the-fly Model Checking Algorithmus entwickeln.

Das in dieser Arbeit vorgestellte Framework lässt sich für sämtliche objektorientierten Sprachen nutzen. Wir beschäftigen uns intensiver mit der Analyse von Java Bytecode und stellen verschiedene Anpassungen vor, die Java-spezifische Eigenschaften modellieren, so z.B. die Art der Repräsentation von Objekten, den Methodenstack oder lokale, und globale Variablen.

Auf Basis der vorgestellten Techniken und Verfahren wurde das prototypische Tool Juggrnaut realisiert, das die Analyse und Verifikation von Java Bytecode Programmen ermöglicht. Zu einem gegebenem Java Programm, einer Abstraktionsgrammatik sowie einem Startzustand, der unter anderem die Eingabe des Programms enthält, generiert es einen abstrakten Zustandsraum, der dann für weitergehende Analysen genutzt wird. Unser Tool unterstützt z.B. die Überprüfung von Zustandseigenschaften in der *Monadischen Logik zweiter Stufe*. Wir präsentieren mehrere experimentelle Ergebnisse, die wir mit Hilfe des Tools erhalten haben und die uns vielversprechende Ergebnisse liefern.

Acknowledgments

In 2007 when I started at the MOVES group, I had the great fortune to freely choose my research topic. The verification of pointer programs immediately caught my attention. Thomas Noll and Stefan Rieger were already working on that topic and they had achieved promising results on the analysis of programs on lists. I am deeply grateful to them for involving me into their research. There is a lot that I learned in the past seven years and I really enjoyed my time at the MOVES group. If I got the chance to decide for a research topic today my decision would definitely be the same.

I want to thank Joost-Pieter Katoen for giving me this unique opportunity and for supporting our ideas and research. But most of all I want to thank him for being an excellent boss and for his almost infinite patience. I also want to express my gratitude to Thomas Noll for his guidance on research and teaching topics. Further acknowledgments go to Barbara König and her group for receiving us several times in Duisburg and for inspiring discussions on various graph related topics, and to Arend Rensink for being the second supervisor of my thesis.

I very much have to mention everyone from i2, lufgi2 and the hybrid systems research group for providing me a perfect working environment. Special thanks go to Daniel Klink, who always supported my conservatism, paid attention to my thoughts and ideas and who proposed the name Juggrnaut for our verification tool. But most of all I have to thank my colleague and coauthor Christina Jansen for an inspiring and exciting time of research and for her substantial comments and thoughts regarding my publications and this thesis.

Finally, I want to thank all the students that I got in contact with during my work. It is also their credit that my time at the chair remains unforgotten to me. Three students should be explicitly mentioned here: Henrik Barthels, who developed *automata based techniques for the detection of hypergraph embedding*, essential to the efficiency of our state space generation, Max Görtz, with whom I developed the basic technique for *checking MSO properties against hyperedge replacement grammars*, and Tobias Hoffmann, who wrote his thesis about *model checking quantified linear temporal logic*.

Contents

1 Introduction **1**

1.1 The Lindstrom Traversal Algorithm . . . 2

1.1.1 Understanding Lindstrom's Algorithm . . . 3

1.1.2 Properties of the Lindstrom Algorithm . . . 3

1.2 Our Approach in a Nutshell . . . 5

1.3 Structure of this Dissertation . . . 8

1.4 Related Work . . . 10

2 Preliminaries **15**

2.1 Notations . . . 15

2.2 Hypergraphs . . . 16

2.3 Hyperedge Replacement Grammars . . . 18

2.4 Backward Rule Application . . . 24

2.5 Inherent Properties . . . 26

2.6 Properties of HRGs . . . 28

2.7 Conclusions . . . 34

3 Abstract Heap Representation **37**

3.1 Heap Representation . . . 38

3.2 Heap Abstraction Grammars . . . 43

3.3 Example Heap Abstraction Grammars . . . 45

3.4 Local Concretisation . . . 49

3.4.1 Concretisation of Vertices . . . 49

3.4.2 Local Greibach Normal Form . . . 51

3.4.3 Concretisation Function . . . 60

3.5 Transition Systems . . . 62

3.6 Conclusions . . . 69

4 Object-Oriented Heaps **73**

4.1 Typed Objects . . . 74

4.2 Data Structure Grammars . . . 79

4.3 Concretisation and Abstraction . . . 87

4.3.1 Concretisation . . . 88

4.3.2 Abstraction . . . 88

4.4 Conclusions . . . 90

5 Abstract JVM **93**
5.1 The Java Virtual Machine 93
5.2 A Formal Definition of the JVM 94
5.2.1 Static environment of a JVM 95
5.2.2 State of a JVM 96
5.3 Modelling JVM States 97
5.3.1 Interfaces and *null* 97
5.3.2 Static Variables, Literals and Boolean Values 99
5.3.3 Complete Heap Representation 99
5.3.4 Method Stack 100
5.3.5 Terminal States 102
5.4 Java Bytecode Execution 103
5.4.1 Basic Modifier 104
5.4.2 Concrete Semantics 107
5.4.3 Abstract Semantics 117
5.4.4 Garbage Collection 121
5.5 Experiments: Lindstrom's Algorithm 121
5.6 Conclusions 126

6 Logics over Hypergraphs **127**
6.1 First Order Logic 128
6.1.1 Prenex Normal Form 130
6.2 FO and Graph Grammars 131
6.2.1 Grammar Adjustment 132
6.2.2 Evaluation Trees for Terminal Graphs 135
6.2.3 Evaluation Trees for Graphs in Context 140
6.2.4 Composition of Evaluation Trees 146
6.2.5 Evaluation 151
6.2.6 Minimisation 152
6.2.7 Pre-Evaluation 154
6.2.8 Evaluation for HRG-Languages 156
6.3 Monadic Second Order Logic 162
6.4 Experimental Results 170
6.5 MSO-Logic and JVM-States 171
6.5.1 Pointers Instead of Edges 172
6.5.2 Typed Vertices 172
6.5.3 Variables 173
6.5.4 MSOJ 173
6.5.5 Experiments: Lindstrom's algorithm 175
6.6 Conclusions 176

7 Quantified Model Checking **181**
7.1 Linear Temporal Logic 182
7.1.1 *LTL* Model Checking 186
7.2 Quantified LTL 192
7.2.1 *qLTL* Model Checking 194
7.3 Abstract Model Checking 197
7.3.1 LTL Model Checking 198
7.3.2 *qLTL* Model Checking 198
7.4 On The Fly Model Checking $qCTL^\star$ 202
7.5 Model Checking Lindstrom's Algorithm 203
7.6 Conclusions 204

8 Conclusion **205**
8.1 Achievements 206
8.2 Future Work 208

Index **211**

Bibliography **215**

Chapter 1
Introduction

Software plays an important role in todays life. We can find software based systems everywhere: in domestic appliances, cars, information systems, aircrafts, trains, etc. . With increasing calculation power the size and complexity of software increased constantly over the past decades. With growing size and complexity software becomes harder to maintain and more error-prone. At the same time software is used in security critical applications like cars, aircrafts and railway systems, where reliability of software is crucial. Therefore techniques for the analysis and verification of software are a pressing research topic. There is an increasing need of tool support for the automatic analysis and verification of code. For the verification of simple, imperative code, automatic techniques such as model checking [BK08] were proposed and successfully applied in a variety of areas.

Software engineering proposed several concepts to manage the growing complexity of software. One prominent concept is object orientation. Object-oriented languages such as Java, C++, C# and Objective-C are widely used. In these languages, program functionality is compacted into objects, which can be allocated at runtime. While this concept is quite successful, dynamic creation of objects imposes new challenges to verification techniques. The potentially unbounded state space induced by these programs prevents the use of standard verification technique such as model checking which (mostly) rely on finite state spaces. While the number of objects at least in practice is limited by the amount of available memory, we cannot limit the size of the structure the objects form on the heap for theoretical considerations. In practice, however, the generated structures are in general too big to consider them. A typical approach to handle big and also infinite state spaces is abstraction. This thesis presents *an abstraction technique for heap structures based on graph grammars*. We consider Java as programming language; however, most of the concepts presented here can easily be generalised and applied to other object-oriented languages.

We focus on the verification of structural properties, i.e. object relational properties of the heap, while we do not consider primitive data values (such as integer and floating point numbers) and arrays.

1.1 The Lindstrom Traversal Algorithm

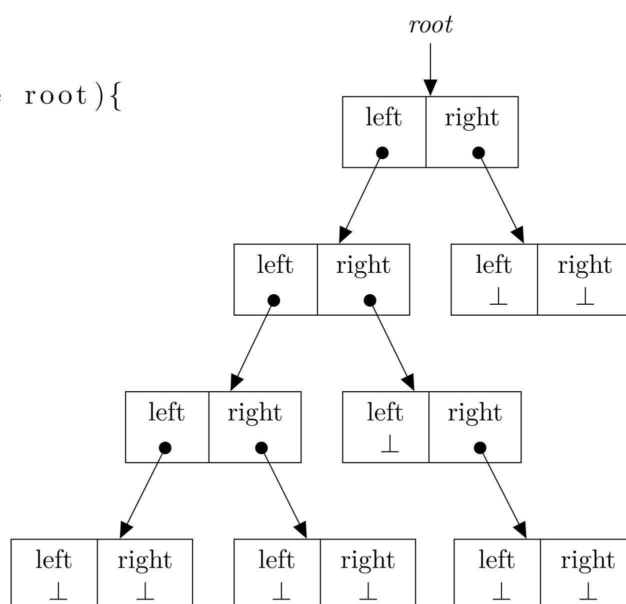
To get a better idea about our objectives, we present Lindstrom's algorithm which will accompany us through the presentation of the various techniques that form our framework for the analysis and verification of object-oriented programs. The Lindstrom algorithm [Lin73] is a variant of the Deutsch-Schorr-Waite traversal algorithm [SW67]. Whereas typical traversal algorithms require the use of a stack, Lindstrom's algorithm traverses trees without any stack (neither explicit nor implicit). Indeed the amount of required additional memory is constant. In Figure 1.1(a) the algorithm is given in form of a Java program. While Lindstrom's algorithm is known to work correctly for any directed acyclic graph, we consider trees only.

```
class Tree{
  Tree left , right;

  static void lindstrom(Tree root){
    Tree sen = new Tree();
    Tree prev = sen;
    Tree cur  = root;

    while(cur != sen){
      Tree next = cur.left;
      cur.left = cur.right;
      cur.right = prev;
      prev = cur;
      cur  = next;
      if (cur == null){
        cur  = prev;
        prev = null;
      }
} } }
```

(a) Lindstrom algorithm.



(b) A binary tree.

Figure 1.1: Tree traversal using Lindstrom's algorithm.

A Tree-object consists of two pointers left and right, used to refer to the corresponding children. In Figure 1.1(b) an example heap representing a valid tree input is given. Valid here refers to the absence of cycles and sharing. Each object is represented as a tuple of pointer variables, one for the left- and one for the right-pointer. Pointer values which are *null* are represented by ⊥, while non-*null* values are represented by directed edges pointing to the referenced object.

1.1.1 Understanding Lindstrom's Algorithm

The algorithm aims at traversing the given tree without any memoization (e.g. marking of visited nodes). To achieve this each tree node is visited three times. At each visit the node pointers are rotated counter clockwise, such that they are toggled between the nodes representing the left and right child and the parent. After the third rotation the original status of the pointers is restored.

To obtain a better understanding of the algorithm we consider the nine intermediate states of a traversal given in Figure 1.2, where we focus on the three visits of one of the nodes. Starting from the root node we traverse the tree until we reach the considered node pointed to by cur for the first time, that is the local state of the node was not changed yet (1). The upper subtree is partially traversed, the two lower trees are untraversed. The execution is at the beginning of the while-loop. The next node that will be visited is the left successor. Therefore we set the next-pointer to point to it (2). Afterwards we rotate the pointers of the node for the first time. First we set the left pointer to the right child (3), then we set the right pointer to the parent node, which is still referenced by the prev-pointer (4). Now that the first rotation is completed we continue with the left child. We first set the prev-pointer to the considered node and then the cur-pointer to the left child (5). Afterwards we start a new iteration of the while loop, this time on the left successor. After completing the traversal of the left subtree we return and cur points again to the considered node (6). The prev-pointer is still set to the left successor. Notice that the left-successor now is the original right child, which will be the next vertex to visit. Thus we set next to it and rotate the pointers a second time (7). Note that now the left-pointer is pointing to the parent node. Therefore when we return to the node, after traversing the right subtree, we set the next-pointer to the parent node (8) and do a last rotation on the pointers which restores the original local state (9). Note that now the whole subtree rooted at the considered node is traversed. We proceed in the partial traversed subtree to the top.

1.1.2 Properties of the Lindstrom Algorithm

We consider four elementary properties of Lindstrom's algorithm: *structure preservation*, *absence of null-pointer dereferences*, *termination*, and *completeness*. By completeness we refer to the fact that each node is visited at least once during the traversal, where we say that a node is visited if variable cur is pointing to it at least once during the traversal. Structure preservation means that starting with a tree as input also the resulting structure is a tree. In addition we can postulate that the original tree is retained after the traversal, i.e. that after the traversal for each node of the tree its left and right successors are the same as before the traversal.

The question is how to prove such properties. We propose a technique allowing us to verify these properties in an automated manner. The next section presents a short

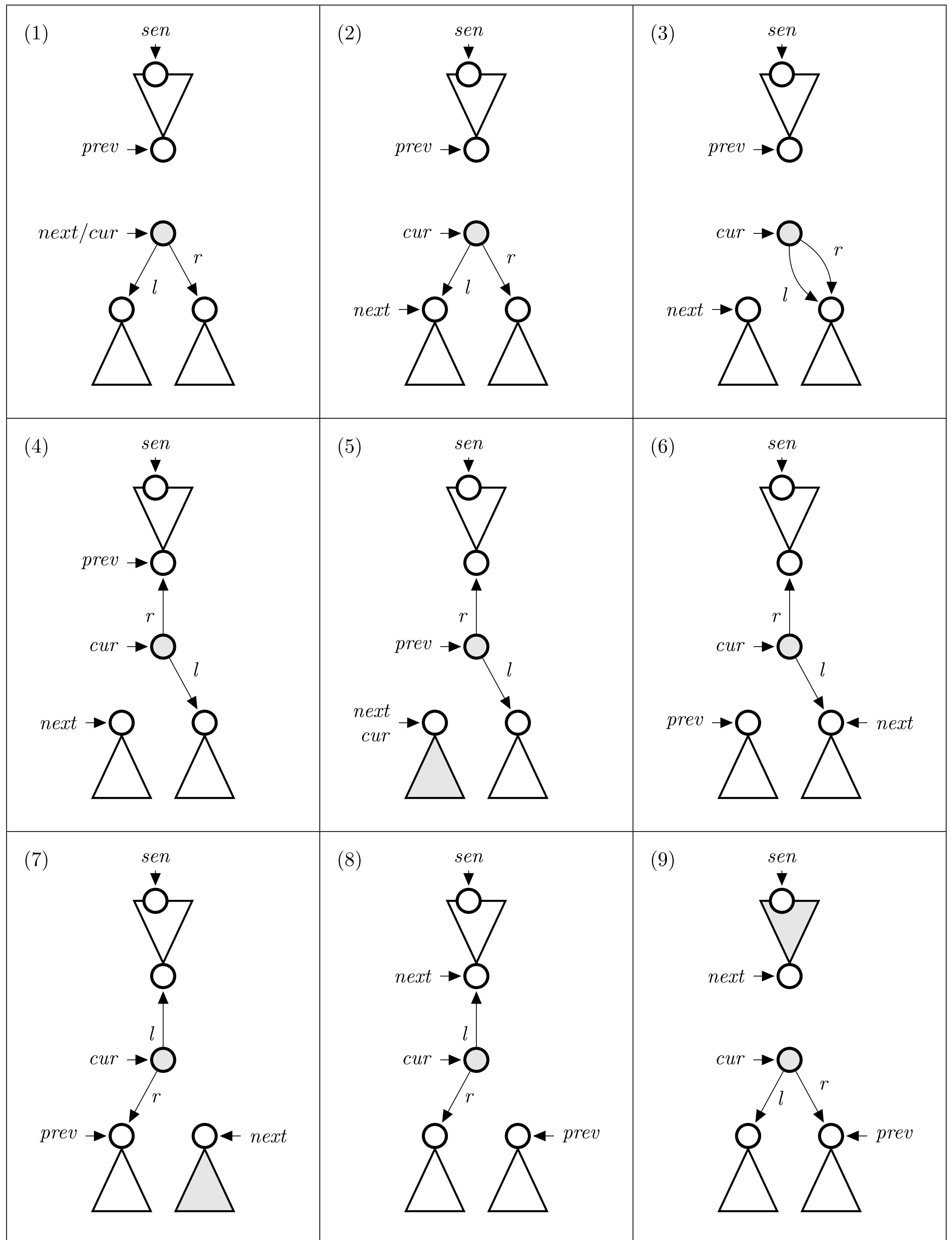


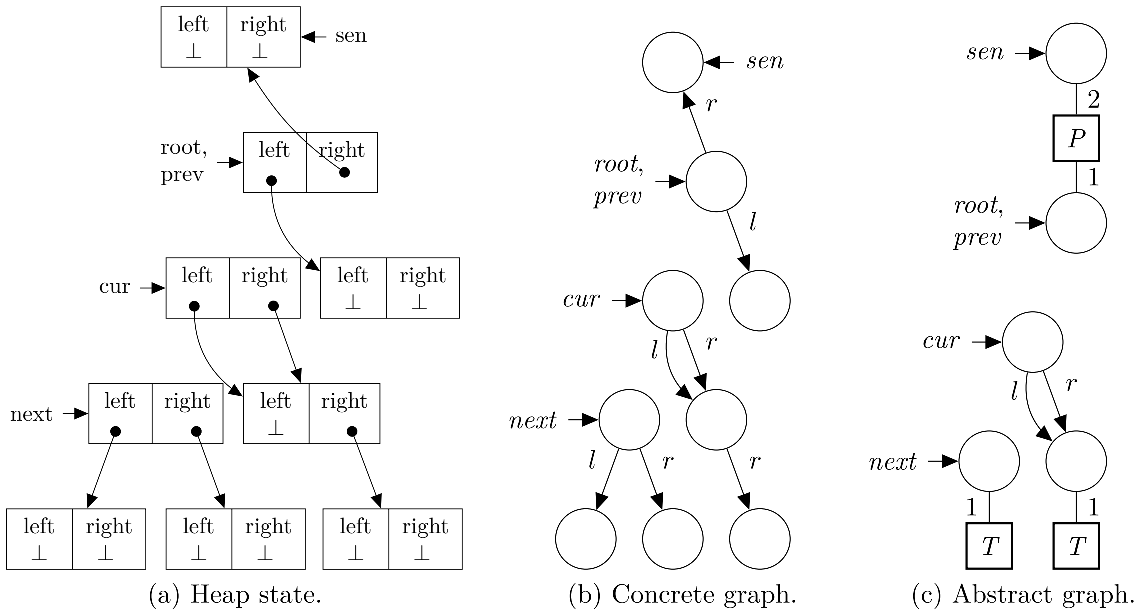
Figure 1.2: Snapshots of Lindstrom's algorithm with focus on one tree node.

overview presenting the key ideas behind our approach. The details will be investigated in detail in this thesis.

1.2 Our Approach in a Nutshell

Our approach is based on standard model checking. We use abstraction techniques to reduce potential infinite state spaces to a finite set of abstract representations as well as to describe an infinite number of inputs by a finite set of abstract ones.

A natural way to represent heaps are graphs. Note that in Figure 1.1(b) we already represented the heap in a graph-like manner where each object is represented as a composition of pointers, and pointer values as directed edges. If we use the represented heap state as input for Lindstrom's algorithm we get the heap depicted in Figure 1.3(a) after two iterations of the while loop. We can simplify the representation by introducing a labelling for edges that replaces the explicit representation of pointer variables. Doing so we get the representation depicted in Figure 1.3(b).



(a) Heap state. (b) Concrete graph. (c) Abstract graph.

Figure 1.3: Representation of heap states.

Note that this is still a concrete representation, containing all information given in the original representation and representing exactly one concrete heap state. The question is how to abstract such graph structures in order to represent infinitely many heaps by a single representation, e.g. all valid inputs for Lindstrom's algorithm, i.e. all binary trees.

Illustrating the algorithm in Figure 1.2 we already used an abstract representation, where we replaced subtrees not relevant for the discussed aspect by a triangular shape.

These placeholders then represented a partial, tree shaped part of the heap. However, we represented the abstracted part without defining formally what a tree shape is. In our abstraction approach we also use placeholders to abstractly represent entire parts of the heap. The placeholders we use are called *hyperedges* and we label them with nonterminals to differ between several shapes. Hyperedges are edges with arbitrarily many incident vertices. And we connect these to those vertices of the abstracted part that are shared with other (concrete or abstract) parts of the heap. In Figure 1.3(c) a graph with hyperedges is given. Hyperedges are depicted as rectangle boxes annotated with the label of the edge. The labels of the edges determine the possible shapes of the represented heap part. Figure 1.3(c) represents the heap given in Figure 1.3(a) and (b). Where edges labelled with T represent subtrees rooted in the attached vertex, while P-labelled edges represent heap parts of tree shape but beside the root we also distinct one of the leaves represented by the second attached vertex. Note that concrete and abstract parts coexist within one representation. That allows us to keep those parts of the heap concrete which are relevant for the current execution while abstracting the parts that have no influence at the current state of the execution.

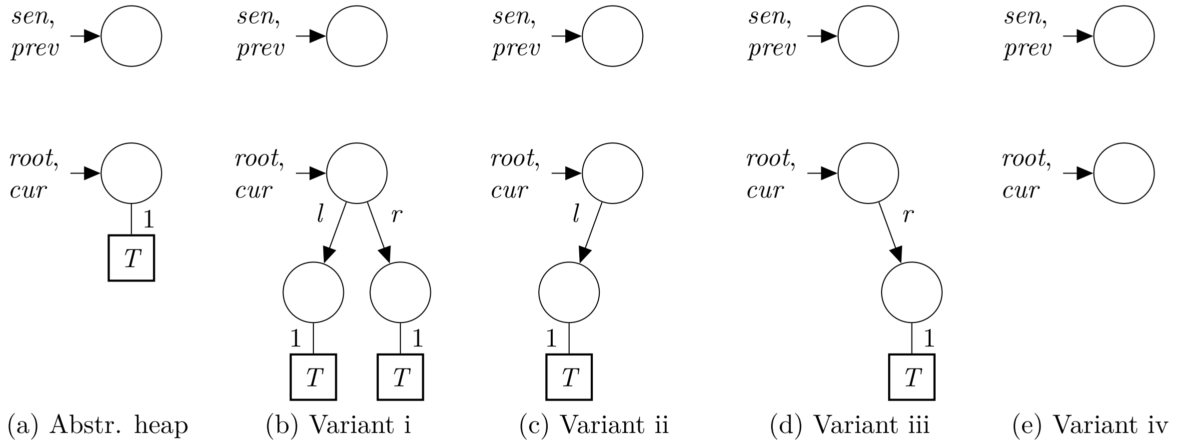


Figure 1.4: Concretisations of an abstract state.

State spaces are used to describe the behaviour of a system over time. In our case states are graphs each representing a single heap state. Transitions describe the order in which states occur. Note that classical object-oriented programming languages as (single threaded) Java are deterministic, therefore any execution results in a single chain of states. However, considering programs with infinitely many possible inputs results in potentially infinitely many chains. In order to handle such programs (e.g. Lindstrom's algorithm) we consider an abstract representation of the start state which covers all possible inputs. Reconsider Lindstrom's algorithm as well as the nonterminal T as introduced in Section 1.1. All valid inputs for the algorithm, i.e. all binary trees can be abstractly represented by the single graph consisting of a root vertex, attached to a single T-labelled edge.

While for concrete states the transition (successor) relation is determined by the semantics of Java it is not obvious how to determine the (potentially) infinite successors of an abstract state. Classically an abstract semantics is defined which determines an abstract representation of the successors. The abstract semantics relies on the executed statement as well as on the used abstraction, and in the worst case has to be constructed specifically for a considered problem. Instead of defining an abstract semantics for each use case, we determine the abstract semantics via combining the concrete semantics (of Java) with an abstraction and concretisation function. The concretisation function corresponds to derivations of hyperedge replacement grammars, i.e. applications of grammar rules, while the abstraction corresponds to the inverse, aka backward application of rules. The basic idea behind the combination is to ensure that whenever we execute a statement, those parts of the heap that are involved in the manipulations or have an impact to the control flow of the program are concretised beforehand.

In Figure 1.4(a) the abstract heap state is given that we reach after executing the part before the while loop. The while loop is executed as *cur* and *sen* point to different objects. Note that the information necessary to check the while guard is not abstracted, because all objects referred to by variables are represented concretely. The first statement of the loop sets the *next*-pointer to the left successor of the root-node. Note that this vertex is not represented concretely but is abstracted within the T-edge (the hyperedge labeled by T). The left successor could either be the root-node of a subtree (where we consider leaves to be trees as well) or *null*. The same holds for the right subtree. Therefore we get four possible concretisations as depicted in Figure 1.4(b)-(e). Note that the four representations together represent the same heaps as the original heap from Figure 1.4(a). Now that we have concretised the successors of *cur* we can determine its left successor and execute the statement for each of the four possible concretisations. After the execution of a statement the heap structure is abstracted as far as possible.

Figure 1.5 on page 8 depicts the idea of combining concretisation and abstraction with the concrete execution for the list traversal algorithm given in Figure 1.6 on page 9. Given a singly linked list of nodes connected via an n-pointer we traverse the list using a *pos* variable from the first to the last node. The list is represented abstractly by an L-labelled edge attached to the first and last node of the list. At the beginning *pos* points to the first node of the list (s_1). We need to concretise (C) the list in order to determine the successor of *pos*. We obtain two concretisations: one where the successor is the last element of the list (s_7) and another with an arbitrarily large list between the successor node and the last node (s_2). On both concretisations the statement can be executed (E), i.e. the variable *pos* is set to its successor. To complete the abstract execution of the statement the resulting graph is abstracted as far as possible (A). Using this combination of concretisation (C), execution (E) and abstraction (A) we generate the complete abstract transition system for list traversal as depicted in Figure 1.5.

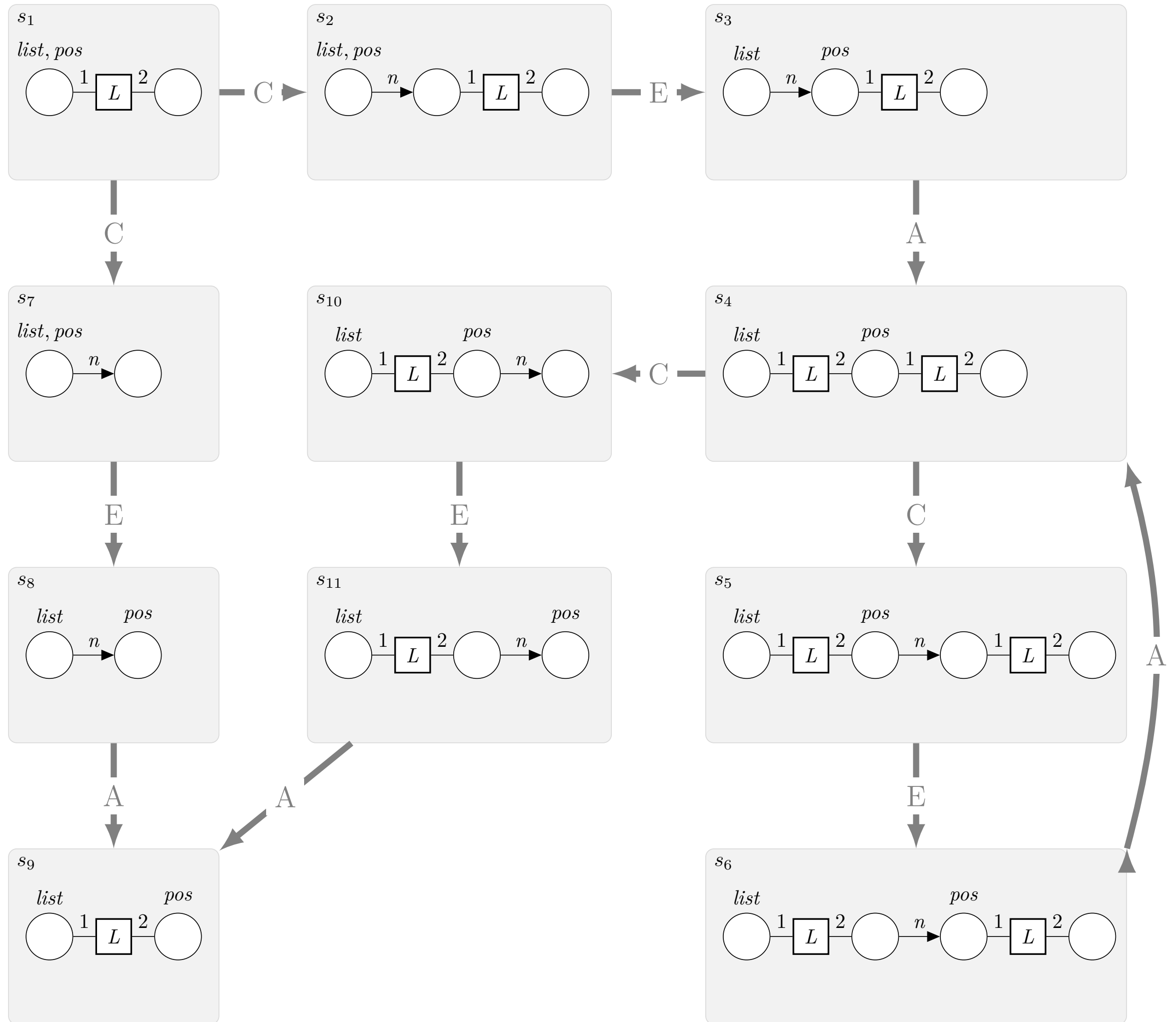


Figure 1.5: State space for a list traversal.

1.3 Structure of this Dissertation

This dissertation presents our approach for the verification of object-oriented programs. The approach is introduced for Java as programming language. As we will see in Chapter 5 we use Java bytecode to which Java programs are compiled, for our actual analysis, as the Java bytecode is easier to handle. The approach is based on the combination of classic model checking algorithms and an abstraction technique for heap structures. The dissertation is organised as follows:

In Chapter 2 we introduce some basic notation and concepts that we use in later chapters. Most of the chapter is dedicated to hypergraphs and hyperedge replacement grammars which we use to model heap states and on which our abstraction technique is based. We introduce the application of graph grammar rules that define the derivation steps of the hyperedge replacement grammars. Rule applications are considered in a forward as well

```
class List{
    List next;
    static void traverse(List list){
        List pos = list;
        while(pos.next != null) pos = pos.next;
}   }
```

Figure 1.6: A list traversal algorithm.

as in a backward manner. Forward application is used for concretisation while backward application yields abstraction. The remainder of Chapter 2 considers various properties of hyperedge replacement grammars that are essential for our approach and used by several proofs in later chapters.

In Chapter 3 we discuss how to model heap states by hypergraphs. We define some restrictions for hypergraphs and hyperedge replacement grammars in order to get a suitable representation of heap states. The rather arbitrary concretisation via rule application is developed towards a local concretisation that permits a directed, locally bounded concretisation of the heap. Local concretisation is based on a normal form for hyperedge replacement grammars, an extension of the well-known Greibach Normal Form for string grammars. At the end of Chapter 3, we introduce transition systems consisting of hypergraphs as states and a transition relation. We also define over-approximations of transition relations and –systems. Transition systems are used to describe the behaviour of a system, in our case the runs of an object-oriented program on a given input, while transition relations are used to describe the semantics of a language. In Chapter 5 we define the Java semantics in terms of a transition relation.

In Chapter 4 we extend the definition for hypergraphs to get a suitable model for typed heaps, where objects are associated with types inducing some useful restrictions to the connectivity of objects. These object types can be found in typed object-oriented languages such as Java. We call typed hypergraphs heap configurations and we use them to represent states of the Java virtual machine.

The state of a Java virtual machine is not defined by just the state of the heap but also by the values of static and local variables, the method stack and corresponding program counters. In Chapter 5 we extend the representation of heap configurations towards a representation of complete states of the Java virtual machine. Further we define the semantics of Java bytecode in form of graph transformations on these representations yielding a transition relation between JVM states. Based on the techniques introduced in Chapter 3 we extract an abstract semantics which gives us the ability to generate abstract transition systems for programs in Java bytecode. For Lindstrom's algorithm we give some experimental results.

In Chapter 6 we consider graph logics in order to express structural properties of program states. We introduce the well-known *first order logic* and *monadic second order logic*. As

we use hypergraphs to abstractly represent a set of concrete heap structures, we need to be able to check if all graphs defined via a hyperedge replacement grammar fulfil a logical formula. From the literature it is known that this problem is decidable [Cou90]. However, there is no efficient algorithm. We present our solution based on so-called evaluation trees, which are partial evaluations that can be combined corresponding to the derivations of a hyperedge replacement grammar. We introduce this technique and give an extension for states of a Java virtual machine as introduced in Chapter 5, where we also consider variables and object types of the state.

In Chapter 7 we then consider the actual model checking of temporal properties. Given an abstract state space of a Java program we can use classical linear temporal logic – or computational tree logic model checking to verify various properties of the program. Both logics are, however, not capable to describe the evolution of objects between states. E.g. given a reversal algorithm for lists we want to verify that the first list element before the reversal is the last element after the reversal. We extend both logics towards a quantified version, allowing us to quantify over the objects of a state. We present an on-the-fly model checking algorithm for these quantified logics and show how to use it with abstract state spaces.

Finally in Chapter 8 we will conclude and present some open research problems that should be considered in the future.

This dissertation is based on and extends earlier publications [HNR10; Jan+11; HBJ12; Hei+14]. The basic motivation for our approach was earlier work on the verification of programs with lists [NR08; DKR04; DRK02]. Attempting to extend these approaches to arbitrary data-structures directly lead us to the work of Sagiv *et. al.* on *shape analysis* [Bog+07; LRS06].

The rest of this chapter is devoted to related work in the area of analysis of heap manipulating programs and approaches related to graph languages.

1.4 Related Work

The analysis of object-oriented or more general heap manipulating programs has been a topic of continuous research interest in the last decades. Various approaches have been tailored or extended to pointer programs such as extensions of Hoare-style reasoning using e.g., separation logic [Rey02], abstract reachability analysis using regular model checking [Bou+06], shape analysis using three-valued logic [SRW02], graph transformations [ZR11] and dedicated extensions of temporal logic approaches [DKR06].

As it is not our intention to consider and compare all alternative techniques, we focus on the three most relevant approaches that are inspiring and related to our approach.

Shape analysis

Shape analysis [SRW96] can be seen as an extension of the abstract-interpretation approach [CC77] towards programs operating on heaps. As in our approach starting from an abstract representation of the program state, all reachable states are explored but in contrast states are explored based on an abstract semantics acting on the abstract states. Extracting abstract semantics was considered in [LMS04]. The framework and some extensions are implemented in the tool TVLA [LS00; LMS04; Bog+07].

Abstraction is realised by summarising nodes (like in our abstraction) by properties expressed as predicates in three-valued logic where the value "don't care" is used so as to deal with loss of information in summary nodes. Typical predicates are shape properties like reachability, cycle membership, etc. As we will see in Chapter 6 most of these are derivable from our heap graph representation and are implicitly provided in our state space exploration. Whereas in shape analysis all nodes satisfying the same predicates are summarised, in our approach nodes that constitute a well-defined (fragment of a) data structure are collapsed. The framework of [SRW02] uses user-provided instrumentation predicates to improve the analysis, in particular to refine the abstract heap representation.

Our running example was also inspired by the work around the tool TVLA. The verification of Lindstrom's algorithm is a recurring case study and was considered in depth in one of their publications [LRS06]. In order to verify Lindstrom's algorithm 24 predicates were used. We need 12 grammar rules to describe the occurring heap shapes (see Section 5.5 on page 121). Beside the significantly smaller number, their predicates encode additional knowledge about the algorithm requiring a deeper understanding of how the algorithm works, while the grammar rules that we use just describe the tree data structure. For instance, the number of completed visits of each node are encoded into the predicates. Thus the algorithm can be verified by our approach with less insight about the algorithm itself and is therefore more automated. In addition we are able to verify the same properties in remarkable less time [HNR10; Bog+07].

In order to maintain the complexity of the analysis they use a manually adapted version of the algorithm, that ensures that the global shape is always a tree [LRS06]. Reconsider Figure 1.2(6) where the left and right successor of the node *cur* point to the same node, i.e. violates the sharing freeness of trees. In this state the tree structure is locally violated. As in shape analysis always the complete heap is described, additional predicates would be necessary to also handle these violations, which in turn leads to a significant increase of complexity. Therefore they added an additional temporal variable in order to avoid the sharing. As our approach is more robust against local validations we do not need this adaption.

In [LRS06] the necessary predicates to verify Lindstrom's algorithm for directed acyclic graphs (DAGs) are given. While they were not able to finish the experiments in reasonable time, the approach can handle such data structures (at least in theory). Our approach is

not able to handle arbitrary DAGs due to the fact that hyperedge replacement grammars can describe graph languages of bounded tree-width only [Sko97].

Graph transformations

To use graph transformation systems for software verification [ZR11] is another approach inspired by TVLA. Like in our case, states of a system are represented as graphs and transitions between states are determined by a set of graph transformation rules, which model the behaviour of the system. This approach has been implemented in the tool Groove [Ren03; Ren04].

While the approach and the tool can handle arbitrary kinds of graph transformations there is also a technique to model Java programs as transformation systems [RZ09]. The properties of the Java program, such as the flow graph, are modelled as graphs and then a given, fixed set of graph transformations is applied which models the Java semantics. Note that this approach considers the actual Java programs while we use the compiled Java bytecode.

Like in our approach, the verification is realised via classical model checking techniques for *LTL* and *CTL* [KR06]. As model checking is based on enumerating all reachable states it is crucial that the considered model is finite. Infinite state spaces need to be abstracted to a finite set of abstract ones. For their approach they use two different abstraction techniques: *neighbourhood abstraction* [Bau+08] and *pattern-based graph abstraction* [RZ12].

In **neighbourhood abstraction** nodes are distinguished based on their neighbour nodes. Nodes with equal neighbourhood are aggregated within a single representant. Each summary node is associated with a bounded cardinality. The abstraction can be refined by adjusting the bound of cardinalities as well as by the radius of the neighbourhood considered. Note that nodes from different parts of the heap are merged, such that information about their connectivity and reachability is lost. Neighbourhood abstraction was implemented into the Groove tool [RZ10]. Tools performing state space exploration heavily depend on testing equality of states, i.e. isomorphism of states. For abstract graph transformation systems it is more suitable to consider *state subsumption* instead [ZR12] where states which behaviour is completely represented by other states are omitted and represented by those more general representants.

In **pattern-based graph abstraction**, graphs are represented by a form of derivation trees where each graph is build from a set of patterns. Graph transformations are given as transformations of derivation trees. Derivation trees are graphs where nodes are labelled by patterns and are abstracted in a manner similar to neighbourhood abstraction. Nodes are summarised if they share the same pattern as well as equal neighbour hood. To our knowledge, the pattern-based abstraction was not implemented so far.

As mentioned before, temporal properties can be verified via standard CTL and LTL model checking [KR06]. However, the known temporal logics are not suitable to express

the dynamic behaviour of objects over time. Therefore we develop the extension *qLTL* and *qCTL* in Chapter 7 that allows us to express properties involving the dynamic behaviour of objects by quantifying over object identities. Rensink proposed a similar approach for *qCTL* (he uses the notation QCTL) [Ren06]. While our approach extends the *on-the-fly* $CTL^\star$ model checking algorithm based on tableaux from Grumberg *et. al.* [BCG95] Rensink reduces the model checking problem for *qCTL* to the model checking problem for *CTL*. In order to model check a qCTL property the formula is skolemized, i.e. the quantified variables are turned into new model constants which are nondeterministically assigned during the evaluation of the formula. In order to realise the nondeterministic assignments the state space is extended by nondeterministic choices for these new model constants. The technique was presented for concrete transition systems only [Ren06] and works as such only for *qCTL* model checking.

Separation logic

Separation logic [Rey02] is an extension of Hoare logic for the description of heaps. Formulas that describe the state of a program consisting of a store and a heap, roughly corresponding to the state of local (or stack-allocated) variables and dynamically-allocated objects as occurring in dynamic data structures. Recursive predicates are used in order to model data structures. As pointed out in [DP08b], there is a one-to-one correspondence between recursive predicates in separation logic and the nonterminals that are used in our abstract heap representations [JGN14]. Separation logic is classically employed in the form of Hoare-style verification using annotations of programs where decidability of entailment is essential. Whereas this problem is undecidable in general, decidable sub-logics for lists and trees have been developed in [BCO06a; BCO06b]. Lately there are some interesting research results about satisfiability and entailment. In [Bro+13] a technique to decide the satisfiability of separation logic formulae is presented, whereas in [IRS13] a translation from separation logic formulae to MSO formulae over graphs (see Chapter 6) is given, for which both satisfiability and entailment are decidable. Both techniques are restricted to the same fragment of separation logic while the latter in addition is restricted to MSO-formulae describing heap graphs of bounded tree-width, i.e. shares the data structure restrictions of our framework.

There are several separation-logic tools such as Smallfoot [BCO06b] (aiming at (doubly) linked lists and trees) or SpaceInvador [Yan+08], as well as its predecessors SLAyer [BCI11] and Predator [DPV11] for C code aiming at the verification of drivers and system code (both supporting linked lists). These tools are entirely focussed on verifying the absence of memory safety errors and are highly automated.

For the analysis of Java programs there is the tool jStar [DP08a] supporting user defined predicates. jStar natively supports lists and trees. Predicate rules for user-defined data structures as well as pre- and post-conditions need to be provided by the user, whereas loop invariants can be determined automatically. The advantage of tools based on deductive methods is their scalability [Yan+08], while they are restricted to a small set of predefined data structures or need user interaction.

MSO over Graphs

In Chapter 6 we introduce Monadic Second Order (MSO) logic over graphs. We aim to use MSO formulae to describe properties that we want to check for represented heap-structures. To do so also for abstract states, which represent a set of concrete ones, we need to check if the set induced by the MSO formula, namely the set of graphs fulfilling the described property, includes the set of graphs represented by an abstract state.

That this inclusion is decidable is part of a famous theorem by Courcelle [Cou90]. Courcelle gives a decision procedure where – like in the string case – the formula is translated into an (tree) automaton that then can be intersected with the given grammar. The automaton, however, cannot be constructed for arbitrary graphs and graph languages but only for those with a bounded tree-width. The size of the resulting automata is nonelementary in the tree-width.

This nonelementary blowup reduces the theorem to a purely theoretical result without any practical impact so far. Some approaches to reduce the size of the automata were proposed, e.g. using MONA [Hen+95]. However, the size remains unmanageable even for relatively simple formulae and small tree-widths [Sog08]. The problem of the automaton construction is that it depends on the formulae and tree-width only and does not take the given grammar into account. Thus the automaton needs to be able to handle any grammar with the given tree-width. In Section 6.3 we present an alternative approach where we take the given grammar into account yielding promising experimental results at least for simple structures such as grids (see Section 6.6).

Langer *et. al.* recently considered another result from Courcelle, that given a formula and a fixed tree-width it is decidable in linear time whether a graph bounded by this tree-width fulfils the formula. This results suffers from the same practical issues as it also involves the construction of the automata. They developed an algorithm that shares some ideas with our approach [KL09; KLR11; Lan+12]. They use a game-based approach building up so-called characteristic trees. These characteristic trees are quite similar to our evaluation trees (Definition 6.5 on page 135). They represent the whole formula within the tree while our evaluation trees represent only the quantifier prefix of a formula in prenex-normal-form with a set of partial evaluations in the leaves. The algorithm works on a *tree decomposition* [ST93] of the input graph. A tree decomposition can be seen as a derivation tree, such that they check the property for a single derivation, while we evaluate formulae over all possible derivations. This includes to ensure some additional properties like the finiteness of occurring evaluation trees, and the splitting of nonterminals to get unique results, whereas both are ensured trivially if a single, finite derivation is considered. The algorithm from Langer *et. al.* supports an extension of MSO allowing to express optimisation and counting problems.

2

Chapter 2

Preliminaries

In this chapter we introduce some basic notations and concepts that are used in later chapters. Starting with some notations for sequences and functions in Section 2.1, we introduce hypergraphs, a generalisation of the classical graph model, in Section 2.2, that we will use to model heaps. Hyperedge replacement grammars as well as the underlying replacement steps are introduced in Section 2.3. In Section 2.4, we introduce backward applications of grammar rules, i.e. the counter part of the replacements introduced in Section 2.3. Finally, in Section 2.5 and Section 2.6 we will focus on some important properties of hyperedge replacement grammars that we exploit in later chapters.

2.1 Notations

A *sequence* over a finite set S is an ordered set of elements from S. The set of all finite sequences over S, including the empty sequence ε, is denoted by $S^\star$. Given a sequence $s = s_1\, s_2\, \dots\, s_n \in S^\star$, the length of s is denoted by $|s| = n$, given $1 \leq i \leq |s|$, we refer to the ith element of s by $s[i] = s_i$. We denote the set of all elements occurring in s by $[s] = \{s_1, s_2, \dots, s_n\}$. The concatenation of two sequences $s = s_1\, s_2\, \dots\, s_n \in S^\star$ and $t = t_1\, t_2\, \dots\, t_m \in T^\star$ is denoted by $s{\cdot}t = s_1 \dots s_n\, t_1 \dots t_m \in (S{\cup}T)^\star$

We denote the disjoint sum of sets A, B by $A \uplus B = A \cup B$ when $A \cap B = \emptyset$ and the power set of A by $\mathcal{P}(A) = \{B \mid B \subset A\}$. Given a tuple $t = (A, B, C, \dots)$ we write A_t, B_t etc. for the components if their names are clear from the context. Function $f{\restriction}S$ is the restriction of f to S. Function $f : A \to B$ is lifted to sets $f : 2^A \to 2^B$ and to sequences $f : A^\star \to B^\star$ by point-wise application. We denote the identity function on a set S by id_S. We define functions $f : S \to T$ explicitly by giving their mapping as $f := [\, s_1 \mapsto t_1, s_2 \mapsto t_2, \dots s_n \mapsto t_n\,]$, with $s_i \in S$, $t_i \in T$ and $f(s_i) = t_i$ for $1 \leq i \leq n$. Given two functions $f : S \to T$ and $g : U \to V$ the union of f and g is $f \cup g : S \uplus U \to T \cup V$, with $(f \cup g)(x) = f(x)$ if $x \in S$ and $(f \cup g)(x) = g(x)$ if $x \in U$. The natural numbers, *not including zero*, are denoted by $\mathbb{N}$.

2.2 Hypergraphs

Classic *(directed) graphs* are defined as a tuple consisting of a set of vertices and an edge relation, defining the source, target and label of edges. Hypergraphs are a generalisation of directed graphs, where instead of defining edges through a relation between vertices, hyperedges are proper objects not restricted to connect exactly two vertices but can connect an arbitrary number of vertices. The number of incident vertices of an edge is the *rank* of the edge. Instead of distinguishing source and target vertices of edges, there is an order on the incident vertices of an edge. Vertices can be marked as external and there is an order on external vertices. External vertices are used in the replacement steps, which we will define in Section 2.3. The presented definitions and notations are based on the ones from [Hab92].

Example 2.1: Hypergraph

In Figure 2.1 a hypergraph is depicted. Vertices are represented as circles, hyperedges as rectangles. External vertices are marked by bold circle. The order on the external vertices is represented by labels ($1_\bullet$, $2_\bullet$) written next to them. We call the lines connecting edges and their incident vertices tentacles. The numbers labelling the tentacles represent the order on the vertices.

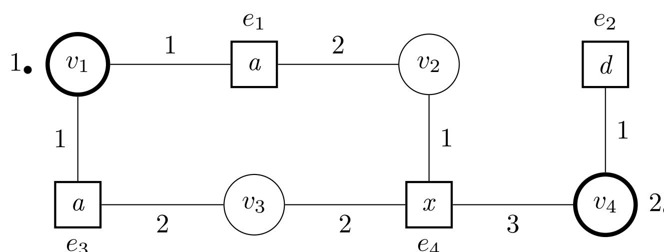


Figure 2.1: Hypergraph $H_{2.1}$

The graph in Figure 2.1 contains four vertices (v_1, v_2, v_3 and v_4). The vertices v_1 and v_4 are external vertices. The first external node is v_1, the second v_4. There are two a-labelled edges (e_1 and e_3) each connecting two vertices. Beside the a-labelled edges there is a d-labelled edge (e_2) only incident to v_4, thus this edge is of rank one and a x-labelled edge (e_4) of rank three, connecting v_2, v_3 and v_4 in that order.

We label hyperedges using a ranked alphabet Σ, that is a set of labels and an associated ranking function rk, assigning each symbol $a \in \Sigma$ its rank $\mathrm{rk}(a) \in \mathbb{N}$. We require the rank of an edge, i.e. the number of incident vertices, to be equal to the rank of its label. Thus in the above example the rank of a is two, while the rank of d is one and the rank of x is three. In the following we will introduce hypergraphs formally.

Definition 2.1 (Hypergraph):
Given a finite ranked alphabet Σ *with associated ranking function* $\mathrm{rk} : \Sigma \to \mathbb{N}$. *A* (labelled) hypergraph over Σ *is a tuple*

$$H = (V, E, \mathrm{att}, \mathrm{lab}, \mathrm{ext})$$

where V *is a finite set of vertices and* E *a finite set of hyperedges. The attachment function* $\mathrm{att} : E \to V^{\star}$ *maps each hyperedge to a sequence of incident vertices, the hyperedge-labelling function* $\mathrm{lab} : E \to \Sigma$ *maps to each edge its label, and* $\mathrm{ext} \in V^{\star}$ *is the (possibly empty) sequence of pairwise distinct external vertices. For every* $e \in E$, *we let* $\mathrm{rk}(e) = |\mathrm{att}(e)|$ *and we require* $\mathrm{rk}(e) = \mathrm{rk}(\mathrm{lab}(e))$.
The set of all hypergraphs over Σ *is denoted by* HG_{Σ}.

The definition uses sequences of vertices to define ordered sets of vertices. Note that the external nodes need to be pairwise distinct, that is that each external vertex has a unique position within the sequence ext. This is not required for the sequences defined by the attachment function, thus edges can be connected to the same node by more than one tentacle. Each edge has at least one incident vertex, as the rank of each label is at least one.

Example 2.2: Formal definition of a hypergraph
Given the ranked alphabet $\Sigma = \{a, d, x\}$ *with ranking function* $\mathrm{rk} : [\, a \mapsto 2, d \mapsto 1, x \mapsto 3\,]$ *we can describe the graph* $H_{2.1}$ *from Fig. 2.1 as:*

$$H_{2.1} = (V, E, \mathrm{att}, \mathrm{lab}, \mathrm{ext})$$

where the vertices are $V = \{v_1, v_2, v_3, v_4\}$ *and the edges* $E = \{e_1, e_2, e_3, e_4\}$. *The attachment function is defined as* $\mathrm{att} := [\, e_1 \mapsto v_1v_2,\ e_2 \mapsto v_4,\ e_3 \mapsto v_1v_3,\ e_4 \mapsto v_2v_3v_4\,]$ *and the labelling function as* $\mathrm{lab} := [\, e_1 \mapsto a,\ e_2 \mapsto d,\ e_3 \mapsto a,\ e_4 \mapsto x\,]$. *The sequence of external vertices is* $\mathrm{ext} = v_1v_4$.

We introduced tentacles informally as the connection lines between edges and incident vertices. As we need $\mathrm{rk}(e) = \mathrm{rk}(lab(e))$ for any edge $e \in E$, the number of tentacles of an edge is defined through the rank of the edge label. We distinguish different tentacle types, depending on the edge label and ordinal of a tentacle.

Definition 2.2 (Tentacle):
Given a symbol $a \in \Sigma$ *and* $1 \leq i \leq \mathrm{rk}(a)$ *we call the tuple* (a, i) *a* tentacle type. *We denote the set of all tentacle types by* $Tent_{\Sigma} = \{(a, i) \mid a \in \Sigma, 1 \leq i \leq \mathrm{rk}(a)\}$. *Given a hypergraph* $H \in HG_{\Sigma}$ *and an edge* $e \in E_H$ *with* $\mathrm{lab}(e) = a$ *we call the tuple* (e, i) *an* (a, i)*-tentacle.*

If it is clear from the context we use tentacle as synonym for tentacle type. In many cases we are interested in the set of tentacles connected to a vertex. We define this set as follows.

Definition 2.3 (Connected tentacles):
Given a hypergraph $H \in HG_\Sigma$ and a vertex $v \in V_H$ the set of tentacles connected to v is defined as

$$Tent_H(v) = \{(e,i) \in E_H \times \mathbb{N} \mid \mathrm{att}_H(e)[i] = v\}.$$

If we are just interested in the different types of tentacles connected to a vertex v we use $Tent_H(v) = \{(\mathrm{lab}(e), i) \mid (e,i) \in Tent(v)\}$.

A special type of hypergraphs are handles. These hypergraphs consist of a single edge and one external vertex for each of its tentacles.

Definition 2.4 (Handle):
*Given $x \in \Sigma$ with $\mathrm{rk}(x) = n$, an x-*handle *is the hypergraph*

$$x^\bullet = (\{v_1, \ldots, v_n\}, \{e\}, [\,e \mapsto v_1 \ldots v_n\,], [\,e \mapsto x\,], v_1 \ldots v_n) \in HG_\Sigma.$$

2.3 Hyperedge Replacement Grammars

Like string grammars describe sets of strings we use graph grammars to describe sets of graphs. Therefore we split the alphabet up in terminal and nonterminal labels. Nonterminal labelled edges will be replaced by hypergraphs as nonterminals are replaced by strings in the string case.

Definition 2.5 (Ranked alphabet):
A ranked alphabet is a set $\Sigma = N \uplus T$, consisting of a set N of nonterminals, and T a set of terminals, with associated ranking function $\mathrm{rk} : N \cup T \mapsto \mathbb{N}$ assigning each label a rank.

Given a ranked alphabet Σ we use T_Σ to refer to the terminal labels of Σ and N_Σ to refer to the nonterminal labels. We denote terminal labels, i.e. elements from the set T by lowercase letters and nonterminal labels, elements of the set N, by uppercase letters. We call edges *nonterminal edges* if they are labelled with nonterminals and edges labelled with terminals *terminal edges*. Given a hypergraph $H \in \mathrm{HG}_\Sigma$ we use $E_H^N = \{e \in E_H \mid \mathrm{lab}_H(e) \in N_\Sigma\}$ to denote the set of nonterminal edges and $E_H^T = \{e \in E_H \mid \mathrm{lab}_H(e) \in T_\Sigma\}$ to denote the set of terminal edges of H. We use T_Σ as alphabet

with ranking function $\text{rk}\upharpoonright T_\Sigma$. Correspondingly HG_{T_Σ} is the set of all hypergraphs with only terminal labels from Σ. We call these graphs terminal graphs.

We now define hyperedge replacement grammars (HRG). Like string grammars, they are based on replacements of nonterminals, i.e. replacements of nonterminal edges by hypergraphs. This works as follows: given a graph we replace a nonterminal edge by merging external vertices of the replacement graph with the incident vertices of the replaced edge.

Definition 2.6 (Hyperedge replacement):
Let $H, K \in HG_\Sigma$ and $e \in E_H$ an edge with $rk(e) = |ext_K|$. The replacement of e by K in H*, denoted $H[K/e]$, is the hypergraph $H' \in HG_\Sigma$ defined by:*

$$\begin{aligned}
V_{H'} &= V_H \uplus (V_K \setminus [\text{ext}_K]) \\
E_{H'} &= (E_H \setminus \{e\}) \uplus E_K \\
\text{lab}_{H'} &= (\text{lab}_H \upharpoonright (E_H \setminus \{e\})) \cup \text{lab}_K \\
\text{att}_{H'}(f)[i] &= \begin{cases} \text{att}_H(e)[j] & ,\textit{if } f \in E_K \;\wedge\; att_K(f)[i] = \text{ext}[j] \\ \text{att}_K(f)[i] & ,\textit{if } f \in E_K \;\wedge\; att_K(f)[i] \notin [\text{ext}_K] \\ \text{att}_H(f)[i] & ,\textit{otherwise } (f \in E_H \setminus \{e\}) \end{cases} \\
\text{ext}_{H'} &= \text{ext}_H
\end{aligned}$$

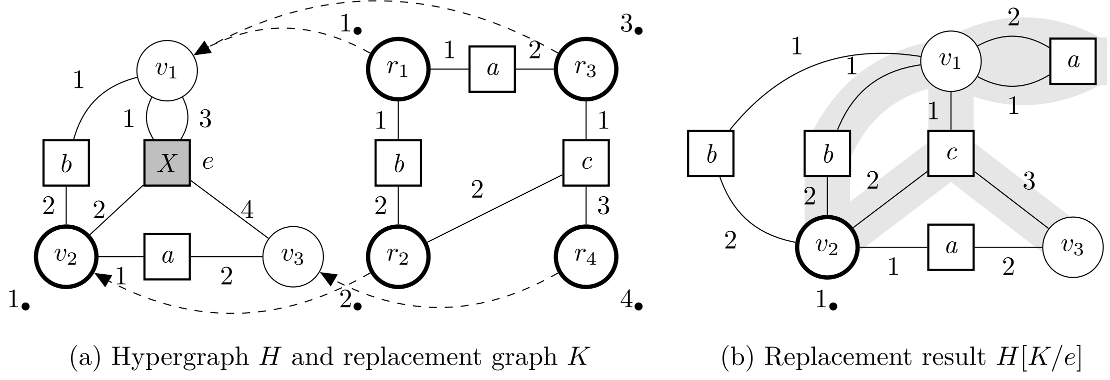
We refer to K as the replacement graph.

Example 2.3: Hyperedge replacement
In Figure 2.2 a replacement is depicted. Figure 2.2(a) depicts the hypergraph H, on the left-hand side with the edge e (the gray X-edge) that is replaced by the replacement graph K given on the right-hand side of the graph. The corresponding mapping of external nodes of K to incident nodes of e is depicted by dashed arrows. The resulting graph $H[K/e]$ is depicted in Figure 2.2(b). The part of the resulting graph that is marked in grey are the elements added to H by the replacement.

Note that in this example the first and third external node of the replacement graph are mapped to the same node. That is because the first and third tentacle of the edge e is attached to the same node.

We assume, without loss of generality, that the set of nodes and edges of H and K are disjoint, i.e. $V_H \cap V_K = \emptyset$ and $E_H \cap E_K = \emptyset$. Should this not be the case, then we can establish this by an appropriate renaming of K or H.

Based on the replacements defined above we introduce hyperedge replacement grammars as a set of grammar rules that describe possible replacements for nonterminal edges.



(a) Hypergraph H and replacement graph K

(b) Replacement result $H[K/e]$

Figure 2.2: Replacement of a hyperedge

Definition 2.7 (Hyperedge replacement grammar (HRG)):
A hyperedge replacement grammar G *over the ranked alphabet* Σ *is a set of production rules of the form* $X \to R$*, where* $X \in N_\Sigma$ *is a nonterminal that forms the left-hand side and* $R \in HG_\Sigma$ *is the* rule graph*, a hypergraph with* $|\text{ext}_R| = \text{rk}(X)$*, the right-hand side of the rule.*

Note that we require the number of external nodes of the rule graph to be equal to the rank of the nonterminal on the left-hand side to meet the needs for the hyperedge replacement as defined in Def. 2.6.

Given a grammar $G \in \text{HRG}_\Sigma$ and nonterminal $X \in N_\Sigma$ we denote the rules for nonterminal X by $G^X = \{X \to R \in G\}$. Correspondingly we use $G^{\overline{X}}$ to denote the grammar rules whose left-hand sides are different from X.

Given a grammar $G \in \text{HRG}_\Sigma$ and production rule $p : X \to R \in G$ we write $H \Rightarrow_p H'$ iff there exists an edge e in E_H with $\text{lab}(e) = X$ and $H[R/e] = H'$. We write $H \Rightarrow_G H'$ iff there exists a rule $p \in G$ with $H \Rightarrow_p H'$. Finally, we denote by $\Rightarrow_G^*$ the transitive reflexive closure of $\Rightarrow_G$ and omit the index G if it is clear from the context. We call $H \Rightarrow^* H'$ a derivation.

We define the language of a HRG based on derivations as define above. Notice that we did not introduce a starting symbol for hyperedge replacement grammars (as in string case). Instead we use a parametrized language definition, where we use start graphs instead of nonterminals. The language is the set of all terminal graphs derivable from the start graph.

Definition 2.8 (Language of a hyperedge replacement grammar):
Given $G \in HRG_\Sigma$ *the language defined by a start graph* $H \in HG_\Sigma$ *is the set*

$$L_G(H) := \{H' \in HG_{T_\Sigma} \mid H \Rightarrow_G^* H'\}.$$

Given a set S of start graphs we use $L_G(S) = \bigcup_{H \in S} L_G(H)$.

Example 2.4: A HRG for trees

In Figure 2.3 a hyperedge replacement grammar is given. The grammar contains four rules for the nonterminal T. The language of this grammar starting from the T-handle $T^\bullet$ contains all binary trees.

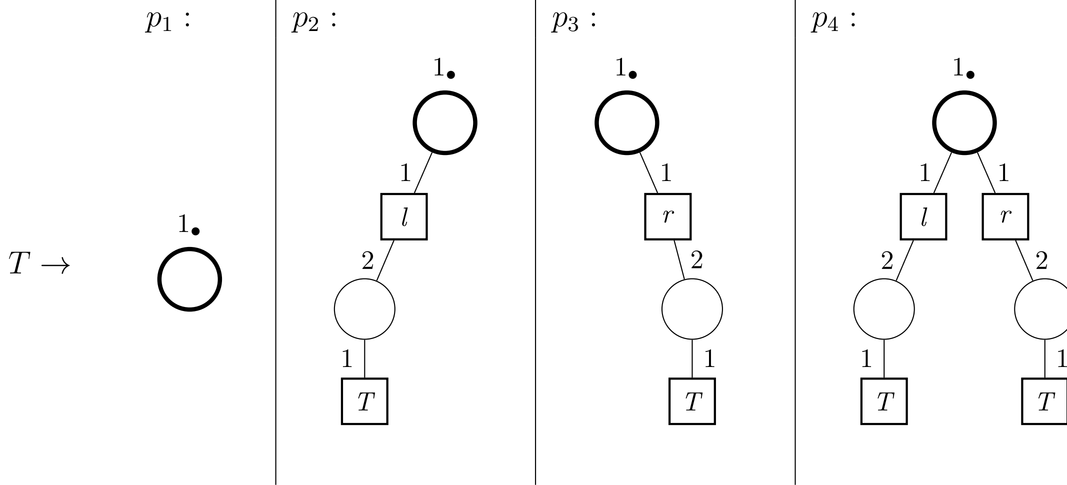


Figure 2.3: A grammar generating arbitrary binary trees.

This grammar defines binary trees as a recursive structure. Starting from the T-handle the unique (external) node is the root node of a binary tree. Depending on which rule we choose to replace the unique edge we get an empty tree (p_1), only a left (or right) successor that again is a binary tree (p_2 or p_3), or a left and a right successor which are both roots of subtrees (p_4).

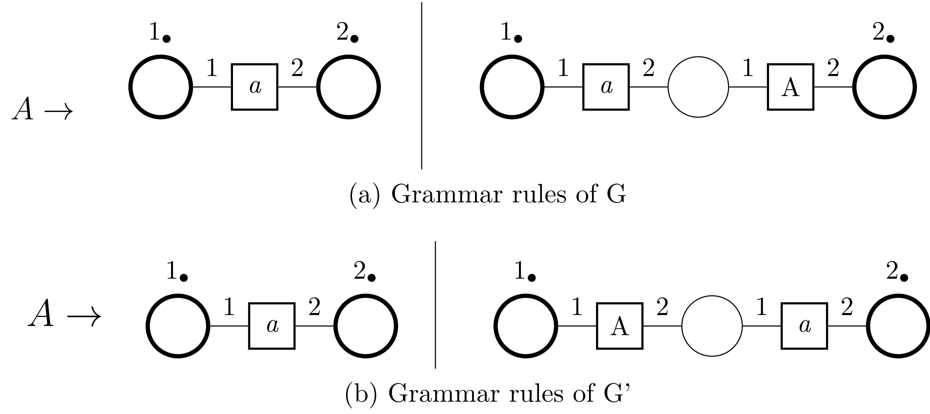
We call two HRGs over the alphabet Σ equivalent if they produce the same language for any start graph $H \in \mathrm{HG}_\Sigma$.

Definition 2.9 (Grammar equivalence):
Grammars $G, G' \in HRG_\Sigma$ are equivalent*, denoted $G \simeq G'$, iff for any hypergraph $H \in HG_\Sigma$ it holds that $L_G(H) = L_{G'}(H)$.*

Example 2.5:

Consider grammars $G, G' \in HRG_\Sigma$, with $\Sigma = (\{A\}, \{a\}, [\, a \mapsto 2, A \mapsto 1\,])$. The rules of G and G' are given in Figure 2.4.

Both grammars can generate lists of a-edges from A. In G, the lists are built from left to right, while in G' the lists are built from right to left. However, $L_G(A^\bullet) = L_{G'}(A^\bullet)$. In Section 2.5, we introduce a decomposition lemma (Lemma 2.4) from which we can conclude $G \simeq G'$ from $L_G(A^\bullet) = L_{G'}(A^\bullet)$.



(a) Grammar rules of G

(b) Grammar rules of G'

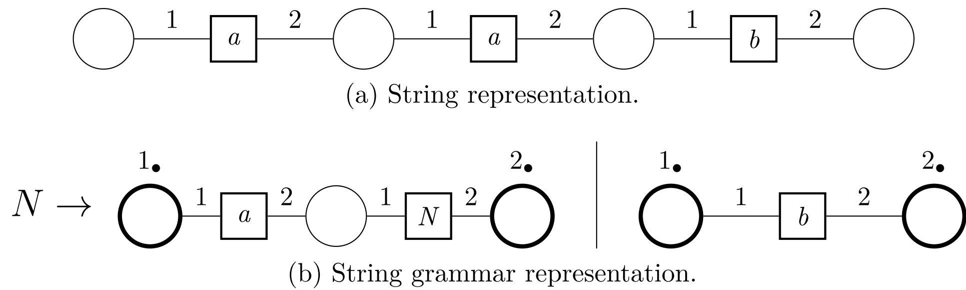
Figure 2.4: Grammar rules.

Strings can be uniquely represented by heap graphs containing chains of terminal edges only, production rules can be translated to heap abstraction grammars analogous to Example 3.6 on page 52. We say that a string grammar is in GNF if and only if its representation as heap abstraction grammar is in LGNF.

Theorem 2.1 (Undecidability of grammar equivalence.):
It is undecidable whether two grammars $G_1, G_2 \in HRG_\Sigma$ are equivalent.

The theorem follows directly from the undecidability result for context-free string grammars [AB02], as context free string grammars can be simulated by HRGs [Hab92]. To do so we represent each character as an edge of rank two and strings as a chain of edges.

Example 2.6:
In Figure 2.5 the graph representations for the word $w = aab$ (a) and the string grammar $N \to aN \mid b$ (b) are given. For any string grammar representation the nonterminals are of rank two.



(a) String representation.

(b) String grammar representation.

Figure 2.5: String graphs

If two grammars are able to produce, up to a finite difference, the same graphs we call them quasi-equivalent.

Definition 2.10 (Grammar quasi-equivalence):
Grammars $G \in HRG_\Sigma$ and $G' \in HRG_{\Sigma'}$ are quasi-equivalent, *denoted $G \approx G'$, iff there exist functions $\gamma_{G\to G'} : HG_\Sigma \to \mathcal{P}(HG_{\Sigma'})$ and $\gamma_{G'\to G} : HG_{\Sigma'} \to \mathcal{P}(HG_\Sigma)$ such that*

$$\begin{aligned} & L_G(H) = L_{G'}(\gamma_{G\to G'}(H)) \quad \forall H \in HG_\Sigma \\ \textit{and} \quad & L_{G'}(H') = L_G(\gamma_{G'\to G}(H')) \quad \forall H' \in HG_{\Sigma'}. \end{aligned}$$

Note that the above definition implies that $T_\Sigma = T_{\Sigma'}$ as otherwise the language equivalence could not be established. Grammar equivalence is a special case of Grammar quasi-equivalence.

Example 2.7:
Reconsider alphabet Σ and grammar G from Example 2.5 and the grammar $G'' \in HRG_\Sigma$ with the grammar rules given in Figure 2.6.

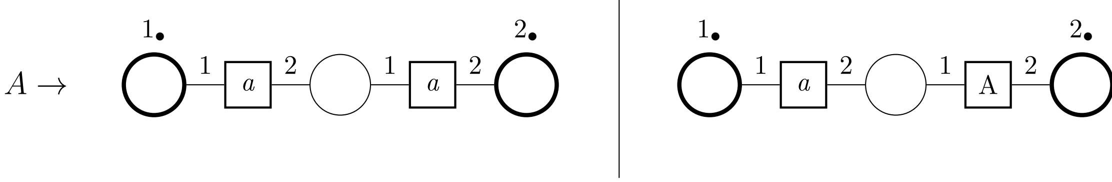


Figure 2.6: Rules of G''.

Grammar G'' is not equivalent to G and G', as nonterminal A in G represents lists of at least one edge, while in G'' the list represented by A contains at least two edges. But as both represent lists composed of a-edges the difference is only one edge. The grammars G'' and G are quasi-equivalent. Here $\gamma_{G''\to G}$ adds additional a-edges to any A-edge within the graph such that it represents a list of at least two a-edges. e.g. $\gamma_{G''\to G}(A^\bullet) = \{R_2\}$, where R_2 is the rule graph of the second grammar rule of G, while $\gamma_{G\to G''}(A^\bullet) = \{a^\bullet, A^\bullet\}$, i.e. a single a-edge or a list of at least two a-edges.

According to the decomposition lemma (Lemma 2.4 on page 27) it suffices to define γ on nonterminals, and extend it to arbitrary nonterminal hypergraphs by considering all possible replacements of nonterminals by graphs defined through the γ-mapping, i.e. for any nonterminal $X \in N$ we replace any X-edges in H by graphs from $\gamma(X^\bullet)$.

2.4 Backward Rule Application

In Definition 2.6 we defined replacements of hyperedges that we use to realise forward applications of grammar rules, namely derivations. Grammar rules can also be applied in a backward manner. If we can derive H' from H ($H \Rightarrow_p^* H'$), i.e. we get H' from H by applying rule p, then we get H' from H by backward application of p, denoted by $H \Leftarrow_p^* H'$.

Obviously the implicit definition of backward application as the inverse of the forward application is not suitable for implementation purposes. Instead we define backward application explicitly through a replacement operation. We replace a whole subgraph by a single nonterminal edge. The subgraph must be isomorphic to some rule graph and is identified as an embedding of the rule graph. The definition of embedding of rule graph, that we give here, is related to subgraph isomorphism [Ull76], where a subgraph is an arbitrary subset of vertices and edges between these. However, the definition of embedding is more restrictive as it requires that only external vertices of the embedded subgraph are connected to the rest of the graph.

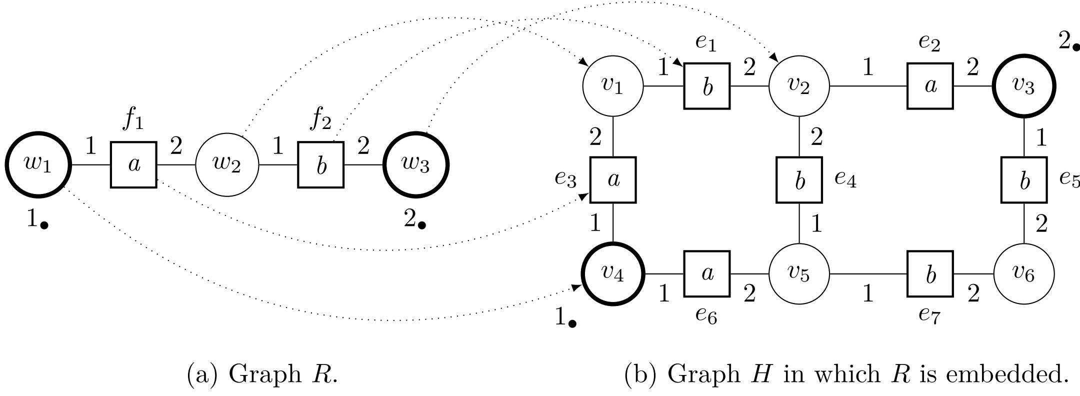
Definition 2.11 (Hypergraph embedding):
Given $I, H \in HG_\Sigma$, an embedding *emb of I in H is a pair of mappings $emb_V : V_I \to V_H$ and $emb_E : E_I \to E_H$ with the following properties:*

$$
\begin{aligned}
emb_V(v) &\notin [ext_H] && \forall v \in V_I \setminus [\mathrm{ext}_I] \\
emb_V(v) &\neq emb_V(v') && \forall v \in V_I, v' \in V_I \setminus [\mathrm{ext}_I].\, v \neq v' \\
emb_E(e) &\neq emb_E(e') && \forall e, e' \in E_I \text{ with } e \neq e' \\
\mathrm{lab}_I(e) &= \mathrm{lab}_H(emb_E(e)) && \forall e \in E_I \\
emb_V(\mathrm{att}_I(e)) &= \mathrm{att}_H(emb_E(e)) && \forall e \in E_I \\
e \notin emb_E(E_I) &\Rightarrow [\mathrm{att}_H(e)] \cap emb_V(V_I \setminus [\mathrm{ext}_I]) = \emptyset && \forall e \in E_H.
\end{aligned}
$$

Let $Emb(I, H)$ denote the set of all embeddings of I in H.

Example 2.8: Hypergraph embedding
In Figure 2.7 hypergraphs R (a) and H (b) are given. An embedding of R in H is denoted by the dotted arrows. The single inner vertex w_2 of R is mapped to the inner vertex v_1 of H, for which any attached edge is part of the embedding. The depicted embedding is the only one for R in H. The vertices v_4, v_5, and v_6 cannot be used for an embedding, as an additional edge is attached to v_5 (which would be the image of the inner vertex w_2) that would not have a counterpart in graph R. If we would try to construct an embedding with v_2, v_3, and v_6 we would map the inner node w_2 to v_3, which is an external vertex.

Given an embedding of a subgraph we can replace the embedded graph by a single hyperedge, as follows.



(a) Graph R.

(b) Graph H in which R is embedded.

Figure 2.7: Hypergraph embedding.

Definition 2.12 (Hypergraph replacement):
Given $G \in HRG_\Sigma$, $I, H \in HG_\Sigma$, $emb \in Emb(I, H)$ and $X \in N$ with $\mathrm{rk}(X) = |ext_I|$, replacing I in H by $X^\bullet$ results in $replace(I, H, emb, X) = K$ where

$$\begin{aligned} V_K &= V_H \setminus emb_V(V_I \setminus [\mathrm{ext}_I]) \\ E_K &= (E_H \setminus emb_E(E_I)) \uplus \{e\} \\ \mathrm{lab}_K &= (\mathrm{lab}_H \restriction E_K) \cup \{e \mapsto X\} \\ \mathrm{att}_K &= (\mathrm{att}_H \restriction E_K) \cup \{e \mapsto emb_V(\mathrm{ext}_I)\} \\ \mathrm{ext}_K &= \mathrm{ext}_H. \end{aligned}$$

Let us explain this definition. We remove the embedded edges as well as the vertices that are images of internal vertices. The resulting gap between the images of external vertices is filled by a single nonterminal edge e, attached corresponding to the ext-sequence of the embedded graph. If we would allow embeddings where an inner vertex is mapped on to a vertex attached to an edge not part of the embedding, this would result in a dangling edge as we do not remove the edge but one of its attached vertices. If we would allow embeddings where inner vertices are mapped to external vertices, the replacement would remove an external vertex.

Example 2.9: Hypergraph replacement
The result of replacing R in H by nonterminal X under the embedding from Figure 2.7 is depicted in Figure 2.8.

The following lemma now readily follows from Definition 2.11 and Definition 2.12. It states that the replacement of embeddings of rule graphs by the nonterminal of its righthand side yields the inversion of the application. It thus is the backward application of the rule.

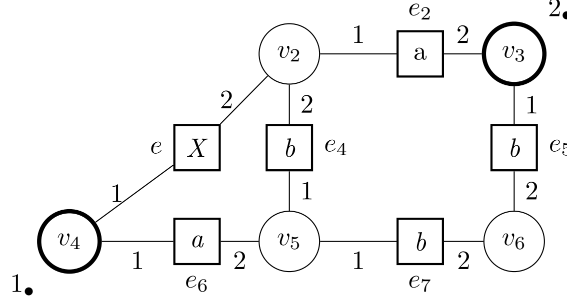


Figure 2.8: The graph $\mathit{replace}(R, H, \mathit{emb}, X)$ with $R, H,$ *and emb* as given in Figure 2.7

Lemma 2.1 (Backward application and hypergraph replacement.):
Given $G \in HRG_\Sigma$ and $H, I, K \in HG_\Sigma$, $K \Rightarrow_G H$ iff there exists a rule $X \to I \in G$ and $\mathit{emb} \in \mathit{Emb}(I, H)$ such that $K = \mathit{replace}(I, H, \mathit{emb}, X)$.

Given grammar $G \in \mathrm{HRG}_\Sigma$, rule $p : X \to R \in G$, hypergraphs $H, H' \in \mathrm{HG}_\Sigma$ and $\mathit{emb} \in \mathit{Emb}(R, H)$, we write $H \Leftarrow_p H'$ iff $\mathit{replace}(R, H, \mathit{emb}, X) = H'$. We write $H \Leftarrow_G H'$ iff there exists a rule $p \in G$ with $H \Leftarrow_p H'$. We omit the index G if it is clear from the context and denote the transitive reflexive closure of $\Leftarrow_G$ by $\Leftarrow_G^*$.

2.5 Inherent Properties

In this section we present some properties of the replacement steps and of the hyperedge replacement grammars. These properties will be used in various proofs of the dissertation.

We call a HRG confluent, if its derivation relation is so. That is the replacement of one hyperedge does not effect the replacement of another. Thus, given replacements of different edges, the order of application does not affect the result.

Lemma 2.2 (Confluence [Hab92]):
Given hypergraphs $H, K_1, K_2 \in HG_\Sigma$ and $e_1, e_2 \in E_H$ with $\mathrm{rk}(e_1) = |ext_{K_1}|$ and $\mathrm{rk}(e_2) = |ext_{K_2}|$ then it holds, that

$$(H[K_1/e_1])[K_2/e_2] = (H[K_2/e_2])[K_1/e_1]$$

As for a set of replacements possible at the same time, the order of application does not change the resulting graph, given $H, K_1, \ldots K_i \in \mathrm{HG}_\Sigma$, $e_1, \ldots e_i \in E_H$ with $\mathrm{rk}(e_i) = |\mathrm{ext}_{K_i}|$ we also write $H[K_1/e_1, \ldots, K_i/e_i]$ as an alternative for $H[K_1/e_1] \ldots [K_i/e_i]$. This

can be read as a parallel replacement of edges in H and will be used to clarify the independence of the replacements.

If we have two subsequent replacements it is possible that the second replacement replaces an edge that was inserted by the first one, i.e. the second replacement was not possible within the original hypergraph. In this case we can also change the order of the replacement, by realising the latter on the replacement graph of the former first. The resulting graph is then be used as replacement graph for the replacement of the edge within the original graph. We say HRGs are associative.

Lemma 2.3 (Associativity [Hab92]):
Given hypergraphs $H, K, I \in HG_\Sigma$ *and* $e_H, \in E_H, e_K \in E_K$ *with* $\mathrm{rk}(e_H) = |ext_K|$ *and* $\mathrm{rk}(e_K) = |ext_{K_I}|$ *it holds, that*

$$(H[K/e_H])[I/e_K] = H[K[I/e_K]/e_H]$$

Hyperedge replacement grammars are sometimes referred to as context-free graph grammars, because the possible replacements of a hyperedge are not affected by its context. The confluence and associativity lemmas state this context-freeness for single derivation steps. The following decomposition lemma extends this to arbitrary derivations.

Lemma 2.4 (Decomposition [Hab92]):
Given $G \in HRG_\Sigma$ *and* $H, H' \in HG_\Sigma$ *with* $\{e_1, \dots e_n\} = E_H$ *it holds that* $H \Rightarrow^* H'$ *iff*

$$\exists K_1, \dots, K_n \in HG_\Sigma.\ (\ H[K_1/e_1, \dots, K_n/e_n] = H' \wedge\ \forall\ 1 \leq i \leq n.\ \mathrm{lab}_H(e_i)^\bullet \Rightarrow^* K_i\ ).$$

The decomposition lemma states that any derivation starting in a hypergraph H can be decomposed into a set of independent derivations, one for each edge of H. Note that these derivations could also be empty derivations, such that $K_i = \mathrm{lab}_H(e_i)^\bullet$, i.e. edges can remain in the graph. This especially is useful for terminal edges. The proof of Lemma 2.4 can be found in [Hab92]. If we move from derivations to languages we get the following result.

Corollary 2.1 (Language decomposition)
Given $G \in HRG_\Sigma$ *and* $H \in HG_\Sigma$ *with* $\{e_1, \dots e_n\} = E_H$ *it holds that*

$$L(H) = \{H[e_1/K_1, \dots e_n/K_n] \mid K_i \in L(\mathrm{lab}_H(e_i)^\bullet)\ \forall 1 \leq i \leq n\}$$

2.6 Properties of HRGs

In this section we consider properties that are not inherent for HRGs but often good to have, and that we need in later chapters.

Productiveness

Productiveness is well-known from string grammars. A grammar is productive if any nonterminal is productive, i.e. a terminal graph can be derived from any nonterminal. Thus the language of any nonterminal is non-empty.

Definition 2.13 (Productive grammar):
We call $G \in HRG_\Sigma$ productive *iff any nonterminal is productive, i.e.*

$$\forall X \in N_\Sigma. \quad L_G(X^\bullet) \neq \emptyset$$

Lemma 2.5 (Equivalent productive grammar.):
For any non-productive $G \in HRG_\Sigma$ there exists a productive $G' \in HRG_{\Sigma'}$ with $G' \approx G$ and $\Sigma' \subseteq \Sigma$.

Proof. We prove Lemma 2.5 by construction of G'. A nonterminal $X \in N$ is productive iff $L(X^\bullet) \neq \emptyset$. Let $N' \subseteq N$ contain the productive nonterminals of Σ. N' is obtained as follows. We initialise the set N' with all $X \in N$ for which at least one X-rule has a terminal rule graph, i.e., $X \to H$ where $H \in \mathrm{HG}_{T_\Sigma}$. Then, $X \in N \setminus N'$ is added if there exists a rule $X \to H$ such that all nonterminals occurring in H are productive, i.e., $E_H^N \subseteq N'$. This procedure terminates as both G and N are finite.

We then collect all rules of G that on both sides only contain nonterminals from N', i.e. we let $G' = \{X \to R \in G \mid X \in N' \wedge E_R^N \subseteq N'\}$. It is evident that this yields $L_G(H) = L_{G'}(H)$ for any $H \in \mathrm{HG}_\Sigma$ as G' contains any rule from G that can be used to derive terminal graphs. As all non-productive nonterminals are removed, the resulting grammar is obviously productive. □

We call a grammar increasing if any derivation results in a larger graph, i.e in a graph containing more (terminal) edges or more nodes.

Definition 2.14 (Increasing grammar):
We call $G \in HRG_\Sigma$ increasing, *if for any production rule $p : X \to R$ either the size of R, $|E_R| + |V_R| > |E_{X^\bullet}| + |V_{X^\bullet}|$ or R has terminal edges, i.e. $E_R^T \neq \emptyset$. A grammar that is not increasing is called* non-increasing.

Lemma 2.6 (Equivalent increasing grammar.):
For every non-increasing $G \in HRG_\Sigma$ there exists an increasing $G' \in HRG_\Sigma$ with $G' \approx G$.

We will not prove Lemma 2.6 here, but in Section 3.4.2 we introduce the local Greibach Normal Form (Definition 3.8 on page 52), that implies increasingness (Lemma 3.4 on page 60). Every (bounded) HRG can be transformed into local Greibach Normal Form (Theorem 3.3).

Backward Confluence

We have seen that confluence is a property of the forward application of production rules. However, this is not the case for backward application. We define the backward confluence of grammars as follows.

Definition 2.15 (Backward confluence):
A grammar $G \in HRG_\Sigma$ is (locally) backward confluent *iff for any $H \in HG_\Sigma$ with $H \Leftarrow_G K_1$ and $H \Leftarrow_G K_2$ there is a $K \in HG_\Sigma$ with $K_1 \Leftarrow^* K$ and $K_2 \Leftarrow^* K$.*

As in the string case, local backward confluence for terminating hypergraph transformation systems implies global confluence [Plu05]. Plump showed that confluence for general hypergraph transformation systems is not decidable [Plu05]. However, there is a class of transformation systems, called coverable systems, for which confluence is decidable [Plu10]. HRGs as defined here belong to this class.

Theorem 2.2 (Decidability of backward confluence):
It is decidable whether a grammar $G \in HRG_\Sigma$ is backward confluent.

To check confluence of hypergraph rewriting systems we consider, as in the string and term rewriting case, so called *critical pairs*. Given $G \in \mathrm{HRG}_\Sigma$ a *critical pair* for the backward application is a graph and a tuple of backward applications $K_1 \Rightarrow_{p_1} H \Leftarrow_{p_2} K_2$ with $p_i = X_i \to R_i$, such that there exist $emb_1 \in Emb(R_1, H)$ with $K_1 = replace(R_1, H, emb_1, X_1)$ and $emb_2 \in Emb(R_2, H)$ with $K_2 = replace(R_2, H, emb_2, X_2)$ with

1. Any element of the graph H is matched by one of the embeddings, i.e. $emb_1(V_{K_1}) \cup emb_2(V_{K_2}) = V_H$ and $emb_1(E_{K_1}) \cup emb_2(E_{K_2}) = E_H$.
2. The embeddings are not independent, i.e. $emb_1(V_{K_1} \setminus \mathrm{ext}_{K_1}) \cap emb_2(V_{K_2} \setminus \mathrm{ext}_{K_2}) \neq \emptyset$ or $emb_1(E_{K_1}) \cap emb_2(E_{K_2}) \neq \emptyset$. That is $emb_1 \notin Emb(R_1, K_2)$ and $emb_2 \notin Emb(R_2, K_1)$.

Note that due to the first requirement critical pairs are minimal and as for any grammar the number of production rules is finite, also the number of critical pairs is finite. We can construct all critical pairs of a grammar by overlapping right-hand sides of rules.

Example 2.10: Critical Pair

Consider the grammar G from Figure 2.4(a) on page 22. There is a unique critical pair for G depicted in Figure 2.9.

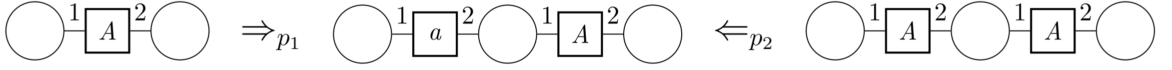


Figure 2.9: A critical pair.

We call a critical pair $K_1 \Rightarrow_{p_1} H \Leftarrow_{p_2} K_2$ *joinable* if there exists a $H' \in \mathrm{HG}_\Sigma$ with $K_1 \Leftarrow^* H' \Rightarrow^* K_2$ [Plu05]. The critical pair from Example 2.10 is not joinable, as there are no rules in G that could be applied backward to K_1 and K_2, which are not isomorphic. A critical pair that is not joinable is a minimal counter example for confluence. Thus any grammar that contains critical pairs that are not joinable is not backward confluent. However, even if all critical pairs are joinable this does not impose confluence.

Example 2.11: A grammar that is not backward confluent.

Consider the grammar G given in Figure 2.10(a). G contains a single critical pair $K_1 \Rightarrow_{p_1} H \Leftarrow_{p_2} K_2$, depicted in Figure 2.10(b). The critical pair is joinable, as K_1 and K_2 are isomorphic. However, if we add an appropriate context to H as in Figure 2.10(c) we get a graph for which the application of p_1 and p_2 yield graphs that are not joinable.

We call a critical pair $K_1 \Rightarrow_{p_1} H \Leftarrow_{p_2} K_2$ *strongly joinable* [Plu05] if it is joinable, i.e. $K_1 \Leftarrow^* H'_1$ and $K_2 \Leftarrow^* H'_2$ with H'_1 isomorphic to H'_2 and there is an isomorphism $i : (V_{H'_1} \cup E_{H'_1}) \to (V_{H'_2} \cup E_{H'_2})$, such that $i \restriction V_{H'_1} = \mathrm{id}_{V_{H'_1}}$. That is we consider the identity of the *persistent vertices*, i.e. vertices from H that remain in H'. If all critical pairs of a hypergraph transformation system are *strongly joinable* then the system is confluent [Plu05]. Note that the critical pair from Example 2.11 is not strongly joinable.

For a transformation system which is *coverable* it holds that it is confluent if and only if its critical pairs are *strongly joinable* [Plu10]. A transformation system is called coverable if for any critical pair there exists a context (*cover*), that cannot be replaced due to some rule of the system and that uniquely identifies the persistent vertices. One possibility to realise such a cover is by edges with labels not occurring in any rule of the system. In Example 2.11 we use an x-edge to realise the cover. If there would be more than two persistent vertices we would use a path of x-edges. This approach does not work if every

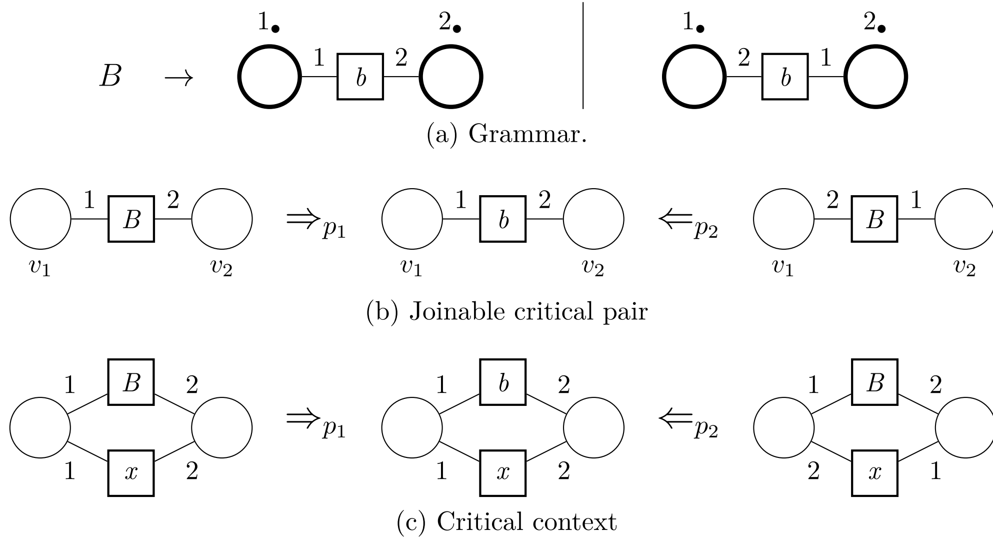


Figure 2.10: Not backward confluent grammar.

label occurs within the rules. However, in the case of HRGs and hypergraphs as defined by us, we can use the sequence of external vertices to uniquely identify the persistent vertices. That is because external vertices can neither be removed nor interchanged via backward applications of rules. Thus HRGs are coverable and thus backward confluence is decidable. Figure 2.11 depicts a cover realised via external vertices.

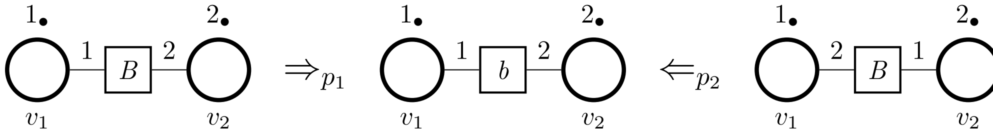


Figure 2.11: A cover based on external vertices.

Compositional Properties

There is a class of properties that we call *compositional properties*. Given a graph composed from subgraphs, the evaluation of a composed property can be determined from the evaluations of the subgraphs. There are various compositional properties of interest. In Chapter 3 we introduce typing (Definition 3.7 on page 44) as compositional property and in Chapter 6 we show that any property defined via a MSO-formula is compositional.

A compositional property consists of a set of possible valuations V and an evaluation function $\eta : \mathrm{HG}_{T_\Sigma} \to V$ mapping terminal hypergraphs to valuations. To make the property compositional the evaluation for a composition must be uniquely determined by the evaluation of the composed parts.

Definition 2.16 (Compositional property):
A compositional property *is a pair* (η, V) *of a final set of values* V *and an evaluation function* $\eta : HG_{T_\Sigma} \to V$*, assigning each terminal graph a value, such that given a nonterminal hypergraph* $H \in HG_\Sigma$ *with nonterminal edges* $\{e_1, \ldots e_n\} = E_H^N$

$$\eta(H[K_1/e_1 \ldots K_n/e_n]) = \eta(H[K'_1/e_1 \ldots K'_n/e_n])$$

holds whenever $\eta(K_i) = \eta(K'_i) \quad \forall\, 1 \leq i \leq n$*, for* $K_i \in HG_{T_\Sigma}$ *and* $|ext_{K_i}| = |att_H(e_i)|$*, i.e. it suffices to know the properties of the replacement graphs to decide the property for the resulting terminal graph.*

Example 2.12: Compositional properties.
The property that determines if a graph contains a terminal edge of a certain label is compositional. The set of values would be $V = \{true, false\}$*. For terminal graphs this property can be easily checked and evaluated to* `true` *or* `false`*. Given a nonterminal graph without a terminal edge of the specific label (otherwise the property would be trivially fulfilled for any replacement) we just have to check if any of the replacement graphs contains such a terminal edge, i.e.* η *maps one of them to the value true.*

Checking if a hypergraph contains exactly three edges of a specific label could also be described by a compositional property where $V = \{=0, =1, =2, =3, >3\}$*, stating that there are none, one, two, three or more than three edges of the label. The number can easily be determined for terminal graphs, and for nonterminal graphs by summing up the number of terminal edges of the label together with the number of each replacement graph.*

Given a nonterminal graph we are in general not interested in a single replacement of the nonterminal edges but in the language this graph induces. Different hypergraphs from the language could evaluate to different values, and thus there is no unique value for the language of a nonterminal graph but a set of possible evaluations. However, by splitting the language of each nonterminal into subsets, such that each evaluates to exactly one value, we can determine a quasi-equivalent grammar for which the language of any nonterminal graph has a unique evaluation.

Lemma 2.7 (Splitting grammar):
For any grammar $G \in HRG_\Sigma$ *and compositional property* $p = (\eta, V)$ *a quasi-equivalent grammar* $G_p \in HRG_{\Sigma_p}$ $(G_p \approx G)$ *can be constructed such that*

$$\forall X \in N_{\Sigma_p}.\ \ I, K \in L_{G_p}(X^\bullet) \ \Rightarrow\ \eta(I) = \eta(K).$$

Proof. Given an alphabet $\Sigma = N \cup T$, grammar $G \in \mathrm{HRG}_\Sigma$ and compositional property $p = (\eta, V)$, we define the alphabet $\Sigma_p = N_p \cup T$ with $N_p = N \times V$ and $\mathrm{rk}((X, v)) = \mathrm{rk}(X)$

$\forall X \in N, v \in V$. The construction is based on the invariant that $\eta(K) = v$ for any $K \in L((X,v)^\bullet)$.

To determine the production rules of $G_p \in \mathrm{HRG}_{\Sigma_p}$ we proceed as follows. We first generate a set of right-hand side graphs with replaced nonterminals for each nonterminal $X \in N_\Sigma$: $R^X = \{t(H) \mid X \to H \in G\}$. Here $t(H)$ is the set of graphs that we get by replacing nonterminals in H by corresponding ones from N_p:

$$t(H) = \{H[X_1{}^\bullet/e_1]\dots[X_k{}^\bullet/e_k] \mid \{e_1 \dots e_k\} = E_H^N \bigwedge_{i=1}^{k} X_i = (lab(e_i), v) \in N_p\}$$

By presuming that for any nonterminal $(X,v) \in N_p$ the evaluation $\eta(K) = v$ for any $K \in L((X,v)^\bullet)$, the grammar G_p is defined as follows:

$$G_p = \{(X,v) \to H \mid X \in N \wedge H \in R^X \wedge \forall K \in L_{G_p}(H).\ \eta(K) = v\}.$$

This definition implies that indeed for any $(X,v) \in N_{\Sigma p}$ and $K \in L_{G_p}((X,v)^\bullet)$ it holds that $\eta(K) = v$.

To show that G and G_p are quasi-equivalent it is sufficient to prove the following equivalence from which the existence of both γ-functions can be easily derived.

$$\forall H \in \mathrm{HG}_{T_\Sigma}.\ (\exists X \in N : X^\bullet \Rightarrow_G^* H) \Longleftrightarrow (\exists Y \in N_p : Y^\bullet \Rightarrow_{G_p}^* H)$$

We prove the above statement by proving both directions independently:

$\forall H \in \mathrm{HG}_{T_\Sigma}.\ (\exists X \in N : X^\bullet \Rightarrow_G^* H) \Longleftarrow (\exists Y \in N_p : Y^\bullet \Rightarrow_{G_p}^* H)$:

Let $(X,v)^\bullet[K_1/e_1]\dots[K_n/e_n] = H$ be a derivation for $H \in \mathrm{HG}_{T_{\Sigma_p}}$ in G_p. We obtain the derivation $X[K_1'/e_1']\dots[K_n'/e_n'] = H$ in G by simply replacing every occurring edge labelled by a nonterminal $(X,v) \in N_p$ by the corresponding nonterminal in N, i.e., $K_i' = (V_{K_i}, E_{K_i}, \mathrm{lab}_{K_i'}, \mathrm{att}_{K_i}, \mathrm{ext}_{K_i})$, where $lab_{K_i'}(e) = lab_{K_i}(e)$ if $lab_{K_i}(e) \in T_\Sigma$ and $lab_{K_i'}(e) = X$ if $lab_{K_i}(e) = (X,v)$. The resulting sequence indeed describes hyperedge replacements, since the rank of edges and hypergraphs are the same in both alphabets. Furthermore the replacement sequence is a derivation in G, as each rule $(X,v) \to t(H) \in G_p$ is a copy of a rule $X \to H \in G$ except for nonterminal relabelling.

$\forall H \in \mathrm{HG}_{T_\Sigma}.\ (\exists X \in N : X^\bullet \Rightarrow_G^* H) \Longrightarrow (\exists Y \in N_p : Y^\bullet \Rightarrow_{G_p}^* H)$:

Let $X^\bullet[K_1/e_1]\dots[K_n/e_n] = H$ be a derivation in G. We construct a corresponding derivation $(X,v)^\bullet[K_1'/e_1']\dots[K_n'/e_n'] = H$ in G_p. Analogously to the first part, we use graphs $K_i' = (V_{K_i}, E_{K_i}, \mathrm{lab}_{K_i'}, \mathrm{att}_{K_i}, \mathrm{ext}_{K_i})$ but here we have to determine the corresponding property for each nonterminal edge. Note that within the derivation each introduced nonterminal edge $e_j' \in E_{K_i'}^N$ will be replaced later on as the resulting hypergraph H is a terminal graph, i.e., $H \in \mathrm{HG}_{T_\Sigma}$. We will give e_j' an appropriate label $(lab_{K_i}(e_j), p(K_j'))$ such that the jth replacement can be properly realised, i.e., $lab_{K_i'}(e_j') = (\mathrm{lab}_{K_i}(e), p(K_j'))$,

thus all nonterminal edges $e \in K_i'$ are labelled with corresponding nonterminals from N_p. The valuations of all K_i' are determined from right to left, starting with the terminal graph K_n'. As each rule in G has a copy in G_p for any combination of edge labels extended by properties and as $lab(e_i) \rightarrow K_i \in G$, there is also a rule $\mathrm{lab}(e_i) \rightarrow K_i$ for each $i \in [1, n]$. □

Note that the resulting grammar G_p is not necessarily productive as it could contain nonterminals representing a property that is not derivable. As stated in Lemma 2.5 an equivalent productive grammar can be constructed.

Given the splitting grammar we can easily determine which values can be fulfilled by deriving terminal graphs from a nonterminal graph H by simply evaluating each graph in $t(H)$.

Corollary 2.2 (Deciding composed properties)
Given $H \in HG_\Sigma$, $G \in HRG_\Sigma$ and compositional property (η, V), the set $\eta(L_G(H)) \subseteq V$ can be computed.

2.7 Conclusions

In this chapter we introduced hypergraphs and hyperedge replacement grammars, which constitute the foundations of our framework. We use HRGs to model data structures and HGs to represent (abstract) heap structures. Abstracted heap structure are modelled by hypergraphs with nonterminal edges. Our concretisation function will be based on derivations, i.e. applications of grammar rules, while the abstraction function relies on the backward application of rules. In Section 2.4, we introduced the backward application as a proper operation on HGs. Within our framework abstract heap structures represent all derivable structures.

In Section 2.5 we considered inherent properties of HRGs. These will be crucial for proofs in later chapters. In particular the decomposition lemma (Lemma 2.4) as well as its corollary language decomposition are important. They allow us to divide problems in smaller entities and combine the individual solutions. This will be useful in various proofs. The compositional properties (Definition 2.16) introduced in Section 2.6 are based on these results.

Reconsider Lindstrom's algorithm introduced in Section 1.1. As mentioned we aim in verifying this algorithm for inputs that are binary trees. In Example 2.4 we introduced a suitable grammar to describe arbitrary binary trees that we could use for the analysis. Note that this grammar is not backward confluent as indicted by the not joinable critical pair depicted in Figure 2.12 on page 35.

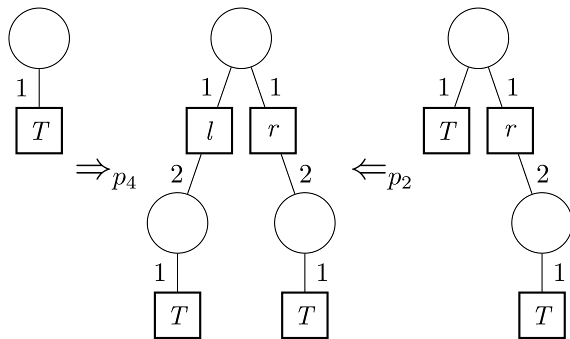


Figure 2.12: A critical pair.

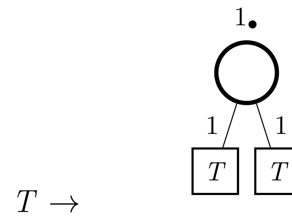


Figure 2.13: Additional T-rule.

In Figure 2.13 on page 35 an additional grammar rule is given which, added to the tree grammar, yields backward confluence. Unfortunately the resulting grammar is not equivalent. Note that at a vertex connected to a T-labelled edge arbitrary many new T-edges can be generated – and therefore also arbitrary many l- and r-edges which for sure is not intended by a grammar used to model data structures.

The backward confluence is not essential for our framework, but nice to have as this leads to unique normal-forms for the abstraction. However, as we will see in Section 3.2 this grammar does not comply with the needs of our framework. We will introduce an alternative grammar in the conclusion of the succeeding Chapter 3 in Section 3.6.

Something we did not consider in this chapter is that expressiveness of HRGs is restricted to languages of *bounded tree-width*. This restriction has a direct effect on our framework as any used data structures has to share the same property and thus data structures such as DAGs (directed acyclic graphs), grid structures or arbitrary graphs are not supported.

Chapter 3
Abstract Heap Representation

In this chapter we introduce the basic idea of using hyperedge replacement grammars as a base for abstracting heaps. We start in Section 3.1 with the representation of heaps. We first consider concrete representations of heaps by terminal hypergraphs and later abstract representations by nonterminal hypergraphs. Using nonterminal hypergraphs as abstract representation the idea is to realise stepwise concretisations via application of production rules, while applying grammar rules backwards results in an abstraction of the representation.

To ensure the correctness of the abstraction, the HRGs have to fulfil certain properties. These are introduced in Section 3.1. This will lead us to the definition of *heap abstraction grammars*, which impose properties on the hyperedge replacement grammars necessary to ensure that abstracting and concretising heap structures results in correct heap representations.

In Section 3.4 we extend the concretisation steps to *local concretisations*, that allow us to concretise around selected vertices. To realise this we introduce a generalisation of the Greibach Normal Form for string grammars that we call local Greibach Normal Form.

In Section 3.5 we introduce transition systems where states are hypergraphs that are connected via transitions. We define an over-approximation of a transition systems and discuss how to get transition systems from transition functions.

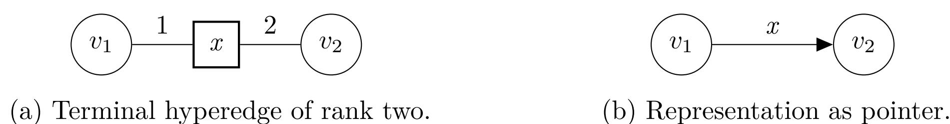
In this chapter we uniquely focus on the representation of heaps and the formal definition of transition systems. We will not consider heap programs and their execution but we develop the basics for proving the correctness of our approach. Later in Chapter 5 we will consider Java programs and give abstract semantics for Java bytecode in form of a transition function.

3.1 Heap Representation

Formally a heap is a set of locations, each containing a set of labelled references (pointers) to locations, i.e. a heap is a directed (labelled) graph. We model heaps by (terminal) hypergraphs, where each vertex represents a location and edges model pointers. The nature of pointers induces some restrictions to the hyperedges and hypergraphs that are used for heap representations. As we use (terminal) edges to represent pointers of the heap, they need to be of rank two, i.e. they always connect two vertices. We interpret terminal edges as pointers pointing from the first incident vertex to the second one. For the sake of readability we depict them as directed edges.

Example 3.1: Pointer representation

Figure 3.1(a) depicts an terminal edge labelled with x. We interpret this hyperedge as a directed edge (pointer) from the vertex v_1 connected by the first tentacle to the one connected by the second tentacle v_2. This interpretation is reflected in Figure 3.1(b) where the edge is depicted as a pointer. In the following we will use this visualisation for terminal hyperedges.



(a) Terminal hyperedge of rank two. (b) Representation as pointer.

Figure 3.1: Terminal hyperedge of rank two depicted as directed edge.

As we interpret terminal edges as pointers directed from the first to the second tentacle, we can also define the set of outgoing edges of a vertex as the set of edges that are connected to the vertex through their first tentacle.

Definition 3.1 (Outgoing edges):

Given a terminal hypergraph $H \in HG_{T_\Sigma}$ and a vertex $v \in V_H$, the set of outgoing edges *$out_H(v)$ of v is defined as:*

$$out_H(v) = \{e \in E_H \mid \text{att}(e)[1] = v\}.$$

We write $out(v)$ instead of $out_H(v)$ if H is clear from the context.

For each location and pointer label in a heap there is no more than one outgoing pointer. That is why we also need the heap graph to have only one pointer for each label and vertex. Thus we get the following definition of a heap.

Definition 3.2 (Concrete heap graph):
We call a terminal hypergraph $H \in HG_{T_\Sigma}$ a (concrete) heap graph *iff*

1. *each edge is terminal and of rank two*
2. $\forall v \in V_H.\ \forall e_1, e_2 \in out(v).\ lab(e_1) = lab(e_2) \Rightarrow e_1 = e_2$.

Example 3.2: Concrete heap graph
In Figure 3.2 a heap graph is given. Here we have a doubly linked list, where each element of the list has a pointer to a binary tree. The labels n *and* p *are the short form of next and previous representing pointers to the successor, respectively predecessor, within the doubly linked list. The pointer* t *points to the tree vertex that is assigned to the list object. Tree vertices have a right and left successor reachable via the* r*, respectively* l*, pointer.*

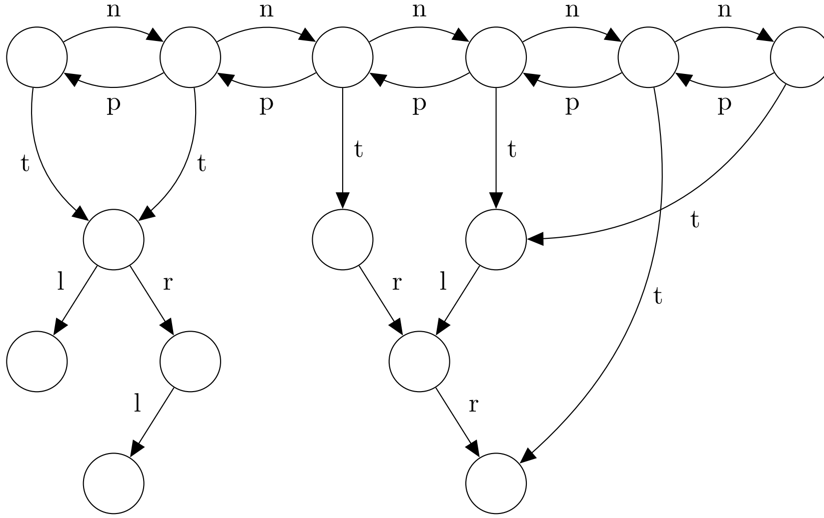


Figure 3.2: A concrete heap graph.

Note that in Example 3.2 we distinguish list and tree vertices by considering the labels of their outgoing edges. Thus the labels of the outgoing edges define the type $type_H(v) = \text{lab}(\text{out}_H(v))$ of a vertex v. The type of list vertices is a subset of $\{n, p, t\}$, while the type of tree vertices is a subset of $\{l, r\}$. While we can distinguish these two types of vertices, we do not distinguish them within the formal definition. In Chapter 4 we extend the graph model by *types* allowing to define a fixed type for each vertex.

For any heap graph H and any vertex $v \in V_H$, the labels of outgoing edges are pairwise distinct. Therefore there is a unique a-successor of vertex v for each label $a \in type_h(v)$, i.e. list vertices have n-, p- and t-successors while tree vertices have l- and r-successors.

Definition 3.3 (Successor):
*Given a hypergraph $H \in HG_\Sigma$, a vertex $v \in V_H$ and $a \in T_\Sigma$, the a-*successor *of the vertex v is defined as*

$$suc_H(v,a) = \begin{cases} \text{att}(e)[1] & \textit{if } \exists e \in out(v).\text{lab}(e) = a \\ \bot & \textit{else} \end{cases}$$

This is well-defined, since $out(v)$ contains at most one edge e labelled with a.

So far we only considered concrete heap representations, representing exactly one heap. An *abstract heap graph* is a representation that covers a (potentially infinite) set of heap graphs. We use nonterminal graphs, i.e. hypergraphs containing nonterminal edges to represent abstract heap graphs and define their sets of represented heap graphs as the language defined by the heap graph over some grammar.

Definition 3.4 (Abstract heap graph):
Given a grammar $G \in HRG_\Sigma$ an abstract heap graph *is a hypergraph $H \in HG_\Sigma$ with*

$$K \in L_G(H) \Rightarrow K \textit{ is a concrete heap graph.}$$

As we will see in Lemma 3.1 it is decidable if a hypergraph is an abstract heap graph. An abstract heap graph contains concrete and abstract parts. While vertices and terminal edges of an abstract heap graph are present in any of the potentially infinite many represented heaps, nonterminals represent separated parts of the heap that are connected to the incident edges. However, as stated by the decomposition lemma (Lemma 2.4) the abstract parts do neither depend on the concrete nor on the abstract context.

Note that the definition of abstract heap graphs includes also concrete heap graphs $H \in \text{HG}_{T_\Sigma}$. In this case $L(H) = \{H\}$, thus a concrete heap graph represents only a single heap.

Example 3.3: Abstract heap graph
In Figure 3.3 an abstract heap graph is given. Considering that from a DL-labelled nonterminal edge arbitrary doubly linked lists can be derived, for which the head is at the tentacle $(DL, 1)$ and the tail is at $(DL, 2)$, the heap graph represents any doubly linked list with size at least five. The grammar for doubly linked lists (and thus for the nonterminal DL) can be found in Section 3.3.

Given a hyperedge replacement grammar G and an abstract heap graph $H \in \text{HG}_\Sigma$ we call a vertex $v \in V_H$ *concrete* if no further outgoing terminal edges can be derived at v,

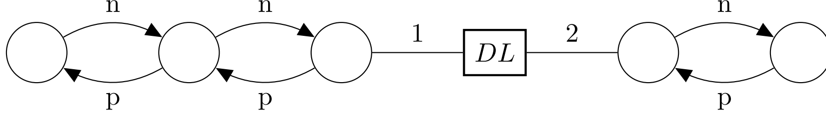


Figure 3.3: A abstract heap graph.

i.e. for any derivable terminal graph $H' \in L_G(H)$ the outgoing terminal edges of v are the same in H as in H':

$$\text{out}'_H(v) = \{e \in E^T_H \mid \text{att}(e)[1] = v\},$$

otherwise we call the vertex *abstract*. In the above example the first two and the last vertex (from left to right) are concrete. The third and fourth one are abstract as we can derive an additional outgoing n-edge and p-edge respectively. Note that any abstract vertex is incident to a nonterminal edge, while not any vertex incident to an nonterminal edge is abstract.

Lemma 3.1 (Decidability of abstract heap graphs):
Given a grammar $G \in HRG_\Sigma$ and a hypergraph $H \in HG_\Sigma$, it is decidable whether H is an abstract heap graph.

Before we prove the above lemma we introduce the label set of tentacles, i.e. the label set contains all possible sets of outgoing edges derivable at a given tentacle.

Definition 3.5 (Tentacle label set):
Given $G \in HRG_\Sigma$ the outgoing label set of a tentacle type $(X, i) \in Tent_\Sigma$ is the set

$$outLabel_G(X, i) = \{\text{lab}_H(\ out_H(ext_H[i])\ ) \mid H \in L_G(X^\bullet)\} \subseteq \mathcal{P}(\Sigma).$$

We call a tentacle (X, i) a reduction tentacle, *if $outLabel_G(X, i) = \{\emptyset\}$, i.e. there are no outgoing edges derivable at the tentacle (X, i).*

Note that for every terminal label $a \in T_\Sigma$ the sets $\text{outLabel}_G(a, 1) = \{\{a\}\}$ and $\text{outLabel}_G(a, 2) = \{\emptyset\}$ are singletons, as $L(a^\bullet) = \{a^\bullet\}$. This is not the case for nonterminal tentacles in general. However, we can determine a quasi-equivalent grammar G_t such that for any $X \in \Sigma$ and $1 \leq i \leq \text{rk}(X)$, $\text{outLabel}_{G_t}(X, i)$ is a singleton. To construct G_t we define the compositional property $t = (\eta, V)$ where $V = \mathcal{P}(T_\Sigma)^\star$ and η maps any terminal graph $H \in \text{HG}_{T_\Sigma}$ to $v_1 \dots v_n$ where $n = |\text{ext}_H|$ and for $1 \leq i \leq n$ $v_i = \text{lab}_H(\text{out}_H(ext_H[i]))$. Note that the set V is not finite as needed for a compositional property (see Definition 2.16 on page 32). However, as long as we consider only the rule graphs of G, we can restrict the set V to sequences no longer than the maximum

rank of nonterminals in N_Σ. Let $G_t \in \mathrm{HRG}_{\Sigma_t}$ be the splitting grammar for $t = (\eta, V)$. Obviously it holds that for any $(X, v) \in N_{\Sigma_v}$, $K \in L_{G_t}((X,v)^\bullet)$ and $1 \leq i \leq \mathrm{rk}((X,v))$ that $\eta(K)[i] = \mathrm{outLabel}_{G_t}((X,v), i)$. As $G \approx G_t$, $H \in \mathrm{HG}_\Sigma$ is an abstract heap graph if and only if $\gamma_{G \to G_t}(H)$ is an abstract heap graph. Note that in $H \in \mathrm{HG}_{\Sigma_t}$ for any vertex v the set of derivable outgoing terminal edges is uniquely defined, simplifying the checking if a graph is a heap graph. We use the quasi-equivalent grammar $G_t \in \mathrm{HRG}_{\Sigma_t}$ for the proof of Lemma 3.1.

Proof. Following Definition 3.2 we check if for a given $H \in \mathrm{HG}_\Sigma$ any $K \in L_G(H)$ is a heap graph. We do so by checking if any $K \in L_{G_t}(H')$, with $H' = \gamma_{G \to G_t}(H)$, fulfils the following two properties:

1. each edge is terminal and of rank two
2. $\forall e_1, e_2 \in \mathrm{out}(v).\ lab(e_1) = lab(e_2) \Rightarrow e_1 = e_2$.

As by definition, $L_{G_t}(H')$ contains only terminal graphs, it suffices to ensure that any terminal label is of rank two. This is rather a property of the alphabet than of the graph and can easily be checked.

For the second property we have to check that any vertex within the graph has no two outgoing edges with the same label. Any vertex in $K \in L_{G_t}(H')$ is either a vertex from $V_{H'}$ or a copy of a vertex from one of the rule graphs used to derive the graph from H'. We now check for any vertex in any $K \in L_{G_t}(H')$ whether it fulfils the necessary property. Therefore we collect all nonterminals that could appear during derivations of terminal graphs. This set of nonterminals defines a set of rules that could be involved in a derivation from H'. We then check for each of their rule graphs if for any occurring vertex the property is fulfilled.

To collect the derivable nonterminals within a set $R \subseteq N_\Sigma$ we simply initialise the set R with all nonterminals occurring in H ($R = E_H^N$). Then we incrementally add nonterminals occurring in the right-hand sides of grammar rules for nonterminals from the set R: $R' = \{X \in E_K^N \mid \exists Y \in R. Y \to K \in G\}$. We repeat this update as long as we get new nonterminals in R. This procedure terminates as R and G are finite, and yields the set R containing exactly the nonterminals, that can be involved in derivations of H. Given the set R we consider the set of reachable hypergraphs $R_H = \{H' \mid X \to H' \in G_t \wedge X \in R\} \cup \{H\}$ and check for any vertex within theses graphs that no two outgoing edges with the same label can be derived by checking if the outLabel-sets of connected tentacles are disjoint

$$\forall H \in R_H,\ v \in V_H, t_1, t_2 \in \mathit{Tent}(v).\ \mathrm{outLabel}_{G_t}(t_1) \cap \mathrm{outLabel}_{G_t}(t_2) = \emptyset.$$

If this holds for any vertex we obviously cannot derive two equally labelled outgoing edges at one vertex. On the other hand if there is a vertex where this does not hold there is a derivable hypergraph containing a vertex with two equally labelled outgoing edges.

□

The proof is based on the assumption that the grammar (or at least the involved rules) is productive. This not guaranteed in general but by Lemma 2.5 we know that any grammar can be transferred into an equivalent productive.

3.2 Heap Abstraction Grammars

For our abstraction approach we will utilise the concretisation strategy by forward rule application and abstraction as reverse operation as explained in the previous section (from now on simply referred to as concretisation and abstraction). We will see that concretisation and abstraction given by arbitrary HRGs may lead to unexpected behaviour. In order to solve this problem, we restrict HRGs to so-called *heap abstraction grammars* by adding three necessary properties – productiveness, increasingness, typedness.

Definition 3.6 (Heap abstraction grammar):
A hyperedge replacement grammar $G \in HRG_\Sigma$ is a heap abstraction grammar *iff*

1. *for any $X \to R \in G$ the graph R is an abstract heap graph,*
2. *G is productive,*
3. *G is increasing,*
4. *G is typed.*

We denote the set of all heap abstraction grammars over Σ by HAG_Σ.

The first three properties are discussed shortly within the next paragraph while the remainder of this section is devoted to the fourth property.

The first property is essential as otherwise nonterminals introduced by abstraction could be derived by later concretisations to graphs that are not heap graphs. Productiveness (Definition 2.13) ensures that from any abstract heap graph a concrete heap graph can be derived, i.e. given a grammar $G \in \mathrm{HAG}_\Sigma$ we ensure for any $X \in N_\Sigma$ that $L_G(X^\bullet) \neq \emptyset$. Increasingness (Definition 2.14) is needed to ensure the termination of the abstraction. As a counterexample consider a production rule of the form $X \to X^\bullet$. When repeatedly applying this rule in a backward fashion, the abstraction process does not terminate. On the other hand increasingness ensures that concretisation steps indeed materialise vertices or at least terminal edges.

If the fourth property is not given, applying grammar rules backward and forward in an alternating fashion, potentially yields no heap graphs. The following example illustrates this.

Example 3.4: Need for typing

Consider the HRG G for binary trees from Example 2.4 depicted in Figure 2.3 on page 21 and the heap graph in Figure 3.4 (left). Abstraction using rule p_3 from Figure 2.3 yields the heap graph in Figure 3.4 (middle). A successive concretisation by applying rule p_4 from Figure 2.3 results in a graph containing a vertex with two outgoing l-labelled terminal edges, which is not a heap graph, see Figure 3.4 (right).

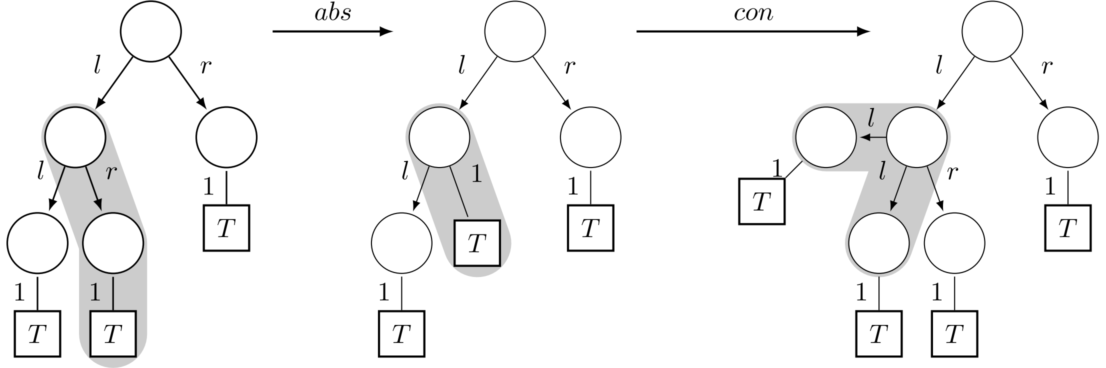


Figure 3.4: Abstraction and concretisation yielding an invalid heap graph.

This can be avoided by imposing typing restrictions. A grammar is typed if every concrete vertex has a well-defined type which is induced by its outgoing edges.

Definition 3.7 (Typing):
$G \in HRG_\Sigma$ is typed *iff for any tentacle $(X, i) \in Tent_\Sigma$, there exists a* $\text{type}(X, i) \subseteq \Sigma$ *such that $outLabel_G(X, i) = \{\text{type}(X, i)\}$.*

Let $G \in \text{HRG}_\Sigma$ be typed. Given $H \in \text{HG}_\Sigma$, for any vertex in H the set of outgoing terminal labels is uniquely defined and neither abstraction nor concretisation can alter this set. Note that this is not the case for untyped grammars. Here the number of possible sets of outgoing tentacles can decrease by concretisation, while abstraction potentially increases the number of possible sets (as Example 3.4 shows).

In the proof of Lemma 3.1 we use a splitting grammar G_t with the special property that $\text{outLabel}_G(X, i)$ is a singleton for any tentacle $(X, i) \in Tent_\Sigma$. Obviously this splitting grammar is typed.

Corollary 3.1 (Construction of typed grammars)
It is decidable whether a HRG is typed. For every HRG a quasi-equivalent typed HRG can be constructed.

In Example 3.4 we noticed that grammar from Example 2.4 for binary trees is not typed. A quasi-equivalent typed grammar for binary trees can be found in the following section.

Combining Lemma 3.1, Lemma 2.5, Lemma 2.6 and Corollary 3.1 we get the following theorem.

Theorem 3.1 (Expressiveness of heap abstraction grammars):
For any HRG a quasi-equivalent HAG can be constructed.

3.3 Example Heap Abstraction Grammars

Now that we have defined how to describe heap structures by grammars we give some examples of common data structures via heap abstraction grammars.

Linked List

A linked list consists of a sorted list of nodes where each node is connected to its successor by a next edge (for readability labelled by n). In Figure 3.5 a grammar is given defining linear lists recursively. A linked list consists of exactly two locations and a pointer pointing from the first to the second or a location with a next pointer pointing to a linked list.

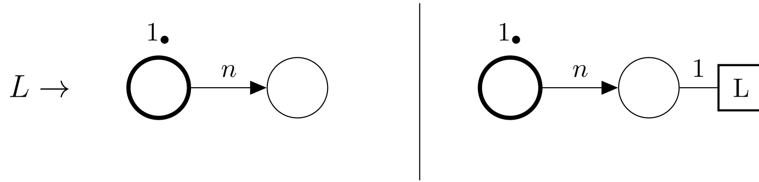


Figure 3.5: Grammar for linked lists.

The nonterminal L has rank one. L is productive and any rule is increasing. The type of the single nonterminal tentacle $(L, 1)$ is $\{n\}$.

Doubly Linked List

A doubly linked list is essentially a linked list with additional pointers from any node to its predecessor. We extend the grammar given in Figure 3.5 by a p-edge between nodes connected by an n-edge. In addition we add an extra external node, such that the grammar describes at partial list between two nodes. The resulting grammar is given in Figure 3.6.

The nonterminal DL hast two tentacles $(DL, 1)$ with type$(DL, 1) = \{n\}$ and $(DL, 2)$ with type$(DL, 2) = \{p\}$. The grammar is productive and increasing.

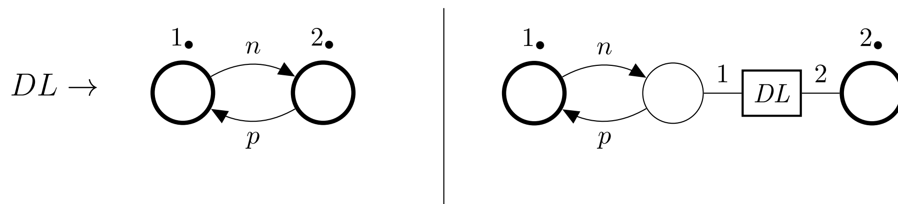


Figure 3.6: Grammar for doubly linked lists.

Fully Branched Binary Tree with Linked Leaves

A binary tree with linked leaves is a fully branched binary tree with special leaf nodes that have an n-pointer to the successive leaf, thus the leaves form a linked list. The corresponding grammar consists of four rules for a single nonterminal LL of rank three. The grammar rules are given in Figure 3.7.

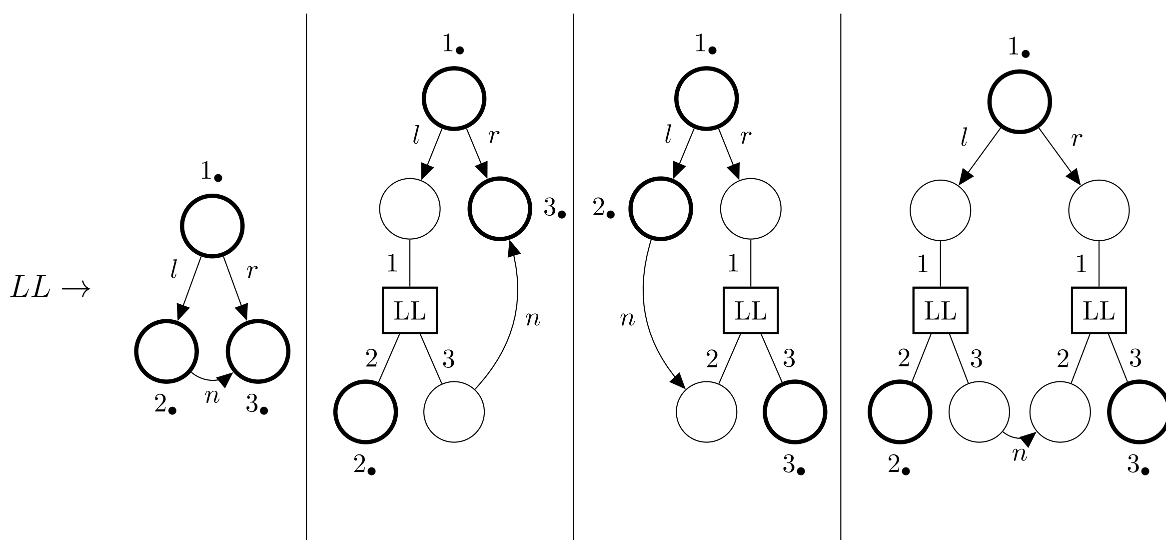


Figure 3.7: A HAG for binary trees with linked leaves.

Here $type(LL, 1) = \{l, r\}$, type$(LL, 2) = \{n\}$ and type$(LL, 3) = \emptyset$.

Partially Branched Binary Tree

A grammar for binary trees is given in Figure 2.3, however, this grammar is neither increasing nor typed. A quasi-equivalent grammar is given in Figure 3.8 and Figure 3.9. To achieve typing, nonterminal T is splited into three nonterminals T_l, T_r and T_{lr}, resulting in 24 rules.

Here $type(T_l, 1) = \{l\}$, type$(T_r, 1) = \{r\}$ and type$(T_{lr}, 1) = \{l, r\}$. The grammar is typed and increasing. However, the size of the grammar is substantial. In Chapter 4 we will add an explicit null-node which allows us to give a grammar for binary trees that is typed and contains only four rules (Figure 5.2 on page 99).

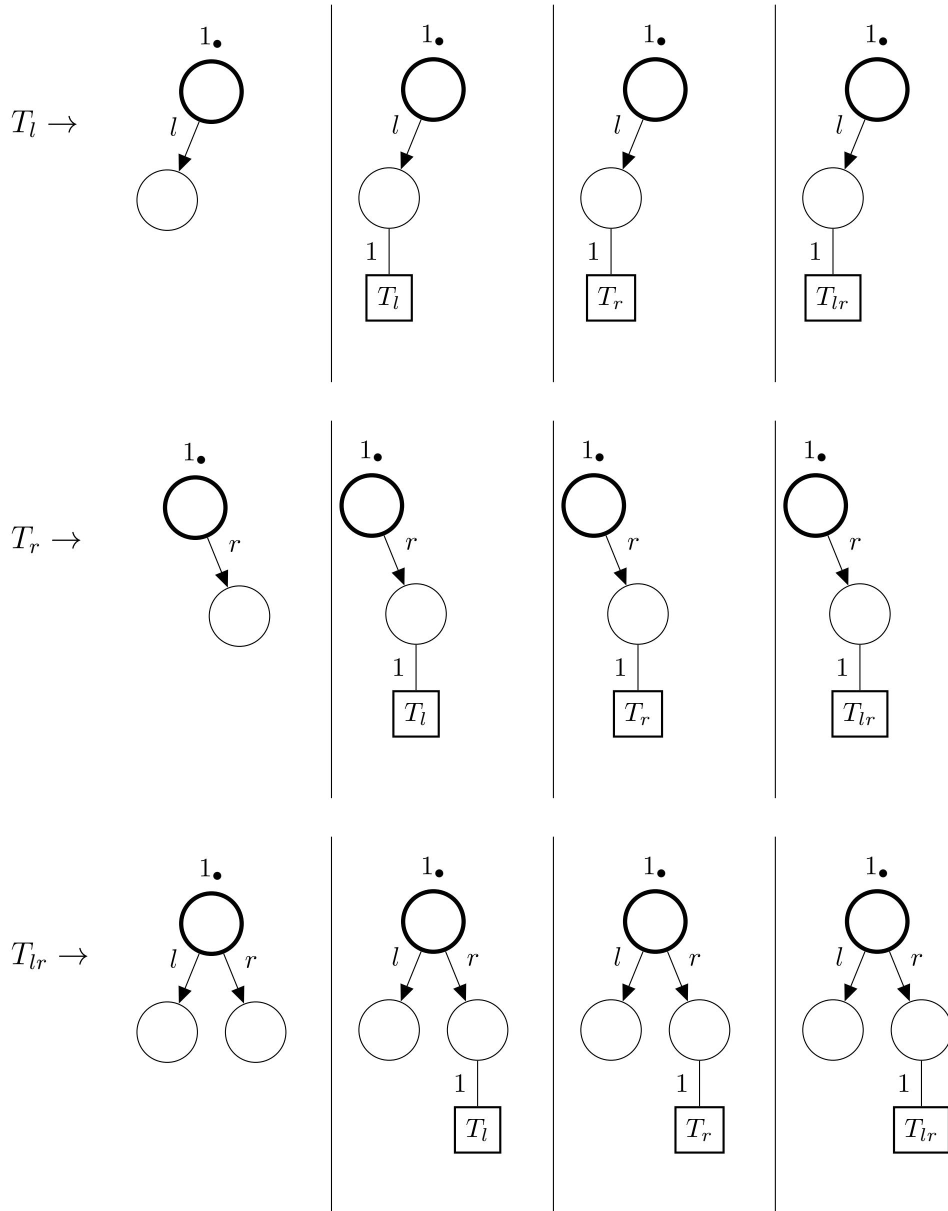


Figure 3.8: A HAG for binary trees (1).

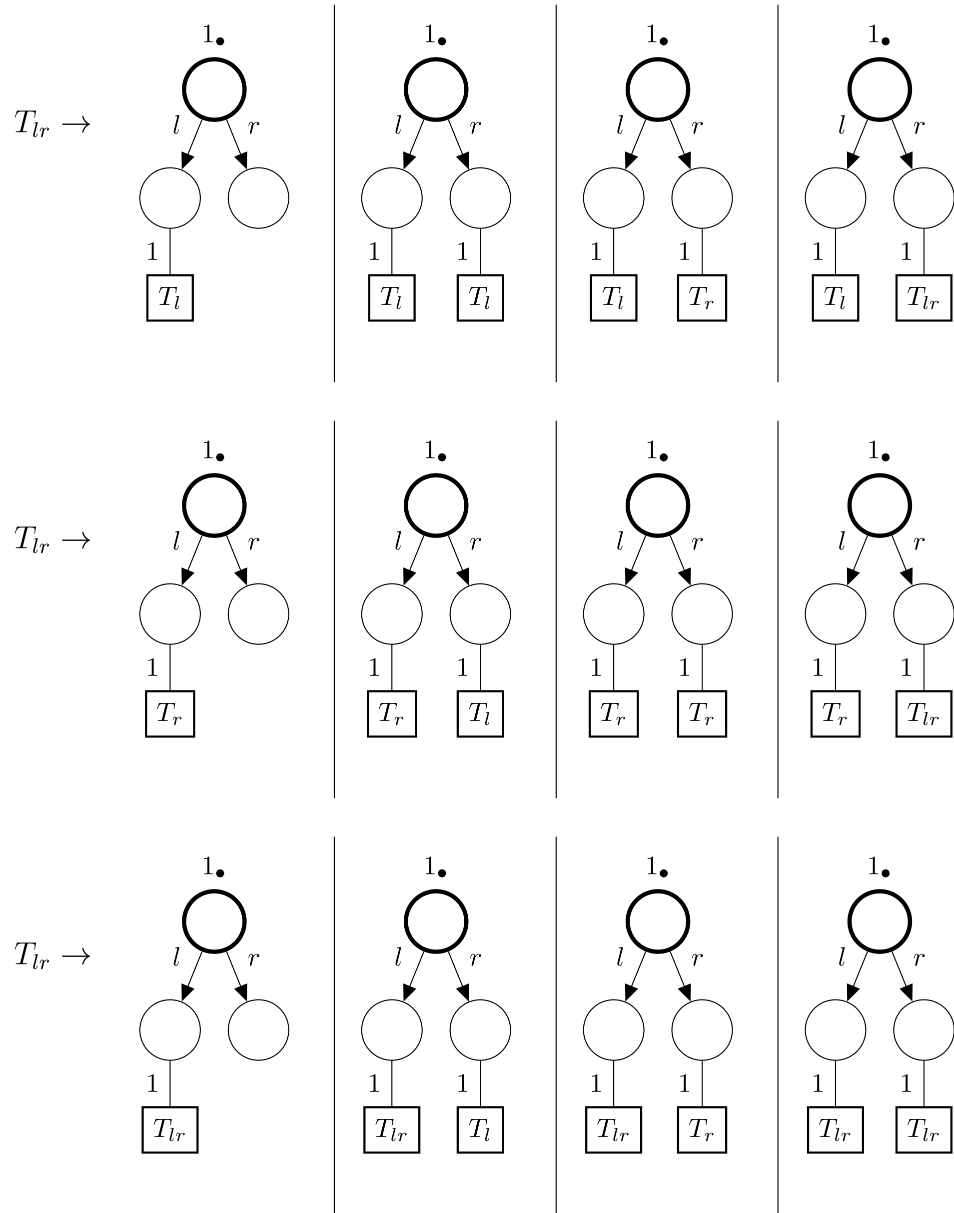


Figure 3.9: A HAG for binary trees (2).

3.4 Local Concretisation

So far we concretised abstract heap graphs in a (somehow) arbitrary way by replacing some nonterminal edges. To enable the abstraction by hyperedge replacement grammars for the analysis of heap-manipulating programs we need a possibility to realise concretisations locally. We distinguish concrete from abstract vertices, where for an abstract vertex not all of its successors are accessible, i.e. some of them are abstracted by nonterminal edges. We want to be able to concretise these successors locally. This kind of concretisation is necessary if we need to update or access the successors of an abstract vertex, due to executing a program statement.

We consider vertex concretisation in Section 3.4.1. We will see that we need special grammars to make the concretisation work in general. For this aim we introduce the *local Greibach Normal Form* for hyperedge replacement grammars in Section 3.4.2, that is oriented at the Greibach Normal Form for string grammars. We give a construction for bringing arbitrary heap abstraction grammars into local Greibach Normal Form and use these in Section 3.4.3 to define a (directed) concretisation function for heap graphs.

3.4.1 Concretisation of Vertices

Our aim is to define a concretisation function that given an abstract heap graph and one of its abstract vertices returns a set of heap graphs for which the vertex is concrete and which together describe the same concrete heaps as the original one. To concretise an abstract vertex we replace the nonterminal edges that abstractly represent the successors of the vertices. Given a heap abstraction grammar any edge connected to the vertex via a non-reduction tentacle abstracts at least one of the successors, i.e. we have to derive these edges to get the successors. The following example illustrates the concretisation of an abstract vertex.

Example 3.5: Local concretisation
Figure 3.10 specifies an hyperedge replacement grammar for doubly-linked lists where each element, beside the next-(n) and previous-pointer (p) has an additional pointer list (l) to a common shared object. Note that $(L, 3)$ *is a reduction tentacle. This grammar can be used to model doubly-linked lists as they often occur in practise. A list is an object containing a head- and tail-pointer pointing to the first, respectively last element of a list of elements linked via next- and previous pointer. The list elements have an additional pointer to the enclosing list object.*

The heap graph in Figure 3.11(a) represents such a list with at least two elements. Vertex v_3 *is concrete as the only connected nonterminal tentacle is a reduction tentacle. The other two vertices are abstract as for* v_1 *an outgoing next-pointer and for* v_2 *an outgoing previous-pointer can be derived. We concretise the vertex* v_1 *by*

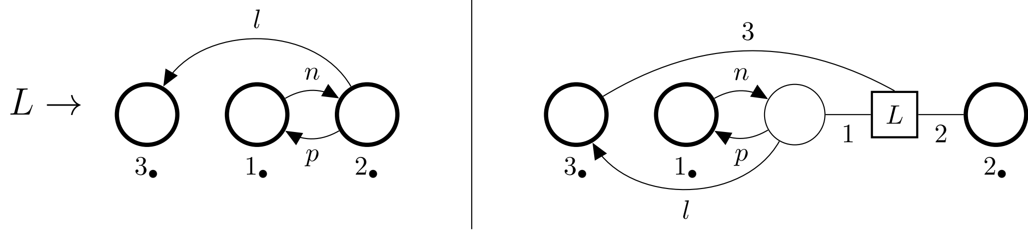
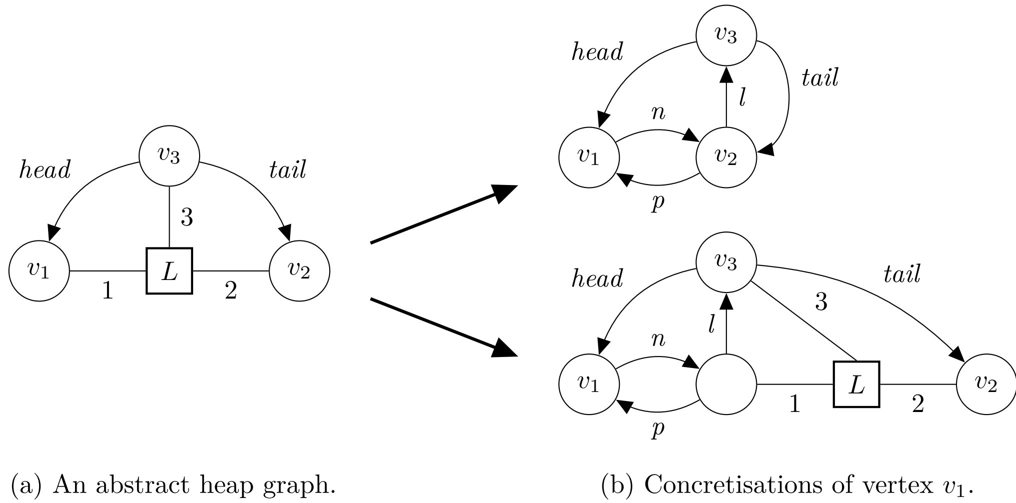


Figure 3.10: A grammar for doubly-linked lists with shared object.

applying one of the production rules for L. Both resulting heap graphs are depicted in Figure 3.11(b). While the upper one represents a list with two elements, the lower one represents all lists with at least three elements.



(a) An abstract heap graph.

(b) Concretisations of vertex v_1.

Figure 3.11: Concretisation of an abstract vertex.

In the above example the vertex v_1 is concrete in both obtained concretisations. The union of the languages represented by the two concretisations is exactly the one represented by the original heap graph from Figure 3.11(a). Thus we can use the concretisations instead of the original heap graph to represent the set of concrete heap graphs.

Theorem 3.2 (Completeness of concretisation):
For $G \in HRG_\Sigma, H \in HG_\Sigma, e \in E_H,\ X \in N_\Sigma$ with $X = lab(e)$:

$$L(H) = \bigcup_{X \to K \in G} L(H[K/e])$$

The above theorem follows directly from the associativity property (Lemma 2.3). It is essential for the correctness of concretisation as it states that instead of considering an

abstract heap graph we can also consider the set of heap graphs that we get by replacing one of its edges by any corresponding rule graph.

Unfortunately the approach for vertex concretisation given in Example 3.5 does not work in general. If we try to concretise vertex v_2 we get the same heap configurations, but the vertex v_2 remains abstract in the lower concretisation. Considering only the upper concretisation, where vertex v_2 is concrete, would restrict the set of represented heap graphs to lists with exactly two elements. Further replacements of edges now attached to the vertex v_2 would yield more concrete representations but would not lead to a concretisation of vertex v_2. This is because in the second rule for L the second external vertex is attached to a $(L, 2)$-tentacle. We say the rule is *locally recursive* at $(L, 2)$, as any replacement of an L-edge by this rule replaces an $(L, 2)$-tentacle by a further $(L, 2)$-tentacle. To concretise a vertex at a $(L, 2)$-tentacle the second external vertex of any L-rule graph must be concrete.

In general we need a grammar where for any rule graph any external vertex is concrete. Engelfriet introduced a Greibach Normal Form for *bounded hyperedge replacement grammars* that fulfils this property [Eng92]. A hyperedge replacement grammar is *bounded* if there is some bound $b \in \mathbb{N}$ such that for any vertex, in any derivable hypergraph, the number of connected edges is at most b. The reason for this restriction is that Engelfriet requires any terminal edge, derivable at some vertex, to be derivable by a single replacement. The one step concretisation of unboundedly many edges would imply rules with unboundedly many vertices. Boundedness is a real restriction for heap abstraction grammars. For example the grammar presented in Example 3.5 is not bounded as unboundedly many incoming l-edges are derivable at tentacle $(L, 3)$. While for heap graphs the degree of vertices is not bounded in general, the number of *outgoing edges* at each vertex is bounded by the number of terminal labels.

In the next section we present the *local Greibach Normal Form*, which can be established for any grammar with bounded *outdegree* which includes any typed HRG. In addition we need the Greibach property to be fulfilled only locally, i.e. for each rule only one of the external vertices must be concrete.

3.4.2 Local Greibach Normal Form

The *local Greibach Normal Form (LGNF)* for hyperedge replacement grammars is a generalisation of the *Greibach Normal Form (GNF)* for string grammars. Production rules for grammars in GNF are restricted to the form $X \to aN_1 \dots N_k$, such that using left derivations only, words $w \in \Sigma^n N_\Sigma{}^\star$ are derived in n steps. Thus terminal words are constructed from left to right extending a terminal prefix by one symbol each step. In the same manner derivations of a LGNF can be realised from one end of the graph to another, in contrast to [Eng92] where graphs are derived from outside to inside.

Definition 3.8 (Local Greibach Normal Form):
A heap abstraction grammar $G \in HAG_\Sigma$ is in local Greibach Normal Form (LGNF) *if for each non-reduction tentacle (X, i) it holds that $G \equiv G_{(X,i)} \cup G^{\overline{X}}$ with*

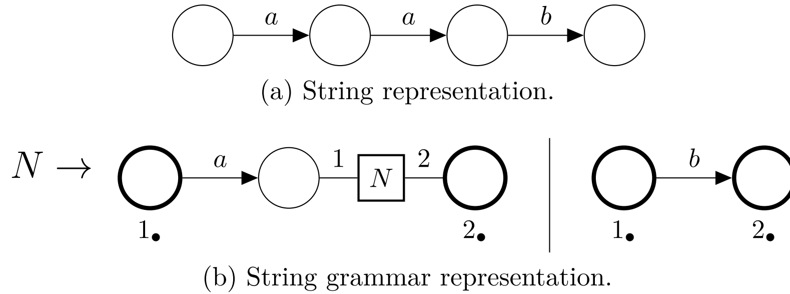
$$G_{(X,i)} = \{X \rightarrow R \in G^X \mid ext_R[i] \text{ is concrete}\}.$$

If $G \in \text{HAG}_\Sigma$ is in LGNF then for any $(X, i) \in Tent_\Sigma$ X-rules for which the ith external vertex is not concrete can be omitted without changing the language. Therefore we do not lose represented concrete heaps if we concretise a X-edge by rules from $G_{(X,i)}$ only. That is $G_{(X,i)}$ is the concretisation grammar for tentacle (X, i).

Whenever a heap abstraction grammar $G \in \text{HAG}_\Sigma$ is in LGNF we use $G_{(X,i)}$ to refer to the corresponding set from the definition.

As we mentioned before there is a strong connection between GNF and LGNF. Indeed LGNF is a generalisation of GNF. Strings can be uniquely represented by heap graphs containing chains of terminal edges only, production rules can be translated to heap abstraction grammars analogously [Hab92]. A string grammar is in GNF if and only if its representation as heap abstraction grammar is in LGNF.

Example 3.6:
In Figure 3.12 the graph representations for the word $w = aab$ (a) and the string grammar $N \rightarrow aN \mid b$ (b) are given. For any string grammar representation the nonterminals are of rank two and for each nonterminal X the second tentacle $(X, 2)$ is a reduction tentacle. Therefore given a string grammar representation in LGNF there is exactly one subgrammar for any nonterminal X namely $G_{(X,1)}$, which contains the rules corresponding to the rules of the string grammar in GNF.



(a) String representation.

(b) String grammar representation.

Figure 3.12: String graphs

Theorem 3.3 (Local Greibach Normal Form):
Any heap abstraction grammar $G \in HAG_\Sigma$ can be transformed into an equivalent heap abstraction grammar $G' \in HAG_\Sigma$ in LGNF.

We prove Theorem 3.3 by giving a corresponding construction. The LGNF grammar G is established by merging the corresponding sets $G_{(X,i)}$, constructed in four steps along the lines of the GNF construction for string grammars.

To generate grammar $G_{(X,i)}$ we have to replace rules where tentacle (X,i) is replaced by a nonterminal tentacle. For $p = X \to R \in G$ we denote by $(X,i) \to_p (Y,j)$ that tentacle (X,i) is replaced by a (Y,j)-tentacle, i.e. $ext_R[i]$ is attached to a (Y,j)-tentacle. Challenging are recursions, i.e. tentacles that can be replaced by themselves.

> **Definition 3.9** (Local Recursion):
> *Let $G \in HAG_\Sigma$, $X \in N_\Sigma$, $1 \leq i \leq \mathrm{rk}(X)$. G is* (directly) locally recursive *at (X,i) if there exists a rule p with $(X,i) \to_p (X,i)$.*

We call G *indirectly locally recursive* at (X,i) if it is not possible to replacing tentacle (X,i) by itself with a single rule application $(X,i) \to_p (X,i)$, but by the application of a sequence of rules $(X,i) \to_{p_1} \ldots \to_{p_n} (X,i)$ with $n > 1$.

Assuming a total order on the non-reduction nonterminal tentacles $T_1,\ldots,T_n \in Tent_{N_\Sigma}$, we start the construction at the lowest tentacle. For each tentacle we first (step 1) remove indirect local recursion by eliminating every rule p implicating $T_i \to_p T_j$ with $j < i$. After the elimination only direct local recursion remains, which then (step 2) is removed. Then (step 3) all rules can be brought into LGNF by just applying hyperedge replacements. In the last step (step 4) rules for nonterminals, newly introduced during the construction, are transformed. In the following we guide through the four construction steps and define them in detail.

Construction of LGNF.

For each non-reduction (X,i)-tentacle we initialise the set $G_{(X,i)} = G^X$ and assign an ordinal inducing an ordering $T_1,\ldots,T_n$ on the non-reduction tentacles T_i.

Step 1: Elimination of rules.

In the first step we remove indirect recursions by reducing them to direct recursions. We eliminate rules $p = X \to H \in G_{(X,i)}$ with $(X,i) = T_k \to_p T_l$, $l < k$. Let $T_l = (Y,j)$, $e \in E_H$ with $lab_H(e) = Y$ and $att_H(e)[j] = ext_H[i]$. Then we replace p by the set $\{X \to H[K/e] \mid Y \to K \in G_{(Y,j)}\}$. As we start the construction at the lowest tentacle, any rule p with $T_l \to_p T_m$, $m < l$ was removed from $G_{(Y,j)}$ before. Therefore we can eliminate all corresponding rules in finitely many steps. It follows from Theorem 3.2 that the elimination does not change the language.

> **Lemma 3.2** (Elimination of rules):
> *Let $G \in HAG_\Sigma$. For a grammar G' originating from G by eliminating a production rule as described above it holds that $G \simeq G'$.*

Step 2: Elimination of local recursion.

After removing indirect recursion in the first step, only *locally recursive* rules p with $T_i \rightarrow_p T_i$ remain. If some tentacle is locally recursive this is because graphs are build up from another tentacle. We remove direct recursion at a tentacle by adding new rules that mimic the behaviour of the rule and allow us to derive the same graphs in a mirrored fashion building up the graphs from the corresponding tentacle.

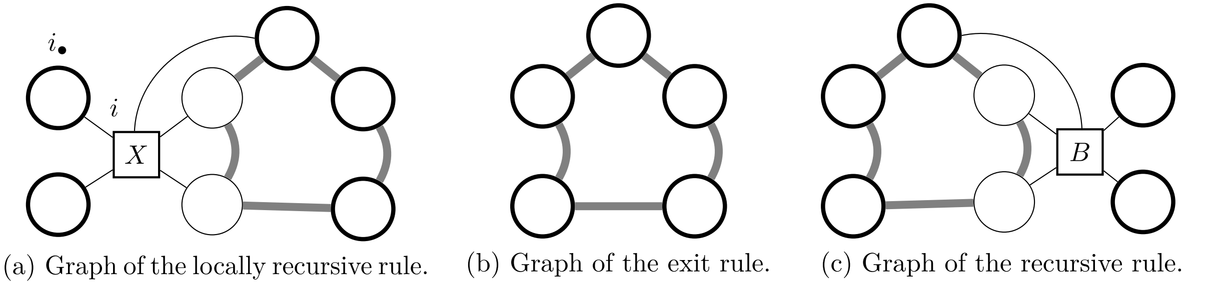
Let $G_r \subseteq G_{(X,i)}$ be the set of all rules locally recursive at (X, i). We remove local recursion in $p = X \rightarrow H \in G_r$ by introducing a new nonterminal B, as well as a corresponding recursive rule $B \rightarrow R$ and an exit rule $B \rightarrow E$. The graphs R and E are defined as follows.

The graph $E = (V_H \backslash V_e,\ E_H \backslash \{e\},\ \mathrm{lab}_H \restriction E_E,\ \mathrm{att}_H \restriction E_E,\ \mathrm{ext}_E)$ is obtained by removing edge e, causing the local recursion, from H. We remove external vertices singly connected to e ($V_e = \{v \in [ext_H] \mid \forall e' \in E_H : v \in [att_H(e')] \Rightarrow e = e'\}$). By removing e, the connected internal vertices move to the border of the graph and become external. Thus the set of external vertices is $[\mathrm{ext}_R] = ([att_{H_j}(e)] \cup [ext_{H_j}]) \cap V_E$. In Figure 3.13 the schema of a rule graph H (a) and the corresponding exit graph E (b) are given.

The rule graph $R = (V_E \cup [ext_R],\ E_E \cup \{e\},\ lab_E[e' \mapsto B],\ att_E[e' \mapsto plug],\ \mathit{fill})$ of the recursive rule extends E by an additional edge e' labelled by B. As this edge models the structure from the other side, it is connected to the remaining external vertices of H that are no longer external. Note that the rank of B is given by E and therefore the introduced gaps in the external sequence are filled by new external vertices that are connected to edge e' ($\mathit{fill}(i) = \mathrm{ext}_E(i)$ *if* $\mathrm{ext}_E(i) \in V_H$, *a new vertex otherwise*). To build up the same structure as (X, i) "from the other side", edge e' has to be plugged in correctly:

$$plug(g) = \begin{cases} \mathrm{ext}_H(y) & , \textit{if } \mathrm{ext}_R(g) \in V_R \\ \mathrm{ext}_E(g) & , \textit{otherwise} \end{cases} , \textit{ with } \mathrm{att}_H(e)(y) = \mathrm{ext}_E(g).$$

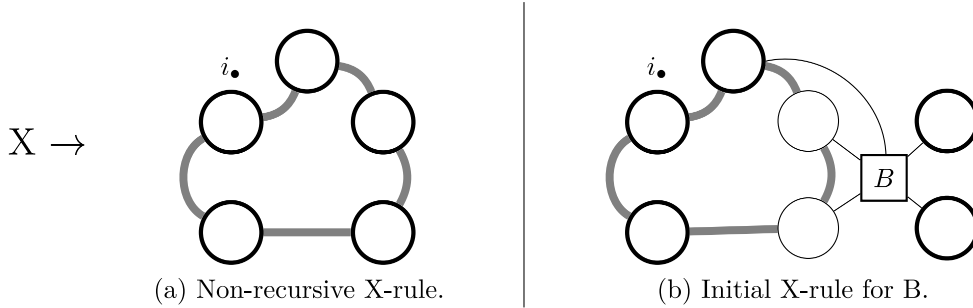
The construction of the graph R for the rule graph from Figure 3.13(a) is depicted schematically in Figure 3.13(c).



(a) Graph of the locally recursive rule. (b) Graph of the exit rule. (c) Graph of the recursive rule.

Figure 3.13: Schematic view of the second step.

The nonterminal B is collected together with the other newly introduced nonterminals in a set N' yielding the new alphabet $\Sigma' = \Sigma \cup N'$. Each (X, i)-recursive derivation has to be terminated by a non-recursive rule from $G_{(X,i)}$. Therefore the corresponding mirrored derivation of X has to be initialised by one of the non-recursive rules from X. We add to $G_{(X,i)}$ a copy of each non-recursive rule extended by an additional B-edge. We connect the B edge to the external vertices in the same manner as we added the B-edge in the graph R. See Figure 3.14 for a schematic representation.



(a) Non-recursive X-rule. (b) Initial X-rule for B.

Figure 3.14: Generation of initial rule for mirrored derivations.

Example 3.7:

The grammar for doubly-linked lists with the production rules $L \to H \mid T$ *given in Figure 3.10 on page 50 is locally recursive at* $(L, 2)$. *We introduce nonterminal* B *and the rules* $B \to E \mid R$, *as depicted in Figure 3.15. We get the terminal graph* E *from* T *by removing the* L*-edge and the second external vertex. The rule graph* R *of the recursive rule is* E *extended with an additional* B*-edge and a newly introduced external vertex* $\text{ext}[1]$. *The grammar* $G_{(L,2)}$ *consists of the terminal rule from* G *and an initial rule for mirrored derivations by* B. *Note that grammar* $G_{(B,2)}$ *is locally recursive at* $(B, 1)$.

After the second step any kind of recursion is removed. That is any non-reduction nonterminal tentacle remaining attached to the ith external vertex of one of the rule graph from $G_{(X,i)}$ has a higher order then (X, i). These remaining nonterminal tentacles are removed in the third step. The rules introduced for the new nonterminal B will be considered in the fourth step. Before we move on to the third step we prove, that the transformations of the second step do not alter the language of the grammar.

Lemma 3.3 (Elimination of local recursion):
Let $G \in HAG_\Sigma$. *For the grammar* $G' \in HAG_{\Sigma'}$ *originating from* G *by eliminating the local recursion at* (X, i), *it holds that* $G \equiv G'$.

We prove the equivalent statement that for any two hypergraphs $H, K \in \text{HG}_\Sigma$ we can derive K from H in G if and only if we can derive H from K in G':

$$(H \Rightarrow^*_G K) \iff (H \Rightarrow^*_{G'} K).$$

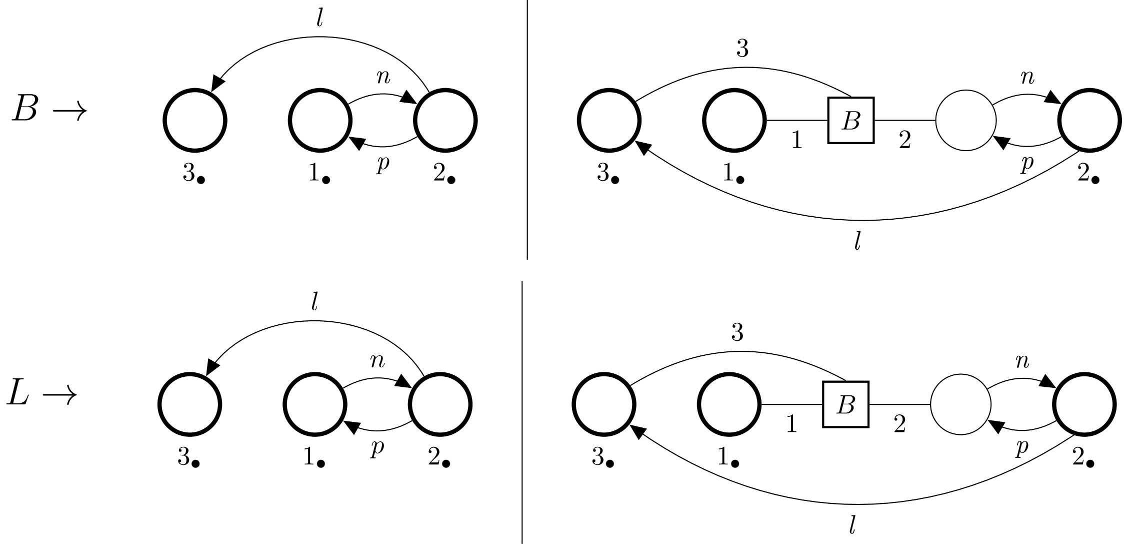


Figure 3.15: Resulting grammar $G_{(B,2)}$ and $G_{(L,2)}$.

Proof. Given a derivation we can reorder the underling replacements as long as we ensure that edges are not replaced before they are introduced, i.e. $e_j \in E_{K_i} \implies j > i$. From the confluence of hyperedge replacement grammars (Lemma 2.2) it follows that the reordering of derivations does not change the result. For any replacement $[K/e]$ of a sequence we define the set of successors of $[K/e]$ recursively by

$$successors([K/e]) = \bigcup_{e' \in E_K^N} successors([K'/e']) \cup E_K^N.$$

We split G' into the set of rules P_{old}, adopted from G without changes, and the set P_{new}, rules generated during elimination of local recursion at (X, i). All rules from P_{new} are of the form $X \to H_B$ or $B \to R \,|\, E$, where H_B contains an edge labelled with B.

Direction $(H \Rightarrow_G^* K) \Longleftarrow (H \Rightarrow_{G'}^* K)$**:**

We reorder a given derivation $H \Rightarrow^* K : H[K_1/e_1] \dots [K_n/e_n] = K$ in $G_{(X,i)}$ such that each $[K_i/e_i]$, with $\text{lab}(e_i) \to K_i \in P_{new}$ is directly succeeded by the elements of $\{[K/e] \in successors([K_i/e_i]) \mid \text{lab}(e) \to K \in P_{new}\}$, i.e. successors contained in P_{new}:

$$H \underbrace{[e_1/K_1] \dots [K_{i-1}/e_{i-1}]}_{\in P_{old}} \underbrace{[K_i/e_i] \dots [K_j/e_j]}_{\in P_{new}} [K_{j+1}/e_{j+1}] \dots = K$$

There can be multiple or no subderivations $[K_i/e_i] \dots [K_j/e_j]$ from P_{new}. The replacements before the first of these subderivations do not involve any rules from P_{new} and thus no edges labelled with B are introduced. Therefore $H[K_1/e_1] \dots [K_{i-1}/e_{i-1}] = H' \in \text{HG}_\Sigma$ can be derived in G accordingly.

Consider the grammar G given in Figure 3.16. By construction we know that P_{new} is of the form given in Figure 3.17 and the subderivation, starting from H' has the form:

$$H'[K/e_i][R_1/e_{i+1}]\dots[R_{j-2}/e_{j-1}][E/e_j] = K' \in \mathrm{HG}_\Sigma.$$

We show by induction over the number $k \in \mathbb{N}$ of $B \to R$ derivations, that for any such sequence a corresponding derivation $H \Rightarrow^* K'$ in G exists.

Induction Base $k = 0$:

The base case is a derivation in G' without recursive rules $H_1 \xRightarrow{X \to H_B} H_2 \xRightarrow{B \to E} K'$, i.e. a derivation that consists only of an initial rule (introducing a B-edge) and a terminal rule (removing the B-edge). The corresponding derivation in G is given by $H_1 \xRightarrow{X \to X_n} H_2' \xRightarrow{X \to X_t} K'$, where $X \to X_n$ generates the vertices and edges from the exit rule $B \to E$ while the remaining graph is generated by $X \to X_t$.

Induction Hypothesis:

A corresponding derivation can be found for an arbitrary but fixed number $k \in \mathbb{N}$ of derivations with recursive rules $B \to R_i$.

Induction Step $k \to k + 1$:

Consider the derivation

$$H \xRightarrow{X \to H_B} H_1 \xRightarrow{B \to R_1} \dots \xRightarrow{B \to R_{k+1}} H_{k+1} \xRightarrow{B \to E} K'$$

By the induction hypothesis we know that

$$H \xRightarrow{X \to H_B} H_1 \xRightarrow{B \to R_1} \dots \xRightarrow{B \to R_k} H_k \xRightarrow{B \to E} K'$$

can be simulated in G via

$$H \xRightarrow{X \to X_n} H_1' \xRightarrow{X \to X_{n_1}} \dots \xRightarrow{X \to X_{n_k}} H_k' \xRightarrow{X \to X_t} K'.$$

We extend this derivation by inserting the rule $X \to X_{n_{k+1}}$ corresponding to $B \to R_{k+1}$ at the beginning of the derivation. See Figure 3.18 for the correspondence of the G'- and the G-derivation. Note that graph parts derived during the i-th derivation step of G', are

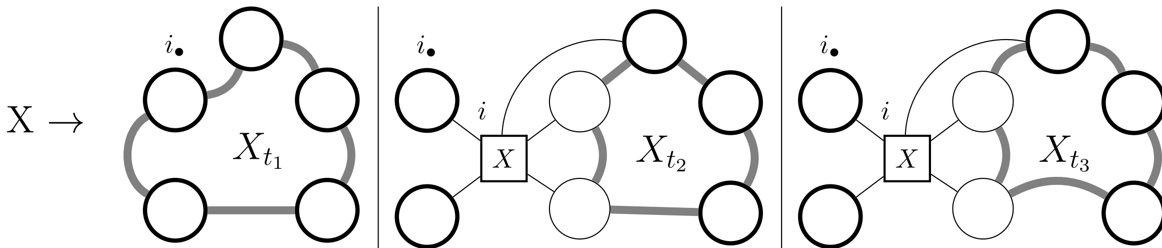
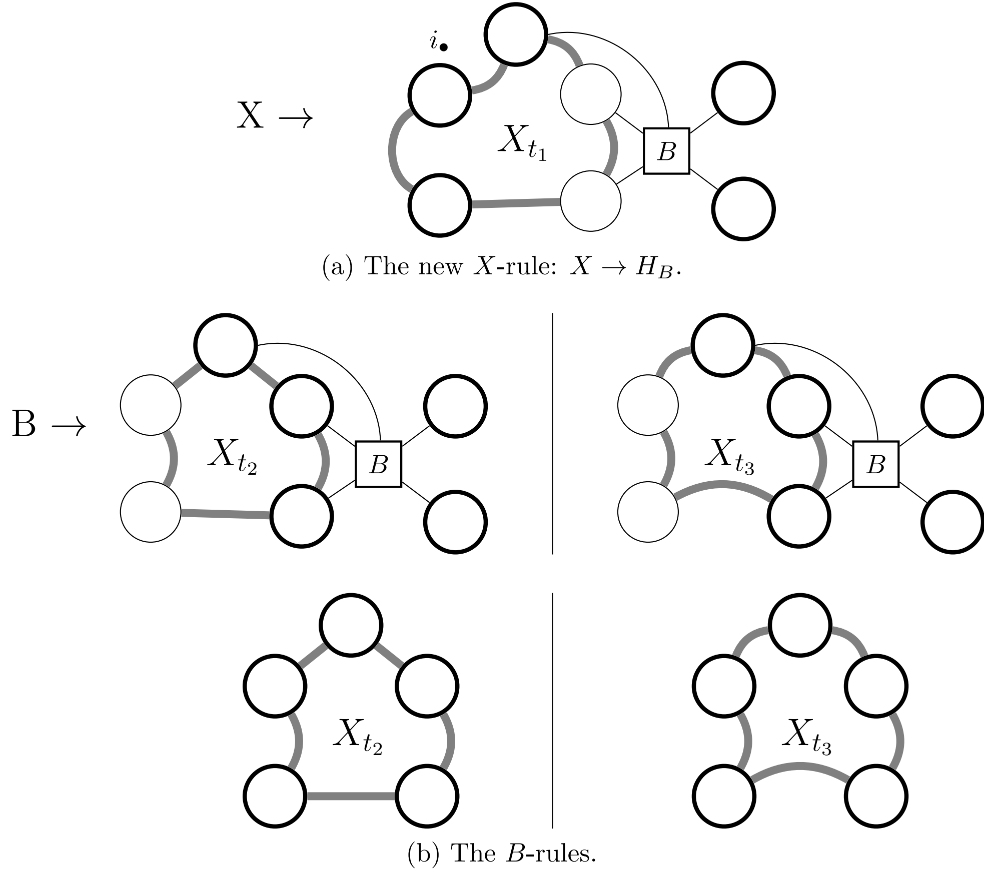


Figure 3.16



(a) The new X-rule: $X \to H_B$.

(b) The B-rules.

Figure 3.17

derived in G by derivation step $(k+1)-(i-1) = k+2-i$, where $k+1$ is the overall number of derivation steps.

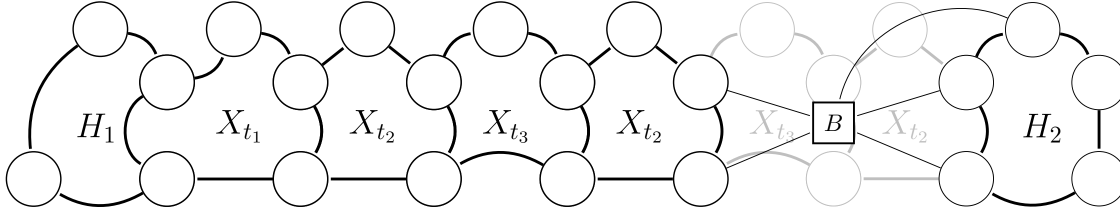
$$H \xRightarrow{X \to X_n} H'_1 \xRightarrow{X \to X_{n_{n+1}}} H'_2 \xRightarrow{X \to X_{n_1}} \dots \xRightarrow{X \to X_{n_k}} H'_{k+1} \xRightarrow{X \to X_t} K'.$$

Direction $(H \Rightarrow^*_G K) \implies (H \Rightarrow^*_{G'} K)$**:**

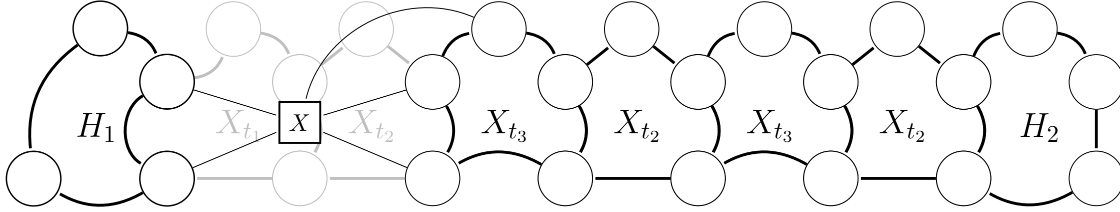
The opposite direction can be proven analogous by reordering the derivation and considering subderivations of local (X, i)-recursive rules. Each subderivation can then be simulated by a corresponding derivation in G'. Therefore every $K \in \mathrm{HG}_\Sigma$ derivable in G is derivable in G'. □

Step 3: Generation of Greibach rules.

Starting at the tentacle with the highest order, LGNF can be established for $G_{(X,i)}$ by eliminating every non-reduction (Y, j)-tentacle connected to external vertex i. This



(a) A derivation in G.



(b) The mirrored derivation in $G_{(X,i)}$.

Figure 3.18: Mirroring derivations.

elimination can be realised in a single replacement step, as $(X, i) < (Y, j)$ (otherwise we would have eliminated the rule in the first step) and as we perform step three from the highest to the lowest tentacle, (Y, j) is already in LGNF.

Step 4: Transforming new nonterminals to GNF.

In the final step we apply steps one to three to the nonterminals newly added in step two. This potentially results in further nonterminals that are added during step two of the construction. Also for these newly introduced nonterminals LGNF has to established. To ensure termination of the construction only finitely many new nonterminals should be generated. We achieve this by merging nonterminals $X, Y \in N_\Sigma$ if they share the same rule graphs, i.e. $\{R \in \mathrm{HG}_\Sigma \mid X \to R \in G\} = \{R \in \mathrm{HG}_\Sigma \mid Y \to R \in G\}$. Merging of nonterminals X, Y is realised by replacing all occurrences of X by Y. Note that the order on external vertices can be chosen arbitrarily. Unnecessary steps introduced by unsuitable orders can be avoided by considering permutations of external vertices to be isomorphic. If we reach an isomorphic nonterminal after arbitrarily many steps all nonterminals inbetween represent the same language and thus can be merged as long as the rank of the nonterminals permit this.

Example 3.8:

Consider the subgrammars $G_{(B,2)}$ and $G_{(L,2)}$, given in Figure 3.15. The left-hand side of the grammars are different nonterminals but the rule graphs are the same for both grammars. Therefore nonterminal B and L describe the same language and we can merge them. Merging yields the grammar G_L in LGNF as depicted in Figure 3.19.

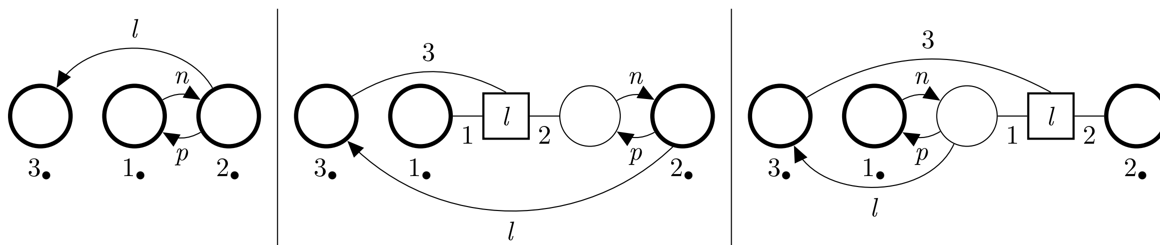


Figure 3.19: Grammar rules for doubly-linked lists in LGNF.

Theorem 3.4 (Termination):
After finitely many steps a nonterminal can be merged, thus the construction of LGNF terminates.

Proof. By construction we know that for every locally recursive rule exactly two production rules for the new nonterminal B are generated. The first is obtained from the respective original rule, where the locally recursive edge and corresponding external vertices are removed. The second is a copy of the first rule with an additional nonterminal edge labelled with B on the right-hand side. This means that the subgraph used as basis for the construction of production rules for new nonterminals remains the same. Exactly one additional edge (labelled with B) is introduced in the construction. As there exist only finitely many possibilities to combine the fix subgraph and edge, eventually a nonterminal will occur whose associated rules coincide with those of an already existing nonterminal (up to renaming). Thus we can merge these nonterminals as any further constructed production rules would be redundant. □

Note that the LGNF ensures the increasingness property, as every production rules belongs to at least one $G_{(X,i)}$ composed by rules with terminal edges at vertex $ext(i)$.

Lemma 3.4 (Increasingness of LGNF):
Each heap abstraction grammar in LGNF is increasing.

While we have restricted the normalisable grammars here to heap abstraction grammars, the procedure can easily be lifted to arbitrary bounded hyperedge replacement grammars.

3.4.3 Concretisation Function

Now that we can transfer any heap abstraction grammar into one in LGNF, we can use the special properties of LGNF grammars to realise directed concretisations. First we introduce the directed concretisation of tentacles.

Definition 3.10 (Tentacle concretisation):
Given a grammar $G \in HAG_\Sigma$ in LGNF, a heap graph $H \in HG_\Sigma$, and a non-reduction tentacle $(e, i) \in Tent_H$, the concretisation set of (e, i) in H is defined as

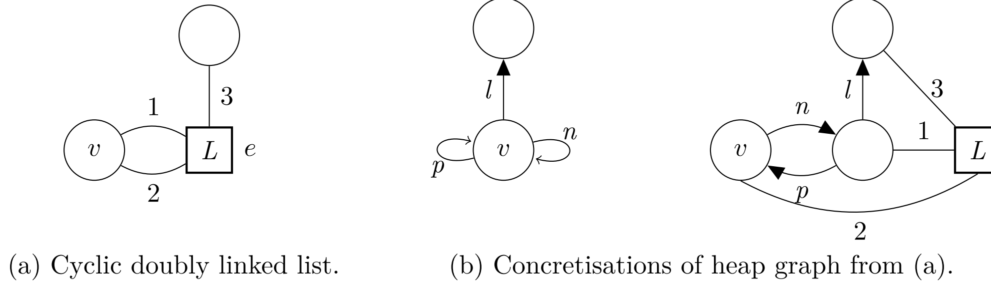
$$tentConc_G(H, (e, i)) = \{H[R/e] \mid \mathrm{lab}_H(e) = X, X \to R \in G_{(X,i)}\}.$$

Remember that given a grammar $G \in \mathrm{HAG}_\Sigma$ in LGNF for any nonterminal $X \in N_\Sigma$ it holds that $L_{(G_{(X,i)} \cup G^{\overline{X}})}(X) = L_G(X)$. Combining this property with Theorem 3.2 and the decomposition lemma (Lemma 2.4) gives us the following result.

Lemma 3.5 (Completeness of tentacle concretisation):
Given a grammar $G \in HAG_\Sigma$ in LGNF, a heap graph $H \in HG_\Sigma$, and one of its tentacles $(e, i) \in Tent_H$ it holds that

$$L_G(H) = L_G(tentConc_G(H, (e, i))).$$

A tentacle concretisation replaces a non-reduction nonterminal tentacle attached to some vertex by one or more terminal and reduction tentacles. Tentacle concretisation thus reduces the number of non-reduction nonterminal tentacles attached to the vertex. By successive tentacle concretisations we can replace all non-reduction nonterminal tentacles attached to a vertex.



(a) Cyclic doubly linked list. (b) Concretisations of heap graph from (a).

Figure 3.20: Concretisation of tentacle $(e, 1)$

Definition 3.11 (Vertex concretisation):
Given a hyperedge replacement grammar $G \in HAG_\Sigma$ in LGNF, a heap graph $H \in HG_\Sigma$, and a vertex $v \in V$, the concretisation set *of v is defined as*

$$conc(H, v) = \begin{cases} \{H\} & \text{, if } v \text{ is concrete in } H \\ \bigcup\ conc(tentConc(H, nrTent_H(v)), v) & \text{, otherwise} \end{cases}$$

where for $v \in V_H$ $nrTent_H(v)$ is the set of all non-reduction tentacles in $Tent_H(v)$.

A vertex concretisation is a repeated application of tentacle concretisations, until the corresponding vertex is concrete. That is for any non-reduction nonterminal tentacle attached to the vertex we realise a concretisation. The results in one set of concretisations for each tentacle, where in any concretisation the corresponding tentacle is replaced by terminal tentacles. However, other nonterminal tentacles could remain and need to be replaced in further steps, therefore we repeat the operation until the vertex is concrete. In each step at least one outgoing terminal tentacle is concretised. Therefore finally all outgoing terminal tentacles are concrete and the concretisation terminates. As vertex concretisations is based on tentacle concretisations, we can extend Lemma 3.5 towards the following theorem.

Theorem 3.5 (Completeness of vertex concretisation):
Given a grammar $G \in HAG_\Sigma$ in LGNF, a heap graph $H \in HG_\Sigma$, and one of its vertices $v \in V_H$ it holds that

$$L_G(H) = L_G(\ conc(H, v)).$$

Example 3.9: Vertex concretisation

Consider the heap graph in Figure 3.20(a) based on the grammar for doubly linked lists from Figure 3.19. Concretisation of tentacle $(e, 1)$ yields the two heap graphs given in Figure 3.20(b). Note that the vertex v remains abstract in the second heap graph, as a $(L, 2)$-tentacle remains at v.

The concretisation of vertex v yields the three heap graphs given in Figure 3.21. The two tentacle concretisations from Figure 3.20(b) are intermediate results of the second and third heap graph. Note that the vertex v is concrete in each of the graphs.

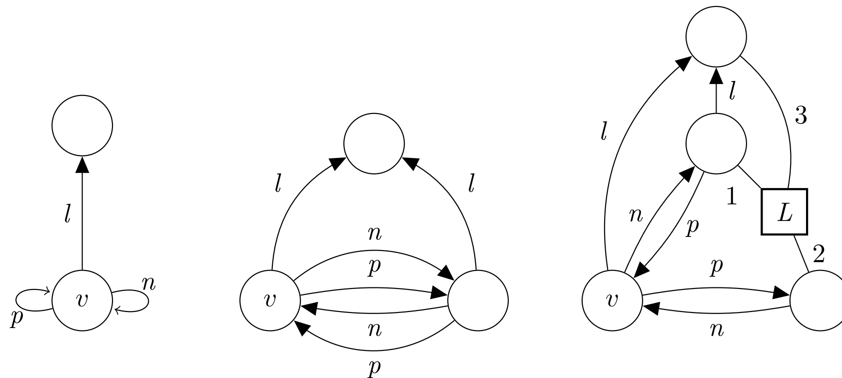


Figure 3.21: Concretisation of vertex v

3.5 Transition Systems

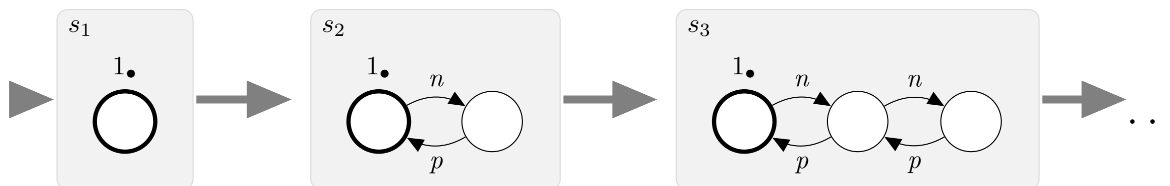
We discussed how to model heaps of object-oriented programs as heap graphs. If we want to describe the behaviour of a program it is not sufficient to consider individual (heap)

states but transition systems [BK08], i.e. the set of states reachable by a program (or system) and the transitions between those. Note that the state of a heap manipulating program is not only determined by the heap but also by the control state reflecting the remaining program statements and the values of local variables. In Chapter 5 we will encode the control state into the graph representation.

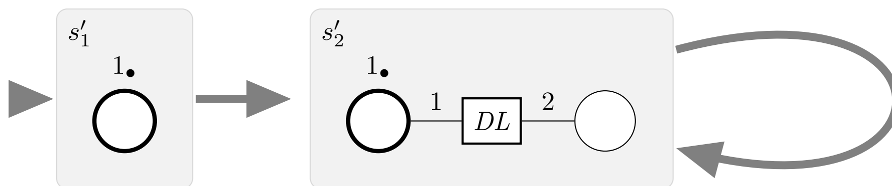
Definition 3.12 (Transition system):
Given a ranked alphabet Σ, a state space over Σ is a tuple $\mathcal{T} = (S, S_0, \triangleright)$, consisting of a (possibly infinite) set of states $S \subseteq HG_\Sigma$, a set of initial states $S_0 \subseteq S$, and a total *transition relation $\triangleright \subseteq S \times S$.*

The set of all transition systems over the alphabet Σ is denoted by TS_Σ.
We call $\mathcal{T} \in TS_\Sigma$ concrete if all states of $\mathcal{T}$ are concrete, i.e. if $S_\mathcal{T} \subseteq HG_{T_\Sigma}$.

Example 3.10:
In Figure 3.22 two state spaces are given. The relation $\triangleright$ is depicted by arrows (→). The state space in (a) has infinite many states. The single initial state s_1, marked by an incoming arrow, consists of a single vertex. Its unique successor s_2 is a doubly linked list with two elements, succeeded by a list of three elements s_3, and so on. Each state represents a list with one element more than its predecessor. The state space from Figure 3.22(a) is concrete. In (b) an abstract state space is given. The single initial hypergraph s'_1 is the same as the one from (a). The succeeding one is an abstract hypergraph. Considering the grammar for doubly linked lists from Section 3.3 it represents an arbitrary doubly linked lists. The state s'_2 succeeds itself, i.e. $s'_2 \triangleright s'_2$ depicted by a self-loop.



(a) Concrete state space $\mathcal{T}_1$



(b) Abstract state space $\mathcal{T}_2$

Figure 3.22: State spaces.

Starting at an initial state we can step through a state space via the transition relation. Such a walk through the state space is called a path.

Definition 3.13 (Paths [BK08]):
Given a transition system $\mathcal{T} \in TS_\Sigma$ a path *in $\mathcal{T}$ is a infinite sequence of states $\pi = s_1\, s_2\, s_3\, \ldots$ with $s_1, s_2, s_3, \ldots \in S_\mathcal{T}$ such that for two succeeding positions $i, i+1$, with $i \in \mathbb{N}$ it holds that $(\pi[i], \pi[i+1]) \in \rhd_\mathcal{T}$. We denote the set of all paths starting in $s \in S$ by $Path_s$*

An initial path *is a path π for which the first element is a initial state $\pi[1] \in S_0$. We denote the set of all initial paths over $\mathcal{T}$ by $Path_\mathcal{T}$.*

In Example 3.10 both state spaces have a single (initial) path. The concrete one $\mathcal{T}_1$ has the path $\pi_1 = s_1\, s_2\, s_3\, s_4 \ldots$, the abstract one $\mathcal{T}_2$ the path $\pi_2 = s'_1\, s'_2\, s'_2 \ldots$. Considering the grammar G for doubly linked lists from Figure 3.6 on page 46, if we pairwise compare the states of the paths we observe that for any $i \in \mathbb{N}$ the hypergraph $\pi_2[i]$ represents the hypergraph $\pi_1[i]$, i.e. $\pi_1[i] \in L_G(\pi_2[i])$. Thus the abstract state space $\mathcal{T}_2$ represents the concrete one $\mathcal{T}_1$. We call the former an over-approximation of the latter if this applies to all paths.

Definition 3.14 (Over-approximation):
Given $G \in HRG_\Sigma$ and $\mathcal{T}_1, \mathcal{T}_2 \in TS_\Sigma$, we call $\mathcal{T}_2$ an over-approximation *of $\mathcal{T}_1$, denoted $\mathcal{T}_2 \succeq \mathcal{T}_1$, if for each path $\pi_1 \in Path_{\mathcal{T}_1}$ there exists a path $\pi_2 \in Path_{\mathcal{T}_2}$ such that $\pi_1[i] \in L_G(\pi_2[i])$ for each $i \in \mathbb{N}$.*

The over-approximation of a transition system can be significantly smaller up to cases where an infinite state space can be represented finitely, as in Example 3.10.

We call a total relation $\rhd \subseteq \mathrm{HG}_\Sigma \times \mathrm{HG}_\Sigma$ a *transition relation*. A transition relation $\rhd$ is *concrete* if for any concrete heap graph $H \in \mathrm{HG}_{T_\Sigma}$ it holds that $\{H' \in \mathrm{HG}_\Sigma \mid H \rhd H'\} \subseteq \mathrm{HG}_{T_\Sigma}$, i.e. successors of concrete states are again concrete states.

A transition relation $\rhd$ induces, together with a set S_0 of initial graphs, the transition system $\mathcal{T}_{S_0,\rhd} = (S, S_0, \rhd \cap (S \times S))$ of reachable states, where S is the set of all graphs reachable via the transition relation $\rhd$ starting from some state of S_0, i.e. S is the smallest set such that $H \in S$ if $H \in S_0$ or exists $H' \in S$ such that $H' \rhd H$.

It results directly from the definition of the induced transition system that given a concrete transition relation $\rhd$ and a set of concrete initial states $S_0 \subseteq \mathrm{HG}_\Sigma$ the induced transition system $\mathcal{T}_{S_0,\rhd}$ is a concrete transition system.

We call a function $f : \mathrm{HG}_\Sigma \to \mathrm{HG}_\Sigma$ a *transition function*. In analogy to transition relations we consider *concrete transition functions*.

Example 3.11:
Consider a transition relation $\triangleright$ *for which* $H_1 \triangleright H_2$ *holds if* H_1 *has exactly one external node* $|\mathrm{ext}_{H_1}| = 1$ *and*

$$H_2 = (V_{H_1} \cup \{v\}, E_{H_1} \cup \{e\}, \mathrm{att}_{H_1} \cup [e \mapsto v\,\mathrm{ext}_{H_1}[1]], \mathrm{lab}_{H_1} \cup [e \mapsto n], v),$$

i.e. in $\triangleright$ *the successor of a hypergraph is a graph that is constructed by adding an additional vertex connected to the single external vertex via a n-edge and the new vertex becomes the new external one. Note that* $\triangleright$ *is a concrete transition function. This is because the construction of the successor does not add nonterminal edges, thus if a hypergraph is concrete, so is its successor. Given* $S_0 = \{(\{v\}, \emptyset, \emptyset, \emptyset, v)\}$ *the transition system induced by* S_0 *and* $\triangleright$ *is* $\mathcal{T}_1$ *from Figure 3.22(a).*

Given a heap abstraction grammar $G \in \mathrm{HAG}_\Sigma$ and two transition relations $\triangleright$ and $\blacktriangleright$, we call $\blacktriangleright$ a safe approximation of $\triangleright$, denoted $\blacktriangleright \succeq \triangleright$, if for any $H_A, H_C, H'_C \in \mathrm{HG}_\Sigma$, with $H_A \Rightarrow^*_G H_C$ and $H_C \triangleright H'_C$ there exists a graph $H'_A \in \mathrm{HG}_\Sigma$ such that $H'_A \Rightarrow^*_G H'_C$ and $H_A \blacktriangleright H'_A$. See Figure 3.23 for a schematic representation.

$$\begin{array}{ccc} H_C & \triangleright & H'_C \\ \Uparrow_* & & \Uparrow_* \\ H_A & \blacktriangleright & H'_A \end{array}$$

Figure 3.23: Safe approximation.

Lemma 3.6 (Safe approximations and over-approximations):
Given $G \in HRG_\Sigma$, $S_0 \subseteq HG_\Sigma$ *and* $\triangleright, \blacktriangleright \subseteq HG_\Sigma \times HG_\Sigma$, *it holds that*

$$\blacktriangleright \succeq \triangleright \quad \Rightarrow \quad \mathcal{T}_{S_0,\blacktriangleright} \succeq \mathcal{T}_{S_0,\triangleright}.$$

Proof. To prove that $\mathcal{T}_{S_0,\blacktriangleright}$ is an over-approximation of $\mathcal{T}_{S_0,\triangleright}$, we have to show that for any initial path $\pi_\triangleright \in Path_{\mathcal{T}_{S_0,\triangleright}}$ there exists a corresponding initial path $\pi_\blacktriangleright \in Path_{\mathcal{T}_{S_0,\blacktriangleright}}$ such that for all $i \in \mathbb{N}$ it holds that $\pi_\blacktriangleright[i] \Rightarrow^*_G \pi_\triangleright[i]$. We prove this by induction over the position $n \in \mathbb{N}$ of corresponding initial path fragments.

Induction Base $n = 1$**:**

Any initial path starts with a state from S_0. As S_0 is the same for both transition systems for any initial path $\pi_\triangleright \in Path_{\mathcal{T}_{S_0,\triangleright}}$ exists an initial path $\pi_\blacktriangleright$ in $\mathcal{T}_{S_0,\blacktriangleright}$ with $\pi_\triangleright[1] = \pi_\blacktriangleright[1]$.

Induction Hypothesis:

Given a arbitrary but fixed $n \in \mathbb{N}$, for any initial path $\pi_\triangleright \in Path_{\mathcal{T}_{S_0,\triangleright}}$ exists a initial path $\pi_\blacktriangleright \in Path_{\mathcal{T}_{S_0,\blacktriangleright}}$ with $\pi_\blacktriangleright[i] \Rightarrow^*_G \pi_\triangleright[i]$ for each $1 \leq i \leq n$.

Induction Step $n \to n+1$:

Let $\pi_{\triangleright} = s_1\, s_2\, \dots\, s_n\, s_{n+1} \dots \in Path_{\mathcal{T}_{S_0,\triangleright}}$. The induction hypothesis gives us a path $\pi_{\blacktriangleright} = s'_1\, s'_2\, \dots\, s'_n \dots \in Path_{\mathcal{T}_{S_0,\blacktriangleright}}$ with $\pi_{\blacktriangleright}[i] \Rightarrow^*_G \pi_{\triangleright}[i]$ for each $1 \le i \le n$. As s_{n+1} is successor of s_n in $\mathcal{T}_{S_0,\triangleright}$ it holds by definition of induced state spaces that $s_n \triangleright s_{n+1}$. We select a new state $s'_{n+1} \in S_{\mathcal{T}_{S_0,\blacktriangleright}}$ such that $s'_n \blacktriangleright s'_{n+1}$ and $s'_{n+1} \Rightarrow^*_G s_{n+1}$. Such a state exists as $\blacktriangleright$ is an abstraction of $\triangleright$ and therefore for each $s_n \blacktriangleright s_{n+1}$ with $s'_n \Rightarrow^*_G s_n$ (holds by induction hypothesis) there exists a s'_{n+1} for which $s'_{n+1} \Rightarrow^*_G s_{n+1}$. We construct the new path $\pi'_{\blacktriangleright} = s'_1\, s'_2\, \dots\, s_n\, \pi_{s'_{n+1}}$ where $\pi_{s'_{n+1}} \in Path_{s'_{n+1}}$ is a path in $\mathcal{T}_{S_0,\blacktriangleright}$ starting in s'_{n+1}. Note that $\pi'_{\blacktriangleright} \in Path_{\mathcal{T}_{S_0,\blacktriangleright}}$ and it holds that $\pi'_{\blacktriangleright}[i] \Rightarrow^*_G \pi_{\triangleright}[i]$ for any $1 \le i \le n+1$. ☐

Given a grammar $G \in \mathrm{HRG}_\Sigma$ and a *concrete* transition relation $\triangleright$, we call $\triangleright$ a *safe transition relation* if , $\triangleright \succeq \triangleright$ i.e. for any $H \in \mathrm{HG}_\Sigma$ it holds that

$$\{H'' \in \mathrm{HG}_\Sigma \mid \exists H' \in L(H).\ H' \triangleright H''\} \quad = \quad \{H'' \in L_G(H') \mid H \triangleright H'\}$$

Analogous for a safe transition relation that is a function f it holds for any $H \in \mathrm{HG}_\Sigma$ that $f(L(H)) = L(f(H))$.

Given a safe transition relation we get an over-approximated state space by simply starting at abstracted initial states.

Lemma 3.7 (Safe transition relations and abstracted initial states):
Given a safe transition relation $\triangleright$ and two initial sets S_0^1 and $S_0^2 \subseteq HG_\Sigma$ where S_0^2 is an abstraction of S_0^1, i.e. $\forall s^1 \in S_0^1\ \exists s^2 \in S_0^2.\ s^2 \Rightarrow^*_G s^1$, it holds that $\mathcal{T}_{S_0^2,\triangleright} \succeq \mathcal{T}_{S_0^1,\triangleright}$

Proof. Lemma 3.7 can be proven by induction in analogy to Lemma 3.6. The only change occurs in the induction base, where for the case $n = 1$ the initial state of the paths can not be chosen equal, in general. However, by definition it holds that for any $s^1 \in S_0^1$ exists a $s^2 \in S_0^2$ such that $s^2 \Rightarrow^* s^1$. Thus a corresponding path exists. ☐

A safe transition relation can be safely approximated by abstracting transitions, i.e. by abstraction of the target of a transition.

Lemma 3.8 (Abstraction of transition relations):
Given $G \in HG_\Sigma$ and a safe transition relation $\triangleright \in HG_\Sigma \times HG_\Sigma$, let $H_1, H_2 \in HG_\Sigma$ with $H_1 \triangleright H_2$ and $H'_2 \in HG_\Sigma$ with $H'_2 \Rightarrow^*_G H_2$, an abstraction of H_2. The transition relation

$$\blacktriangleright \;=\; (\,\triangleright \setminus \{(H_1, H_2)\}\,) \cup \{(H_1, H'_2)\}$$

is a safe approximation of $\triangleright$.

The correctness of the lemma follows directly from the definition of safe approximation. We can also concretise a transition of a safe transition relation $\blacktriangleright$. This results in a transition system $\triangleright$ with $\blacktriangleright \succeq \triangleright$, which is still a safe approximation of any concrete transition function that is safely approximated by $\blacktriangleright$.

Lemma 3.9 (Concretisation of transition relations):
Let $G \in HAG_\Sigma$ be a grammar in LGNF, $\blacktriangleright \in HG_\Sigma \times HG_\Sigma$ a safe transition relation, $H_1, H_2 \in HG_\Sigma$, with $H_1 \blacktriangleright H_2$. Let $e \in E_{H_2}$ with $\mathrm{lab}_{H_2}(e) = X$, and $K \subseteq HG_\Sigma$ a set of graphs with $L_G(K) = L_G(X^\bullet)$. For the transition relation

$$\blacktriangleright' = \; (\blacktriangleright \setminus \{(H_1, H_2)\}) \cup \{(H_1, H_2[H/e]) \mid H \in K\}$$

it holds that for any concrete transition function $\triangleright \in HG_\Sigma \times HG_\Sigma$ that

$$\blacktriangleright \succeq \triangleright \quad \Rightarrow \quad \blacktriangleright' \succeq \triangleright$$

Proof. We have to show that for a concrete path $\pi_\triangleright$ with abstract path $\pi_\blacktriangleright$, such that $\pi_\triangleright[i] \Rightarrow_G^* \pi_\blacktriangleright[i]$ for all $i \in \mathbb{N}$, there exists a path $\pi_{\blacktriangleright'}$ with $\pi_\triangleright[i] \Rightarrow_G^* \pi_{\blacktriangleright'}[i]$ for all $i \in \mathbb{N}$.

Given a path $\pi_\blacktriangleright$ such that for none of the positions $i \in \mathbb{N}$ it holds $\pi_\blacktriangleright[i] = H_1$ and $\pi_\blacktriangleright[i+1] = H_2$, we have $\pi_{\blacktriangleright'} = \pi_\blacktriangleright$, as any transition used in $\pi_\blacktriangleright$ also exists in $\blacktriangleright'$.

If $\pi_\blacktriangleright[i] = H_1$ and $\pi_\blacktriangleright[i+1] = H_2$ for some $i \in \mathbb{N}$, we copy $\pi_\blacktriangleright$ up to position i, i.e. $\pi_{\blacktriangleright'}[i] = \pi_\blacktriangleright[i]$. Let $\pi_\triangleright[i+1] = H_2^\triangleright$. We know that $H_2 \Rightarrow_G^* H_2^\triangleright$ and by definition of $\blacktriangleright'$ that there is some $H_2^{\blacktriangleright'}$ with $H_1 \blacktriangleright' H_2^{\blacktriangleright'}$ and $H_2^{\blacktriangleright'} \Rightarrow_G^* H_2^\triangleright$. Hereby $H_2^{\blacktriangleright'} = H_2^\blacktriangleright[H/e]$, for some $H \in K$ and there is such an element from K, as given by Lemma 2.4 and $L_G(K) = L_G(X^\bullet)$ any concrete graph derivable from $H_2^\blacktriangleright$ is still derivable from some $H_2^\blacktriangleright[H/e]$. See Figure 3.24 for a schematic representation of position i, $i+1$ and $i+2$ of the three paths.

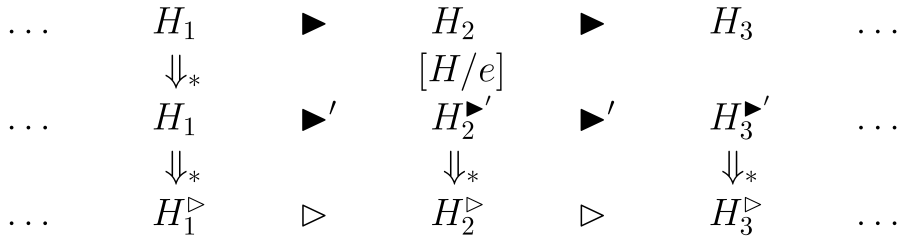


Figure 3.24: A path in $\triangleright$ with corresponding paths in $\blacktriangleright$ and $\blacktriangleright'$

Analogous to the proof of Lemma 3.6 we can now prove by induction that for any $j > i$ it holds that there exists $\pi_{\blacktriangleright'}[j] \in \mathrm{HG}_\Sigma$ with $\pi_{\blacktriangleright'}[j-1] \blacktriangleright' \pi_{\blacktriangleright'}[j]$ and $\pi_{\blacktriangleright'}[j] \Rightarrow_G^* \pi_\triangleright[j]$,

whenever $\pi_{\blacktriangleright'}[j-1] \Rightarrow_G^* \pi_{\triangleright}[j-1]$ and $\pi_{\triangleright}[j-1] \triangleright \pi_{\triangleright}[j]$. Compare also Figure 3.23 with the right lower corner of Figure 3.24.

Potentially at some position $j > i$ again $\pi_{\blacktriangleright}[j] = \pi_{\blacktriangleright'}[j]$ holds. If this is the case for position j holds the same as we have proven for position i. □

There exist transition relations that are not safe, but fulfil the necessary property on a subset of hypergraphs. Given a heap abstraction grammar $G \in \mathrm{HAG}_\Sigma$ we call a transition relation $\triangleright$ *safe under* $K \subseteq HG_\Sigma$ if $\triangleright\restriction K$ is a safe transition relation, i.e. for any $H \in K$ it holds that

$$\{H'' \in \mathrm{HG}_\Sigma \mid \exists H' \in L(H).\ H' \triangleright H''\} \quad = \quad \{H'' \in L_G(H') \mid H \triangleright H'\}.$$

A function $c : \mathrm{HG}_\Sigma \to \mathcal{P}(K)$ that maps any $H \in \mathrm{HG}_\Sigma$ into a finite set $c(H) \subseteq K$ with $L(c(H)) = L(H)$, is called a *K-concretiser*.

Given a transition function $\triangleright$ that is safe under K and an K-concretiser c, we can combine both to a safe transition relation $\blacktriangleright$, that first transforms any graph into one from K and then realises transitions via $\triangleright$.

Lemma 3.10 (Safe approximations of transition relations):
Given a heap abstraction grammar $G \in HAG_\Sigma$, a set $K \subset HG_\Sigma$, and a transition relation $\triangleright$ safe under K, as well as a K-concretiser c, the relation

$$\blacktriangleright \quad = \quad \{(H_1, H_2) \in HG_\Sigma \times HG_\Sigma \mid \exists H \in c(H_1).\ (H, H_2) \in \triangleright\}$$

is a safe approximation of $\triangleright$.

Lemma 3.10 follows directly by applying Lemma 3.9 to a transition relation that is safe under K.

Theorem 3.6 (Combined abstraction of transition functions):
Given a heap abstraction grammar $G \in HAG_\Sigma$, a set $K \subset HG_\Sigma$, an under K safe transition relation $\triangleright$ as well as a K-concretiser c, and an abstraction function $a : HG_\Sigma \to HG_\Sigma$ with $a(H) \Rightarrow_G^* H$ for any $H \in HG_\Sigma$, the relation

$$\blacktriangleright \quad = \quad \{(H_1, a(H_2)) \in HG_\Sigma \times HG_\Sigma \mid \exists H \in c(H_1).\ (H, H_2) \in \triangleright\}$$

is a safe approximation of $\triangleright$

Theorem 3.6 can be proven by combining Lemma 3.10 and Lemma 3.8.

In Chapter 5 we will define abstract Java bytecode statements as a combination of a safe transition relation (defining the concrete semantics) with a suitable concretisation and an abstraction relation. We then use the abstract states induced by the abstract transition relation in Chapter 7 for model checking.

3.6 Conclusions

In this chapter we concretised how heaps can be represented by hypergraphs. Especially we formalised under which conditions a hypergraph is a valid representation of a heap. Such graphs are called heap graphs. We introduced heap abstraction grammars (Definition 3.6) as a restriction of HRGs suitable to represent data structures. We formalised four grammar properties which are essential for the correctness of our approach.

Beside the representation we also reconsidered the concretisation function. We introduced a local concretisation that allows us to determine the pointers and successors of any vertices in an abstract representation. This concretisation is essential for our framework and is based on the LGNF introduced in Section 3.4. We proved that for any heap abstraction grammar there exists a quasi-equivalent one in LGNF (Theorem 3.3 on page 52).

Reconsider the analysis of Lindstrom's algorithm. In the previous chapter we introduced a grammar for trees for which we have seen that it is not suitable for heap reperesentation, it is not a HAG. Namely the grammar from Figure 2.3 on page 21 is neither increasing nor typed. In this chapter we presented grammars for various data structures in Section 3.3 and amongst those another tree grammar which is a HAG and which is quasi-equivalent to the one from Figure 2.3. However, the grammar is unfortunately much bigger (24 instead of four rules) which could have a negative influence on the analysis performance. Instead we consider the tree grammar in Figure 3.25. This grammar generates not arbitrary trees but only fully branched ones, i.e. trees where each node has either an l- and an r-successor or none. This restricts the possible types of vertices to two. The grammar fulfils all properties of a HAG and in addition is also backward confluent.

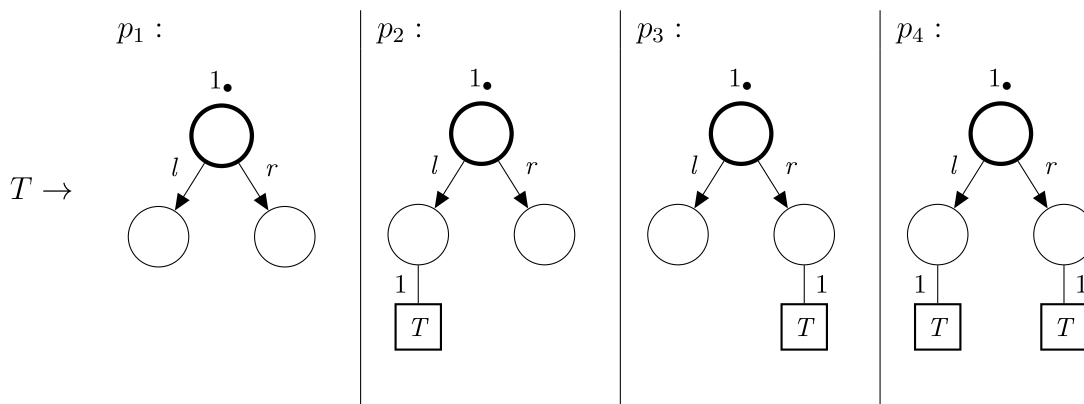


Figure 3.25: HAG for fully branched binary trees.

If we review the explanations in Section 1.1 on page 2 we will notice that there are two

different types of placeholders for trees in Figure 1.2. One with one specific vertex which is the root of the tree. This placeholder corresponds to a T-labelled edge, and another with two specific vertices, where one is the root while the other is one of its leaves. In Figure 1.3(c) this placeholder is represented by a P-labelled edge. This placeholder can be seen as a representation of a path through the tree. Corresponding P-rules are given in Figure 3.26. The idea behind the rules is as follows: Starting at the root node (the first external vertex) the specific leaf (the second external vertex) could either be a direct successor (p_5 - p_8) or the leaf of one of the two subtrees (p_9 - p_{12}). In both cases it could either be in the left (p_5, p_6 and p_9, p_{10}) or right subtree (p_7, p_8 and p_{11}, p_{12}). Note that as within the rule both successors of the root node are concrete we also have to decide whether the uninvolved subtree is a leaf (e.g. p_5 vs p_6 and p_7 vs p_8).

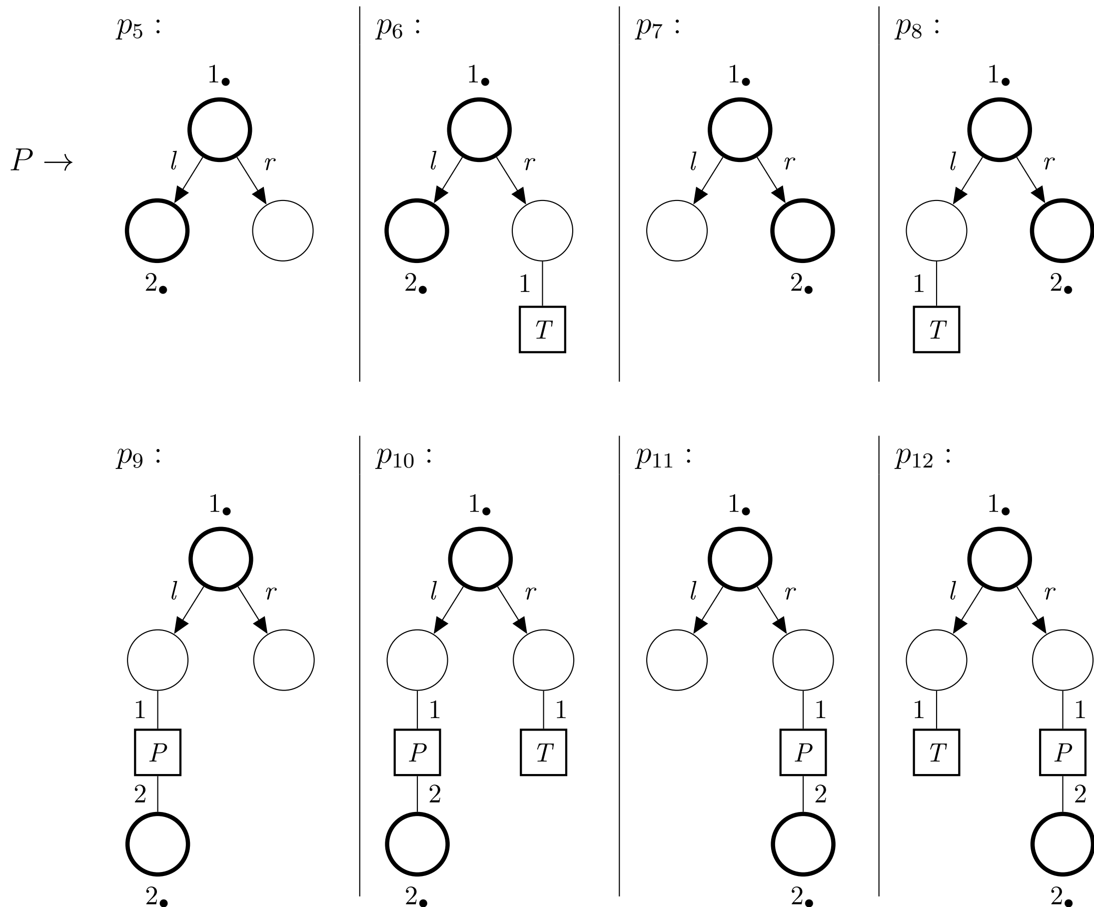


Figure 3.26: Path grammar for fully branched trees.

The grammar composed of the T and P rules is a HAG as the four properties are fulfilled. In addition the grammar is in LGNF, as for the nonterminals T and P there is a unique non-reduction tentacle $(T, 1)$, respectively $(P, 1)$ ($(P, 2)$ is a reduction tentacle) and in any rule the first external vertex is concrete. The grammar, however, is not backward confluent as indicated by the non-joinable critical pair depicted in Figure 3.27.

Fortunately we can resolve this issue by adding the T-rule from Figure 3.28, which in contrast to the case from Section 2.7 results in an equivalent and backward confluent grammar.

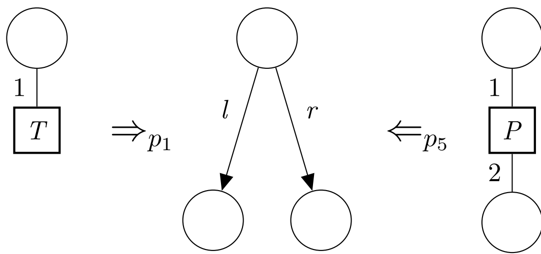


Figure 3.27: Not joinable critical pair.

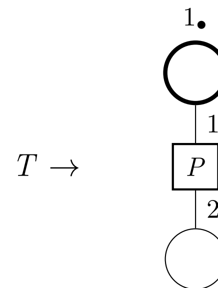


Figure 3.28: Additional rule.

Of course it is inconvenient that this grammar restricts the considered trees to those which are fully branched. The problem we encounter here is the implicit modelling of *null*-pointers by absence of outgoing edges, which leads to a variety of different vertex types. In Chapter 5 we will introduce an explicit representation of *null* which allows us to define a compact grammar for arbitrary binary trees.

In Section 3.5 we introduced transition systems where the states are hypergraphs. The ability of safely over-approximate these transition system by abstracting its transition function (Lemma 3.6) is used in Chapter 5 to define an abstract semantics. The abstract transition function aka abstract semantics for Java bytecode that we define in Chapter 5 is based on the technique of combining a K safe concrete transition relation with a concretisation and abstraction function (Theorem 3.6 on page 68).

Chapter 4
Object-Oriented Heaps

In the previous chapter we have seen how hyperedge replacement grammars can be used to describe pointer structures. Given an object-oriented language such as Java we do not just have locations and pointers, but each location is handled as an object of a well-defined type. The object type determines a set of member variables (outgoing pointers) and a set of member methods to interact with the object. In order to deal with typed objects, we extend the hypergraph definition by associating each vertex with a type. In Figure 4.1 a graph with associated types is depicted, where the types are given as vertex labels. The graph represents a tree with linked leaves, where tree nodes are inner nodes (type I) or leaves (type L). Any node has a pointer to its parent(p). Inner nodes have a left(l) and right(r) child, while the leaf nodes are connected through next(n) pointers. This data structure will serve us as running example in this chapter. Java class definitions for the different node types are given in Example 4.1.

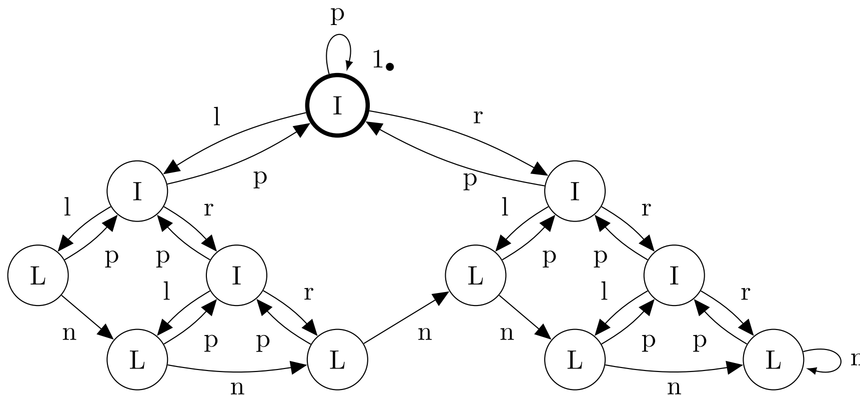


Figure 4.1: A typed hypergraph representing a tree with linked leaves.

In the following we consider Java programs and type definitions therein. We extract *typed alphabets* from class definitions in Section 4.1, which we use as a basis for the definition of heap configurations. *Heap configurations* are an extension of heap graphs as presented in Chapter 3, that reflects type information. In Section 4.2 we will define the so called

data structure grammars as restricted extension of hyperedge replacement grammars that ensure that any derivable graph is a valid heap configuration.

The concepts presented in this chapter are motivated and based on Java as programming language. However, the concepts can easily be transferred to other object-oriented languages such as C++ or Objective C.

4.1 Typed Objects

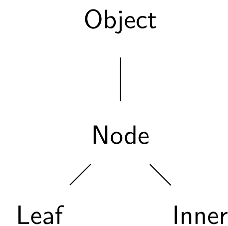
In Java, object types are defined through *class definitions*. A class definition declares an object type and determines the available *member methods* and *fields* (*member variables*) for objects of that type. The class definitions define a partial order, called subtype relation, on the set of types. We call the poset determined by the set of types and the subtype relation *type poset*. A type inherits member methods and fields from its super types, i.e. member methods and fields defined for some type are also available in any object of a subtype.

Example 4.1: Class Definition

Figure 4.2 depicts (a) a Java class definition and (b) the resulting type poset, as defined by the subtype relation. The class definition describes tree nodes as used for trees with linked leaves, where any node has a parent pointer. We distinguish between inner nodes, that have a left and a right successor and leaf nodes which have a next-pointer for the linking between leaves. Thus the three different types that are declared are, Node*,* Inner *and* Leaf*.*

```java
class Node{
  Inner parent;
}
class Inner extends Node{
  Node left, right;
}
class Leaf extends Node{
  Leaf next;
}
```

(a) Java classes.



(b) Type poset.

Figure 4.2: Class definitions and resulting poset.

Note that in Java any type is a subtype of the type Object *by default. As* Inner *and* Leaf *are subtypes of* Node*, the field Node.parent is defined for both. To get a unique name for each field, we prefix field names by the name of the class where they are defined.*

Given a set of class definitions we denote the set of types by $\mathbb{T}$, the subtype relation by $\preceq$, and the set of field names by $\mathbb{F}$. We aim at modelling heap structures as graphs, where we use types as vertex labels and field names as edge labels. While in Section 2.2 we used a ranking function to determine the number of tentacles for edges, we now use a typing function *types* to define a sequence of typed tentacles, where the type determines the type of the vertices that this tentacle is attached to. Further we distinguish between *entrance*- and *reduction*-tentacles, also named $\triangledown$- and $\blacktriangle$-tentacles, respectively. A $\triangledown$-tentacle defines an entry point into the structure represented by an edge, whereas a $\blacktriangle$-tentacle represents only exit points. Thus if a vertex is connected to a $\triangledown$-tentacle we can derive outgoing pointers at this vertex by replacing the corresponding edge, whereas at a vertex connected to $\blacktriangle$-tentacles of the edge, we can derive incoming pointers only. Figure 4.3 depicts this schematically for a X-labelled edge with both a $\triangledown$- and a $\blacktriangle$-tentacle. The two attached vertices are of type Leaf (L) with outgoing pointers for the fields *Node.parent* (p) and *Leaf.next* (n) and of type Inner (I) with incoming pointers for the fields *Inner.left* (l) and *Inner.right*(r). Note that a vertex connected to a $\blacktriangle$-tentacle can have outgoing pointers, but outside of the structure defined by the edge.

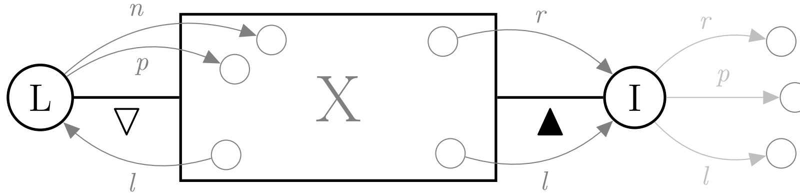


Figure 4.3: Entrance and reduction tentacles.

Given a set A we write $A_\blacktriangle$ for $A \times \{\blacktriangle\}$, $A_\triangledown$ for $A \times \{\triangledown\}$, and $A_{\triangledown\blacktriangle}$ for $A_\blacktriangle \cup A_\triangledown$. We denote $(a, \blacktriangle)$ and $(a, \triangledown)$ by $a_\blacktriangle$ and $a_\triangledown$, respectively. If we use elements of $A_{\triangledown\blacktriangle}$ where elements from A are expected, we refer to the projection on the first element.

Definition 4.1 (Heap alphabet):
A heap alphabet $\Sigma = (\mathbb{F} \cup N, (\mathbb{T}, \preceq), types)$ is a tuple with $\mathbb{F}$ a set of field labels, a set N of nonterminals, $(\mathbb{T}, \preceq)$ a poset of types, and a typing function $types : \mathbb{F} \cup N \to \mathbb{T}_{\triangledown\blacktriangle}{}^\star$, where $types(\mathbb{F}) \subseteq \mathbb{T}_\triangledown \cdot \mathbb{T}_\blacktriangle$ (here $\cdot$ represents the concatenation of elements).

As for ranked alphabets we distinguish terminal from nonterminal labels. Here the terminal labels are the field names $\mathbb{F}$. The ranking function is implicitly given by $\mathrm{rk}(f) = |types(f)|$. On type sequences $\preceq$ is the point-wise extension of $\preceq$.

In Java, each field declaration defines pointers pointing from an object of the defining class to an object of the target type. Correspondingly given the class definition **class** A{ B f;} we define $types(A.f) = A_\triangledown \cdot B_\blacktriangle$. Further we define the function *fields*, which maps a type

from $\mathbb{T}$ to the set of outgoing pointers defined for objects of this type, i.e. for $t \in \mathbb{T}$ we get

$$\mathit{fields}(t) = \{f \in \mathbb{F} \mid \mathit{types}(f)[1] \succeq t_{\triangledown}\}.$$

Note that this definition reflects that pointers defined in some type are also defined for any of its subtypes.

Example 4.2: Heap alphabet extracted from Java class definitions

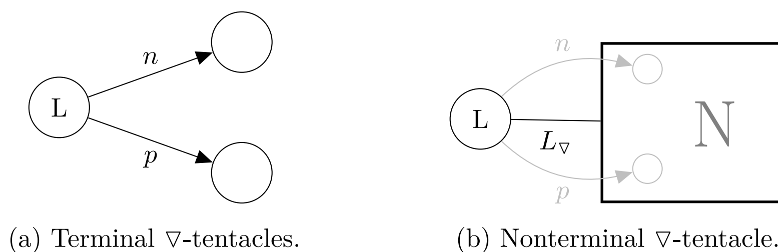
Reconsider the Java-class definitions given in Fig. 4.2(a) we get the set of types $\mathbb{T} = \{\mathsf{Object}, \mathsf{Node}, \mathsf{Inner}, \mathsf{Leaf}\}$ *and the set of fields defined for these types* $\mathbb{F} = \{\mathit{Node.parent}, \mathit{Inner.left}, \mathit{Inner.right}, \mathit{Leaf.next}\}$*. The poset* $(\mathbb{T}, \preceq)$ *defined by the subtype relation is given in Fig. 4.2(b). The type sequence for the field parent is* $\mathit{types}(\mathit{Node.parent}) = \mathsf{Node}_{\triangledown}\mathsf{Inner}_{\blacktriangle}$*, for the fields left and right* $\mathit{types}(\mathit{Inner.left}) = \mathit{types}(\mathit{Inner.right}) = \mathsf{Inner}_{\triangledown}\mathsf{Node}_{\blacktriangle}$ *and* $\mathit{types}(\mathit{Leaf.next}) = \mathsf{Leaf}_{\triangledown}\mathsf{Leaf}_{\blacktriangle}$*.*

The function fields is defined as $\mathit{fields}(\mathsf{Inner}) = \{\mathit{Inner.left}, \mathit{Inner.right}, \mathit{Node.parent}\}$*,* $\mathit{fields}(\mathsf{Node}) = \{\mathit{Node.parent}\}$*, and* $\mathit{fields}(\mathsf{Leaf}) = \{\mathit{Leaf.next}, \mathit{Node.parent}\}$*.*

Using heap alphabets we define a special class of heap graphs called *heap configurations*. From the heap graphs we carry over the sets of nodes and edges, as well as the labelling and attachment function for edges. In addition we have a labelling of vertices with types, that involves additional restrictions.

For vertices we impose requirements regarding the attached $\triangledown$-tentacles. These tentacles must represent exactly the pointers defined for the type of the node, as in Java there are no undefined pointers. Whereas terminal $\triangledown$-tentacles represent a single outgoing pointer, we assume nonterminal $\triangledown$-tentacles to represent all outgoing pointers of a vertex, i.e. for vertices we require that there is one terminal $\triangledown$-tentacle for each field defined for the vertex type or a single $\triangledown$-tentacle, that belongs to a nonterminal edge.

Nodes connected to terminal $\triangledown$-tentacles are called concrete, those connected to nonterminal $\triangledown$-tentacles, i.e. with abstracted outgoing pointers, abstract.



(a) Terminal $\triangledown$-tentacles. (b) Nonterminal $\triangledown$-tentacle.

Figure 4.4: Vertices and there $\triangledown$-tentacles.

In Figure 4.4 both cases are depicted. In (a) there is a vertex labelled Leaf with corresponding terminal edges labelled with the fields n and p for *next* and *parent*. In (b) a single nonterminal $\triangledown$-tentacle is attached to a vertex labelled Leaf. This edge abstractly

represents both fields n and p as depicted by the grey nodes within the nonterminal box and the corresponding pointers.

External vertices can be considered to be references to vertices outside the graph and their outgoing pointers are either all inside the graph and the external vertex is therefore an entrance-vertex ($\triangledown$) as we can enter the graph from this node, or are all outside the graph and the external vertex is annotated as reduction-vertex ($\blacktriangle$). In the latter case there should be no $\triangledown$-tentacle connected to the vertex.

Edges have to connect vertices labelled by types that correspond to the type sequence of the edge label. Terminal edges represent fields. For each field we have a well-defined source and target type, as defined by its type sequence. As fields are also defined for subtypes, the source vertex could be labelled with a corresponding subtype as well. The same holds for target vertices.

(a) Terminal edge. (b) Nonterminal edge.

Figure 4.5: Edges and their connected vertices

Figure 4.5(a) depicts a terminal edge labelled with the field name *Inner.left*. The type sequence for *Inner.left* is $\mathsf{Inner}_\triangledown\mathsf{Node}_\blacktriangle$. As there are no subtypes defined for Inner the source vertex has to be of type Inner, while the target vertex could be labelled with Node or a subtype, i.e. with Inner or Leaf.

For nonterminal tentacles we distinguish between $\triangledown$- and $\blacktriangle$-tentacles. As a vertex connected to a nonterminal $\triangledown$-tentacle cannot be connected to further $\triangledown$-tentacles, the $\triangledown$-tentacle has to represent exactly the fields defined for the corresponding tentacle type (see Figure 4.4(b)). This only holds if the type of the tentacle is exact the one of the connected vertex, since in gerneral subtypes define more fields and supertypes less. The same restriction does not hold for nonterminal $\blacktriangle$-tentacles, which represent incoming pointers only. These can also be connected to vertices of subtypes.

Figure 4.5(b) depicts a nonterminal edge labelled by X, that has a $N_\triangledown$- as well as a $N_\blacktriangle$-tentacle. The $N_\triangledown$-tentacle, which represents uniquely a *parent*-pointer, can only be connected to a vertex labelled by Node, because for Object the *parent*-pointer is not defined and for Inner and Leaf additional fields are defined. The $N_\blacktriangle$-tentacle, however, can be connected to a vertex labelled by Node as well as by Inner or Leaf.

Considering the above observations we get the following definition of heap configurations. Within the definition we use the shortcut $\triangledown_H(v)$ that for a given heap configuration H and one of its vertices $v \in V_H$ represents the set of edges connected to the vertex v through an entrance tentacle, i.e. $\triangledown_H(v) = \{e \in E_H \mid (\exists i \in \mathbb{N} : \mathit{types}(\mathrm{lab}(e))[i] \in \mathbb{T}_\triangledown \wedge \mathrm{att}(e)[i] = v\}$.

Definition 4.2 (Heap Configuration):
A heap configuration *over a heap alphabet* Σ *is a tuple* $H = (V, E, \mathrm{lab}, \mathrm{type}, \mathrm{att}, \mathrm{ext})$, *where* V *is a set of vertices,* E *a set of edges. Function* $\mathrm{lab} : E \to \mathbb{F}_\Sigma \cup N_\Sigma$ *determines the edge labels and* $\mathrm{type} : V \to \mathbb{T}_\Sigma$ *the vertex labels. The function* $\mathrm{att} : E \to V^*$ *maps each hyperedge to a sequence of attached vertices and* $\mathrm{ext} \in V^\star_{\blacktriangle\triangledown}$ *is a (possibly empty) sequence of pairwise distinct external vertices.*

For edge $e \in E$ *we require that:*

$$\begin{array}{lllll} & lab(e) \in \mathbb{F}_\Sigma & \Rightarrow & types_\Sigma(lab(e)) \succeq & \mathrm{type}(\mathrm{att}(e)) \quad (1) \\ \wedge & lab(e) \in N_\Sigma & \Rightarrow & types_\Sigma(lab(e)) \succeq_N & \mathrm{type}(\mathrm{att}(e)) \quad (2) \end{array}$$

where $t_\blacktriangle \succeq_N t'_\blacktriangle$ *iff* $t \succeq t'$ *and* $t_\triangledown \succeq_N t'_\triangledown$ *iff* $t = t'$.

For $v_\blacktriangle \in ext$ *we require* $\nabla_H(v) = \emptyset$*, whereas for* $v \in V$ *such that* $v_\blacktriangle \notin ext$ *we require:*

$$\begin{array}{ll} & \mathrm{lab}(\nabla_H(v)) = \mathit{fields}_\Sigma(\mathrm{type}(v)) \quad \wedge \quad x, y \in \nabla_H(v) \Rightarrow \mathrm{lab}(x) \neq \mathrm{lab}(y) \quad (3) \\ \vee & \mathrm{lab}(\nabla_H(v)) \subseteq N \ \wedge \ |\nabla_H(v)| = 1 \quad (4) \end{array}$$

The set of all heap configurations over Σ *is denoted by* HC_Σ.

The definition reflects what we discussed before. We distinguish between terminal edges, whose tentacles should be connected to a vertex of a subtype of the tentacle type (1) and nonterminal edges, where we differ between $\triangledown$- and $\blacktriangle$-tentacles. For $\triangledown$-tentacles we require that they are connected to a vertex of the same type as the tentacle, while $\blacktriangle$-tentacles can be connected to any vertex of a subtype (2). A vertex can be connected to either every non-reduction terminal tentacle defined by the type (3) or a single nonterminal $\triangledown$-tentacle (4).

We do not distinguish between isomorphic heap configurations.

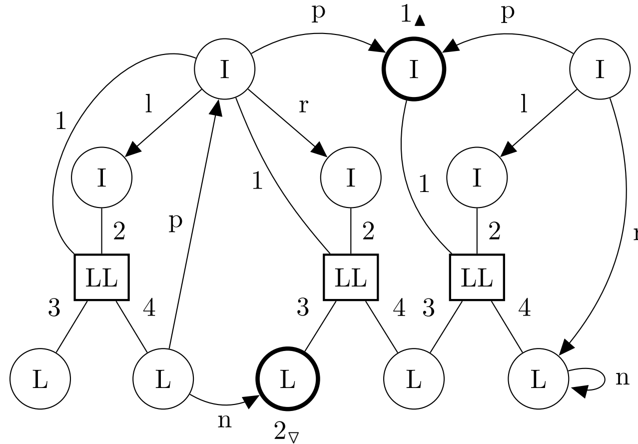


Figure 4.6: A heap configuration.

Example 4.3: Heap configuration

In Figure 4.6 a heap configuration over the heap alphabet from Example 4.2 is given, extended by the nonterminal LL with types(LL) = $I_{\blacktriangle} I_{\triangledown} L_{\triangledown} L_{\blacktriangle}$*. Nonterminal edges labelled by LL represent trees with linked leaves. The tentacles are connected to the root node (2) of a subtree, its parent node (1), the leftmost leaf (3) and the n-reference (4) of the rightmost leaf of the subtree. A corresponding grammar for LL is given in Section 4.2. The labelling of external vertices is extended by an* $\triangledown\blacktriangle$*-index to mark them as* $\triangledown$*- or* $\blacktriangle$*-vertices. As the first external is a* $\blacktriangle$*-node, it has no outgoing edges and is connected exclusively to* $\blacktriangle$*-tentacles, whereas the second one has abstracted outgoing edges represented by the* $\triangledown$*-tentacle* $(LL, 3)$.

Note that the right-most leaf of the tree has a next pointer to itself. This represents the end of the list of leaves. In Java we would use a null*-pointer to mark the end of the leaf-list. So far we do not represent* null*-pointers but we introduce the concept in Section 5.3.1.*

Given a heap configuration H we use the notation $E_H^{\mathbb{F}} = \{e \in E_H \mid \mathrm{lab}_H(e) \in \mathbb{F}\}$ to denote the set of all terminal edges, and E_H^N for the set of nonterminal edges correspondingly. We use $\Sigma_{\mathbb{F}}$ to denote a heap alphabet Σ without nonterminals, i.e. $N = \emptyset$. If a heap configuration does not contain nonterminal edges ($E_H^{\mathbb{F}} = E_H$), i.e. is defined over a heap alphabet $\Sigma_{\mathbb{F}}$, we call it *concrete* ($H \in \mathrm{HC}_{\Sigma_{\mathbb{F}}}$), otherwise *abstract* ($H \in \mathrm{HC}_{\Sigma}$). Note that in a concrete heap configuration all vertices are concrete.

4.2 Data Structure Grammars

In Chapter 2 we introduced hyperedge replacement grammars to describe sets of graphs. These grammars are based on replacement operations that replace edges by graphs. In Section 2.3 we defined replacement for untyped graphs (Definition 2.6 on page 19). The following definition extends the replacement to include also the type information of vertices.

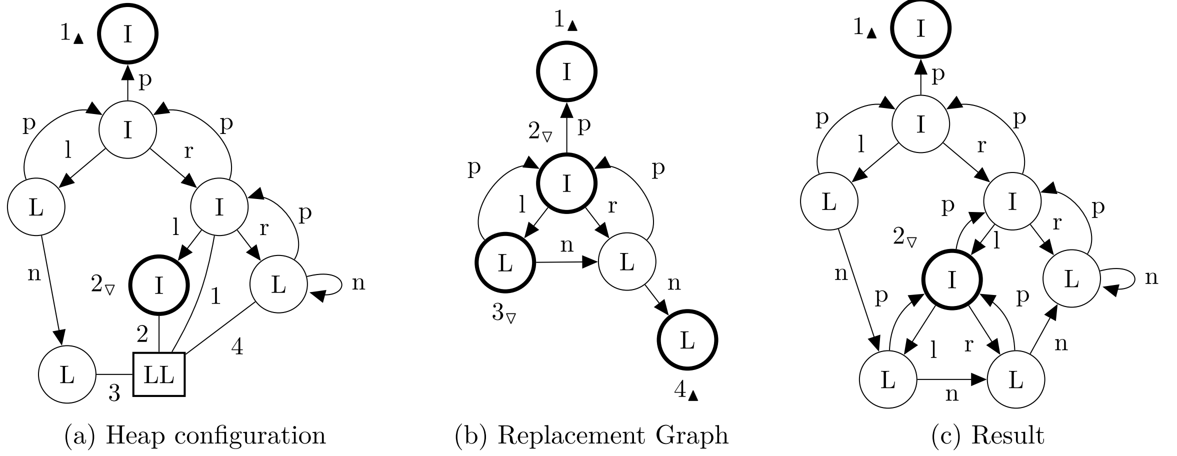
Definition 4.3 (Typed Hyperedge Replacement):

Given hypergraphs $H, K \in HC_{\Sigma}$ *and an edge* $e \in E_H$ *with* $|att_H(e))| = |ext_I|$*, the replacement of* e *in* H *by* K *is defined as* $I = H[K/e] = (V_I, E_I, \mathrm{lab}_I, \mathrm{type}_I, \mathrm{att}_I, \mathrm{ext}_I)$*:*

$$
\begin{array}{lll}
V_I = V_H \uplus (V_K \setminus ext_K) & & E_I = (E_H \setminus \{e\}) \uplus E_K \\
\mathrm{type}_I = \mathrm{type}_H \uplus \mathrm{type}_K \restriction V_I & & \mathrm{lab}_I = \mathrm{lab}_H \restriction E_I \uplus \mathrm{lab}_K \\
\mathrm{att}_I = \mathrm{att}_H \uplus \mathrm{att}_K \circ (id_{V_K} \restriction V_I \cup \{ext_K(i) \mapsto \mathrm{att}_H(e)(i) \mid i \in [1, |\mathrm{ext}_I|]\}) & & \\
\mathrm{ext}_I = \mathrm{ext}_H & &
\end{array}
$$

Example 4.4:

Consider the heap configuration H from Figure 4.7(a) containing exactly one nonterminal edge e labelled with LL. The rank of LL is equal to the number of external vertices of the concrete heap configuration K in Figure 4.7(b), thus we can replace e by K and get $I = H[K/e]$, depicted in Figure 4.7(c).



(a) Heap configuration (b) Replacement Graph (c) Result

Figure 4.7: Hyperedge replacement for heap configurations.

The replacement result in Example 4.4 is a heap configuration. The reason for this is that $\mathit{types}(LL) \preceq_N \mathrm{lab}_I(\mathit{ext}_I)$. This can be generalised and leads to the following theorem.

Theorem 4.1 (Closure of replacements):
Heap configurations are closed under replacements, i.e. given $H, K \in HC_\Sigma$, $e \in E_H$ with $\mathit{types}(\mathrm{lab}_H(e)) \preceq_N \mathrm{type}(\mathit{ext}_K)$, the replacement $H[K/e] \in HC_\Sigma$ is a heap configuration.

Proof. In order to prove that the result $I = H[K/e]$ is a heap configuration we show that the graph I fulfils the requirements:

For any edge $e \in E_I$ it holds that:

$$\begin{aligned} & \mathrm{lab}_I(e) \in \mathbb{F}_\Sigma \Rightarrow \mathit{types}_\Sigma(\mathrm{lab}_I(e)) \succeq \mathrm{type}_I(\mathrm{att}_I(e)) && (1)\\ \wedge\ & \mathrm{lab}_I(e) \in N_\Sigma \Rightarrow \mathit{types}_\Sigma(\mathrm{lab}_I(e)) \succeq_N \mathrm{type}_I(\mathrm{att}_I(e)) && (2) \end{aligned}$$

and for any $v_\blacktriangle \in \mathrm{ext}_I$ that $\triangledown_I(v) = \emptyset$, whereas for $v \in V_I$ with $v_\blacktriangle \notin \mathrm{ext}_I$ it holds that:

$$\begin{aligned} & \mathrm{lab}_I(\triangledown_I(v)) = \mathit{fields}_\Sigma(\mathrm{type}_I(v)) \ \wedge\ x, y \in \triangledown_I(v) \Rightarrow \mathrm{lab}_I(x) \neq \mathrm{lab}_I(y) && (3)\\ \vee\ & \mathrm{lab}_I(\triangledown_I(v)) \subseteq N \ \wedge\ |\triangledown_I(v)| = 1 && (4) \end{aligned}$$

We prove properties (1) and (2) first:

As $E_I = (E_H \setminus \{e\}) \uplus E_K$, any edge $e \in E_I$ is either from E_H or E_K. As $H \in \mathrm{HC}_\Sigma$ it holds that $\mathit{types}(\mathrm{lab}_H(e')) \succeq \mathrm{type}_H(\mathrm{att}_H(e'))$ for all $e' \in E_H^{\mathbb{F}}$ and $\mathit{types}(\mathrm{lab}_H(e')) \succeq_N \mathrm{type}_H(\mathrm{att}_H(e'))$ for all $e' \in E_H^N$. The construction of I preserves the labelling and the attachment for edges $e' \in E_H \setminus \{e\}$ as well as the types of the vertices from V_H and we get $\mathit{types}(\mathrm{lab}_I(e')) \succeq \mathrm{type}_I(\mathrm{att}_I(e'))$, respectively $\mathit{types}(\mathrm{lab}_I(e')) \succeq_N \mathrm{type}_I(\mathrm{att}_I(e'))$ for any $e' \in E_I$, immediately.

If $e' \in E_I$ is not from H ($e' \notin E_H$) then e' is from K ($e \in E_K$) and it holds that $\mathit{types}(\mathrm{lab}_K(e)) \succeq \mathrm{type}_K(\mathrm{att}_K(e))$ as $K \in \mathrm{HC}_\Sigma$. The construction of I preserves labelling of edges $e \in E_I$ and the types of the vertices $v \in V_I$. We get $\mathit{types}(\mathrm{lab}_I(e)) \succeq \mathrm{type}_I(\mathrm{att}_K(e))^{(\star)}$. From the definition of $\mathrm{att}_I(e)$ for $e \in E_K$ and $\mathit{types}(\mathrm{lab}_H(e)) \succeq \mathrm{type}_K(\mathrm{ext}_K)$ it follows that $\mathrm{type}_I(\mathrm{att}_K(e)) \succeq \mathrm{type}_I(\mathrm{att}_I(e))$ and together with $(\star)$ we get $\mathit{types}(\mathrm{lab}_I(e)) \succeq \mathrm{type}_I(\mathrm{att}_I(e))$.

It remains to prove properties (3) and (4):

As $H, K \in \mathrm{HC}_\Sigma$ and by the construction of I it follows that for all edges $e \in E_I \setminus [\mathrm{att}_H(e)]$ either property (3) or (4) holds, as they as well as their context are carried over from H and K. For vertex $v \in [\mathrm{att}_H(e)]$, and $i \in \mathbb{N} : \mathrm{att}_H(e)[i] = v$ we get $\triangledown_I(v) = \triangledown_K(\mathit{ext}_K[i]) \cup \triangledown_I(v) \setminus \{e\}$ due to the merging, realised by the adjustment of att_K. We distinguish the following two cases:

1. $\mathit{types}(\mathrm{lab}_I(e))[i] \in \mathbb{T}_\blacktriangle$, i.e. v and e are connected through a $\blacktriangle$-tentacle, then $e \notin \triangledown_H(v)$ and $\triangledown_K(\mathrm{ext}_K[i]) = \emptyset$, thus $\triangledown_I(v) = \triangledown_H(v)$, fulfilling (3) or (4).

2. $\mathit{types}(\mathrm{lab}_I(e))[i] \in \mathbb{T}_\triangledown$, i.e. v and e are connected through an $\triangledown$-tentacle then $e \in \triangledown_H(v) = \{v\}$ (3) and thus $\triangledown_I(v) = \triangledown_K(\mathrm{ext}_K[i])$, fulfilling (3) or (4).

□

Definition 4.4 (Data Structure Grammar):
Any $G \in HRG_\Sigma$ satisfying $\forall X \to R \in G.\, \mathit{types}(X) \preceq_N \mathrm{lab}_R(\mathrm{ext}_R)$ is a data structure grammar *(DSG).*

We denote the set of all data structure grammars over Σ by DSG_Σ.

The definitions and notations for derivations and the language of a data structure grammar are the same as for HRGs.

Corollary 4.1 (Languages over DSGs)
Given a data structure grammar $G \in DSG_\Sigma$ and $H \in HC_\Sigma$ it holds that every derivable graph is a heap configuration, i.e. $H \Rightarrow_G^* H' \Rightarrow H' \in HC_\Sigma$.

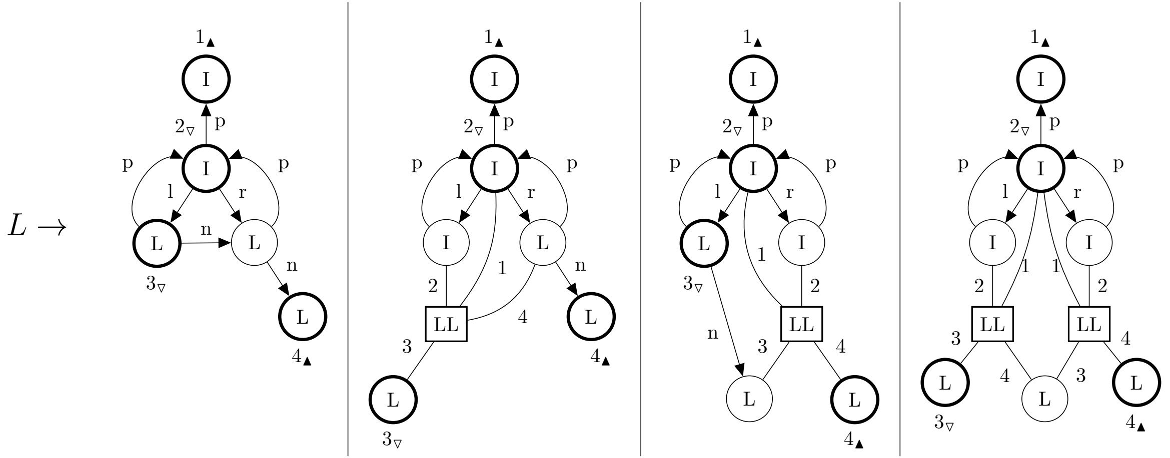


Figure 4.8: DSG for trees with linked leaves.

Example 4.5:

Figure 4.8 depicts a DSG for trees with linked leaves and parent pointers. The DSG consists of four rules for nonterminal L with types$(LL) = I_{\blacktriangle} I_{\triangledown} L_{\triangledown} L_{\blacktriangle}$*, introduced in Example 4.3. Every right-hand side is a heap configuration with* type(ext) = *types*(LL).

The rules recursively define the data structure. The smallest tree represented by LL is a tree where both child nodes of the root node are leaves (first rule). *Larger trees are recursively defined as trees where either one* (second and third rule) *or both* (fourth rule) *children of the root node are trees, with properly linked leaves.*

As for HRGs, we define languages for a start configuration.

Definition 4.5 (Language of a heap configuration):

For $G \in DSG_\Sigma$ *and* $H \in HC_\Sigma$*, the language of* H *with respect to* G*, denoted* $L_G(H)$*, is defined as* $\{H' \in HC_\Sigma \mid H \Rightarrow^*_G H'\}$*. For sets of heap configurations* $M \subseteq HC_\Sigma$ *we define* $L(M) = \bigcup_{H \in M} L(H)$.

It follows from Corollary 4.1 that the language of an abstract heap configuration over a DSG contains concrete heap configurations only. It remains to show that the restrictions in the definition of DSGs do not impair the expressiveness, i.e. that the languages representable by data structure grammars are exactly the languages over heap configurations ($\subseteq \mathrm{HC}_\Sigma$) representable by hyperedge replacement grammars. This issue is addressed in the following.

Example 4.6: Data structure grammar for linked lists

Figure 4.9 depicts the rules of a DSG for linked lists. The rules are, up to the type labels, the same as the ones given in Section 3.3 (Figure 3.5 on page 45). The type E

represents list elements, and there is a single field label n representing next-pointers. The type sequence for the label n is $types(n) = E_{\triangledown}E_{\blacktriangle}$. *The type sequence of the single nonterminal L is* $types(L) = E_{\triangledown}$.

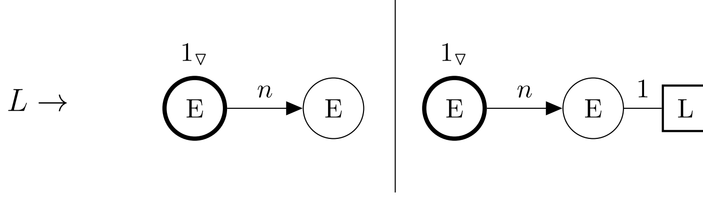


Figure 4.9: Data structure grammar for linked lists.

Obtaining a DSG from a given HAG is not always as easy as in Example 4.6. Figure 4.10 depicts the rules of the HAG for doubly linked lists as introduced in Section 3.3 (Figure 3.6 on page 46). List elements of a doubly linked list have a next(n) as well as a previous(p) pointer. Simply adding type labels to the vertices does not lead to a data structure grammar. That is as at the external vertices only one terminal tentacle is present instead of both, as required. However, we can construct a corresponding DSG, as stated by the following theorem.

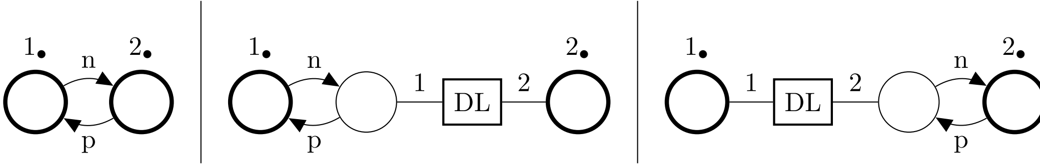


Figure 4.10: Rule graphs of the HRG for doubly linked lists.

Theorem 4.2 (Expressiveness of data structure grammars):
Let a heap alphabet $\Sigma = (\mathbb{F}, \emptyset, types)$ *and the ranked alphabet* $\Sigma_R = (\mathbb{F}_\Sigma, N, \mathrm{rk})$ *where* rk *is the projection of* $types_\Sigma$ *to the sequence length, i.e.* $rk := [\, x \mapsto |types_\Sigma(x)|\,]$ *for all* $x \in \mathbb{F}_\Sigma$. *Then for every* $G \in HAG_{\Sigma_R}$ *exists a heap alphabet* $\Sigma' = (\mathbb{F}_\Sigma, N, types)$ *and* $G' \in DSG_{\Sigma'}$ *such that for every* $K \in HG_{\Sigma_R}$ *exists a* $K' \in HC_{\Sigma'}$ *with:*

$$L_{G'}(K') = \{H \in HC_\Sigma \mid (E_H, V_H, \mathrm{lab}_H, \mathrm{att}_H, \mathrm{ext}_H) \in L_G(K)\}.$$

Following the theorem, for any heap abstraction grammar that – up to the missing vertex types – generates heap configurations, there is a DSG generating exactly the heap configurations that we get by adding the types to the vertices. In fact the theorem is a little bit stronger, stating that if for some of the derivable heap graphs there exists a typing that transforms them to valid heap configurations, then there is a DSG describing exactly these heap configurations.

We prove the theorem by giving a construction for the grammar G'. In Example 4.7 the construction is performed for the heap abstraction grammar defined by the rules from Figure 4.10 for doubly linked lists.

Proof. Note that as $G \in \mathrm{HAG}_{\Sigma_R}$ it is typed (Definition 3.7). Without loss of generality we assume G to be in LGNF (Definition 3.8) and that there is a heap graph $H \in L_G(S)$ for which a type function *type* exists such that $(V_H, E_H, \mathrm{lab}_H, \mathrm{type}, ext_H)$ is a heap configuration. Otherwise the theorem is trivially fulfilled and G' is the empty grammar.

Recall that for typed grammars there is a fixed set of outgoing pointers $\mathrm{out}(X,i)$[1] for each nonterminal tentacle $(X,i) \in Tent_{\Sigma_R}$. For DSGs the set $\mathrm{out}(X,i)$ corresponds to the type of the tentacle (X,i), i.e. $\mathrm{out}(X,i) = \mathit{fields}(\,\mathit{types}(X,i)\,)$. The idea behind the construction is to add outgoing edges to the external vertices of the rule graphs, such that the resulting extensions of the set $\mathrm{out}(X,i)$ comply with one of the types from $\mathbb{T}$, that then will be used as type of the tentacle. As it is not necessarily clear to which type a tentacle should be extended, we extend it to any type that could be obtained by adding edges.

Given an untyped hypergraph $H \in \mathrm{HG}_\Sigma$ the possible types of each vertex $v \in V_H$ are restricted by the connected ▽- and ▲-tentacles, as tentacles have to be connected to a subtype of its own type. If there are several types that are subtypes of all connected tentacles, then there is a minimal one which is the least upper bound of the tentacle types. The maximal type of tentacles is a compositional property $p = (\eta, V)$ (Definition 2.16 on page 32), where each graph $H \in \mathrm{HG}_\Sigma$ is mapped by $\eta : \mathrm{HG}_\Sigma \to V$ to a value from $V = \mathbb{T}^\star$, a sequence of types representing the maximal type of each external node, i.e. $\eta(H) = t$ with $t \in \mathbb{T}^{|ext_H|}$ and $t[i] = \sqcup Tent_H(ext_H[i])$. Here $\sqcup$ is the least-upper-bound operator for $(\mathbb{T}, \preceq)$. The splitting grammar $G_S \in \mathrm{HRG}_{\Sigma_S}$ is defined over the alphabet $\Sigma_S = (N_S, \mathbb{F}, \mathrm{rk}_\Sigma)$ with $N_S \subseteq N \times \mathbb{T}^\star$, i.e. any nonterminal from N_S is of the form (X,t).

We start the construction at the splitting grammar G_S which we adapt to the data structure grammar $G' \in \mathrm{DSG}_{\Sigma_T}$ in three steps.

Step 1: Adding outgoing edges to external vertices

In the first step we extend the outgoing edges of the external vertices of the rule graphs to meet the possible types.

When graphs are within a context then external vertices can be connected to additional edges. The additional edges can induce further restrictions to the possible type of a vertex, therefore we use an alphabet with an extended set of nonterminals with more restrictive types.

$$N_T = \{(X,t') \in N \times \mathbb{T}^\star \mid (X,t) \in N_S \;\wedge\; t' \preceq t\}.$$

[1]The method that we refer to by out is named type in Section 3.2. We use the name out here to avoid confusion with the type mapping of heap configurations that is also called type.

The initial grammar is defined as

$$G_T = \{(X, t') \to R \mid (X, t) \to R \in G_S \ \land \ t' \preceq t\}.$$

By definition the external vertices of the rule graphs of a data structure grammar have either outgoing edges for any field defined for its type, $\mathrm{out}(X, i) = \mathit{fields}(\mathit{types}(X)[i])$ or none $\mathrm{out}(X, i) = \emptyset$ (reflecting $\triangledown$- and $\blacktriangle$-tentacles). Correspondingly for each nonterminal $(X, t) \in N_T$ we determine the fields for which a outgoing edge is missed, i.e. fields that for some $1 \leq i \leq |t|$ are defined for the type $t[i]$ but are not element of $\mathrm{out}(X, i)$.

$$\mathit{missing}(X, t) = \{(i, f) \mid \mathrm{out}(X, i) \neq \emptyset \land f \in \mathit{fields}(t[i]) \setminus \mathrm{out}(X, i)\}.$$

For each missing field f we add to each corresponding rule an edge from the external vertex i to a new external vertex (i, f). Figure 4.11 depicts this for one $(i, f) \in \mathit{missing}(X, t)$. For rule $(X, t) \to R \in G_T$ we get the rule $(X, t) \to R'$ with:

$$\begin{array}{lcllr}
V_{R'} & = & V_R & \cup & \{v_{i,f} \mid (i, f) \in \mathit{missing}(X, t)\} \\
E_{R'} & = & E_R & \cup & \{e_{i,f} \mid (i, f) \in \mathit{missing}(X, t)\} \\
\mathrm{lab}_{R'} & = & \mathrm{lab}_R & \cup & \{e_{i,f} \mapsto f \mid (i, f) \in \mathit{missing}(X, t)\} \\
\mathrm{type}_{R'} & = & \mathrm{type}_R & \cup & \{v_{i,f} \mapsto \mathit{types}(t)[2] \mid (i, f) \in \mathit{missing}(X, t)\} \\
\mathrm{att}_{R'} & = & \mathrm{att}_R & \cup & \{e_{i,f} \mapsto \mathit{ext}_R[i]\ v_{i,f} \mid (i, f) \in \mathit{missing}(X, t)\} \\
\mathrm{ext}_{R'} & = & \mathrm{ext}_R & \circ & \{v_{i,f} \mid (i, f) \in \mathit{missing}(X, t)\}.
\end{array}$$

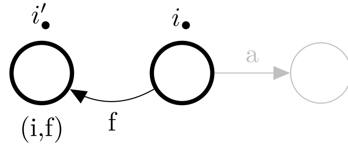


Figure 4.11: New introduced edge and vertex for $(i, f) \in \mathit{missing}(X, t)$.

Step 2: Replacing nonterminals

After adding external vertices in the previous step we need to adapt the ranks of the nonterminals accordingly. Given an edge e labelled by nonterminal (X, t) we add one tentacle for each (i, f) in $\mathit{missing}(X, t)$ to e and connect it to the vertex v that is the f-successor of the vertex at tentacle i, i.e. $\exists e \in \triangledown_H(v) \ \land \ \mathit{lab}(e) = f \ \land \ \mathit{att}[e][2] = v$ (see Figure 4.12). The edge e has to exist as otherwise there would be no proper type for the node v. However, the edge could be abstracted within a further nonterminal. The edge can then be concretised in a single derivation step (possible because G in LGNF). If this is the case we get one rule for any resulting concretisation.

After adding the additional tentacle we remove the terminal edge e', as this one is now included in the abstracted part represented by e.

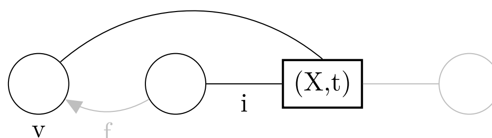


Figure 4.12: Adding one tentacle for each $(i, f) \in \mathit{missing}(X, t)$.

Step 3: Adding type information

In a last step we transfer the untyped alphabet Σ_T into a heap alphabet by replacing the ranking function by a typing function. Any tentacle $((X, t), i)$ is of type $t[i]$ and is a $\blacktriangle$-tentacle if it accrued by adding external nodes in the first step or if $\mathrm{out}(X, i) = \emptyset$, a $\triangledown$-tentacle otherwise. Given the type of the nonterminals we can determine directly if an external vertex is a $\triangledown$- or $\blacktriangle$-vertex. The typing of vertices can be determined by the set of $\triangledown$-tentacles connected to it.

Given a start graph S we can transform it into the language equivalent typed start graph S' by applying the second and third step. $\square$

The above construction is illustrated in the following example for doubly linked lists.

Example 4.7:

We use the above construction to build a data structure grammar for doubly linked lists. The grammar contains a single nonterminal DL of rank two with $\mathit{out}(DL, 1) = n$ *and* $\mathit{out}(DL, 2) = p$*. In Figure 4.10 the rule graphs of the grammar are given. Note that the grammar is typed and in LGNF.*

Let us consider a heap alphabet with a single type Element *(*E*) ,fields* $\mathbb{F} = \{n, p\}$ *and* $\mathit{types}(n) = \mathit{types}(p) = E_\triangledown E_\blacktriangle$*. For the nonterminal DL the missing set is given by* $\mathit{missing}(DL) = \{(1, p), (2, n)\}$*.*

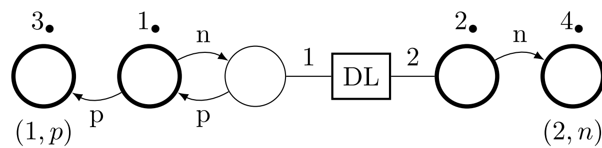
In the ***first step*** *we add additional external nodes and terminal edges as defined by* $\mathit{missing}(DL)$*. The result for the second rule of Figure 4.10 is given in Figure 4.13(a).*

In the ***second step*** *we add the new tentacles and we remove the terminal edges represented by them. The resulting graph is depicted in Figure 4.13(b).*

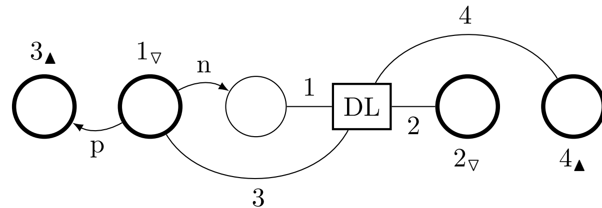
In the ***third step*** *we now can determine the type of each vertex, as well as the* $\triangledown\blacktriangle$*-information for external vertices. The result is depicted in Figure 4.13c.*

The other two resulting rules of the grammar are given in Figure 4.14.

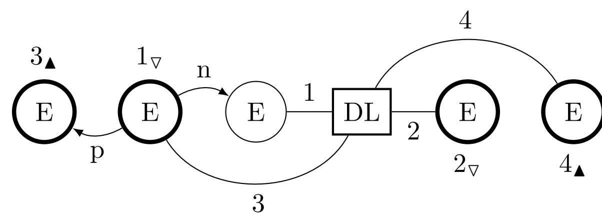
For the other grammars given in Section 3.3 corresponding data structures can be determined by simply adding type information for the vertices. The heap abstraction grammar for trees does not generate any heap graphs that can be transformed to heap configurations, as the leaves are tree objects but do not have outgoing pointers. In



(a) First step of the transformation.



(b) Second step of the transformation.



(c) Third step of the transformation.

Figure 4.13: Construction of a data structure grammar.

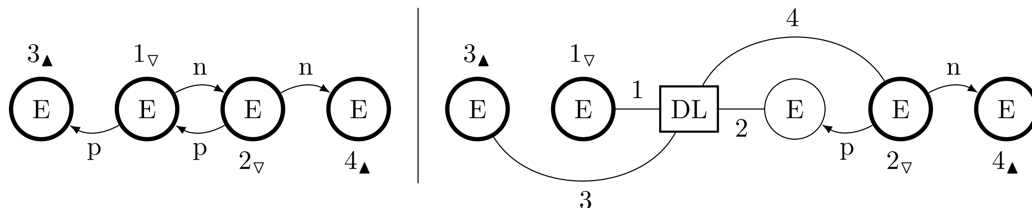


Figure 4.14: The other rules of the data structure grammar.

Figure 5.2 we give a data structure grammar for binary trees that uses an additional vertex to represent null-pointers.

4.3 Concretisation and Abstraction

As before in Chapter 3 for heap graphs we define concretisation and abstraction of heap configuration through forward, respectively backward application of grammar rules. The typing of hypergraphs induces some additional restrictions to the backward application, resulting in a more restricted embedding definition that we introduce in Section 4.3.2. In the same section we introduce an abstraction function based on the successive application of backward applications. We will extend this abstraction function towards a parameterised version allowing us to define parts of the heap that should not

be abstracted, e.g. to preserve information that we consider to be important in the future. In Section 4.3.1 we consider a concretisation function for heap configurations. We can reuse the one given in Section 3.4.3, however, the special properties of heap configurations allow us to significantly simplify the vertex concretisation.

4.3.1 Concretisation

In Chapter 3, we introduced the notions of concrete and abstract vertices. Recall that for a heap configuration $H \in \mathrm{HC}_\Sigma$, a vertex $v \in V_H$ is concrete if all attached non-reduction tentacles are terminal, and abstract otherwise. Based on the LGNF we introduced in Section 3.4.3, a tentacle concretisation function that allows us to concretise tentacles in a single step. We realised concretisation of vertices by successive concretisation of tentacles connected to a vertex.

For heap configurations $H \in \mathrm{HC}_\Sigma$ any vertex v is connected to at most one non-reduction nonterminal tentacle. It holds that either $\triangledown(v) \subseteq \mathbb{T}$, thus all non-reduction tentacles are terminal ones, thus the vertex is concrete, or that $\triangledown(v) \subseteq N$ is a singleton and the vertex is abstract. Therefore vertex concretisation for heap configuration reduces to a single hyperedge replacement.

Given $G \in \mathrm{DSG}_\Sigma$ in LGNF, $H \in \mathrm{HC}_\Sigma$ and an abstract vertex $v \in V_H$, the *concretisation set* of vertex v is defined as

$$\mathit{conc}_G(H, v) = \{H[e/R] \mid \triangledown_H(v) = \{e\} \wedge \mathrm{att}[i](e) = v \wedge (\mathrm{lab}(e) \to R) \in G_{(\mathrm{lab}(e),i)}\}.$$

If v is a concrete vertex, then $\mathit{conc}_G(H, v) = \{H\}$. Note that, given by the tentacle concretisation, for any $G \in \mathrm{DSG}_\Sigma$, $H \in \mathrm{HC}_\Sigma$ and $v \in V_H$ it holds that $L_G(H) = L_G(\mathit{conc}_G(H, v))$.

4.3.2 Abstraction

Abstraction, defined as the inverse of concretisation, can also be defined explicitly by searching for embeddings of rule graphs and replacing them by the corresponding nonterminals (see Section 2.4). We adapt the embedding definitions from Definition 2.11 on page 24 to take type information into account. This yields a typed version of embedding that equals Definition 2.11 on page 24 up to the typing restrictions.

Definition 4.6 (Embedding (typed)):
Given $I, H \in HC_\Sigma$ an embedding *of I in H consists of two mappings $emb_V : V_I \to V_H$, and $emb_E : E_I \to E_H$, fulfilling the properties from Definition 2.11 and additionally:*

$$\begin{aligned} \mathrm{type}_I(v) &\preceq \mathrm{type}_H(emb_V(v)) \quad \forall v \in \{v \in V_I \mid v_\blacktriangle \in \mathrm{ext}_I\} \\ \mathrm{type}_I(v) &= \mathrm{type}_H(emb_V(v)) \quad \forall v \in V_I \setminus \{v \in V_I \mid v_\blacktriangle \in \mathrm{ext}_I\} \end{aligned}$$

Given $I, H \in HC_\Sigma$ $Emb(I, H)$ denotes the set of all embeddings of I in H.

We also adapt Definition 2.12 on page 25 for hypergraph replacement. Given $G \in \mathrm{DSG}_\Sigma, I, H \in \mathrm{HC}_\Sigma$, $emb \in Emb(I, H)$ and $X \in N$, we replace I in H which results in $replace(I, H, emb, X) = K$, with V_K, E_K, lab_K, att_K, and ext_K as in Definition 2.12 and $\mathrm{type}_K = \mathrm{type}_H \restriction V_K$.

$abstr_G(H) = \{replace(R, H, emb, X) \mid X \to R \in G, emb \in Emb(R, H)\}$ are the heap configurations we get via one abstraction step ($H' \in abstr_G(H)$ *iff* $H' \Rightarrow_G H$), $abstr^\star_G(H)$ is the transitive reflexive closure ($H' \in abstr^\star_G(H)$ *iff* $H' \Rightarrow^\star_G H$). We denote by $maxAbstr_G(H)$ the fully abstracted heap configurations, i.e. configurations for which no further abstraction steps are possible:

$$maxAbstr_G(H) = \{H' \in abstr^\star_G(H) \mid \forall X \to R \in G : Emb(R, H') = \emptyset\}.$$

Note that the set $maxAbstr_G(H)$ in general is not a singleton but finite. If the set $maxAbstr_G(H)$ is a singleton for any $H \in \mathrm{HC}_\Sigma$ then G is *backward confluent* (Definition 2.15).

When we abstract a heap configuration then potentially there are some parts that should be kept concrete as they contain details that will be crucial in the future. We introduce a parameterised version of $maxAbstr_G$, that allows us to define a set of vertices that can be abstracted, i.e. any vertex not part of the set has to kept concrete.

Before we can define the parameterised version of $maxAbstr_G$ we first have to introduce parameterised versions of embeddings and $abstr_G$. Given heap configurations $I, H \in \mathrm{HC}_\Sigma$ and vertices $V \subseteq V_H$, we denote by $Emb(I, H, V)$ the set of all embeddings of I in H restricted to V:

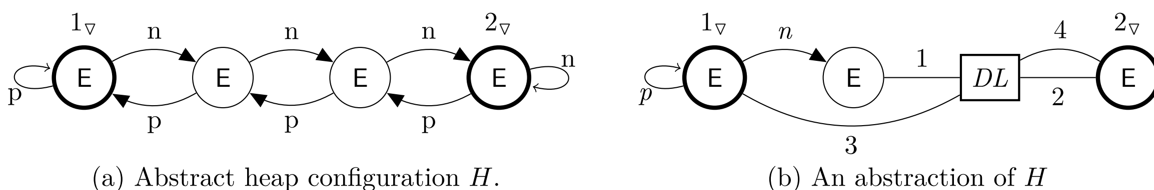
$$Emb(I, H, V) = \{emb \in Emb(I, H) \mid emb(V_I \setminus \{v \in V_I \mid v_\blacktriangle \in ext_I\}) \subseteq V\}$$

That is, $Emb(I, H, V)$ contains any embedding from $Emb(I, H)$, for which the vertices of I are mapped exclusively to vertices from V. We define abstraction steps restricted to V as $abstr_G(H, V) = \{replace(R, H, emb, X) \mid X \to R \in G, emb \in Emb(R, H, V)\}$, and the transitive closure of abstraction steps $abstr^\star_G(H, V)$.

Definition 4.7 (Parameterised abstraction):
Given an abstract heap configuration $H \in HC_\Sigma$ and a set of vertices $V \subseteq V_H$, the set of maximal abstractions *of H restricted to V is defined as:*

$$maxAbstr_G(H, V) = \{H' \in \ abstr^\star_G(H) \mid \forall X \to R \in G : Emb(R, H', V) = \emptyset\}.$$

Example 4.8:
Reconsider the data structure grammar for doubly linked lists $G \in DSG_\Sigma$ given by the rules depicted in Figure 4.13(c) and Figure 4.14. In Figure 4.15(a) a heap configuration $H \in HC_\Sigma$ is given as well as in (b) the single element of the set $maxAbstr_G(H, V_H \setminus \mathrm{ext}_H[1])$, i.e. the maximal abstraction of H restricted to any vertex of H less the first external one. The grammar G is backward confluent, thus there is only a single maximal abstraction. Note that as $\mathrm{ext}_H[1]$ was left out of the abstractable set it remains concrete, while the other vertices are all abstracted.



(a) Abstract heap configuration H.

(b) An abstraction of H

Figure 4.15: Parameterised abstraction.

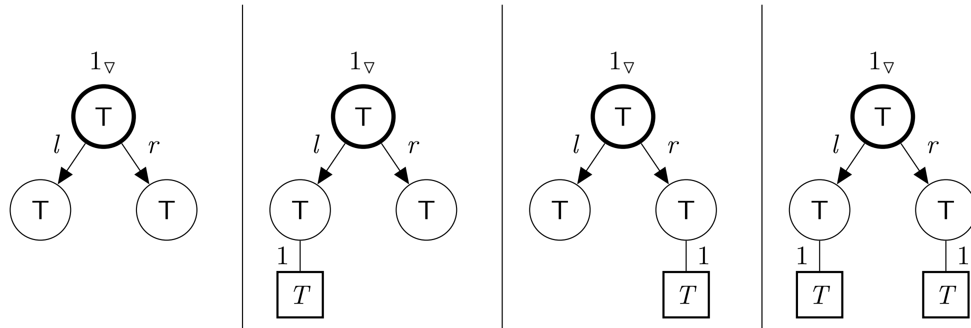
4.4 Conclusions

In this chapter we evolved the heap graphs and heap abstraction grammars from Chapter 3 towards typed graphs and grammars. The introduced types correspond to the classes of Java programs and include also inheritance between classes reflected by a type poset. An important result of this chapter is that, on the one hand, the introduced heap configurations are closed under replacements (Theorem 4.1), such that from an abstract heap configuration only valid heap configurations can be derived. On the other hand, any graph language that represents a valid typed heap (e.g. of a Java program) and that can be expressed by a HRG can also be represented by a DSG (Theorem 4.2).

A convenient side effect of heap configurations and DSGs is that the local concretisation of vertices as introduced in the previous chapter can always be done in a single concretisation step, as all outgoing edges of a vertex are always represented by a single nonterminal.

Reconsider the tree and path-tree grammar that we introduced in Section 3.6. In Figure 4.16 we give the corresponding DSG. The type T corresponds to the Java class Tree as used in the introduction (Figure 1.1a on page 2).

$T \rightarrow$



$P \rightarrow$

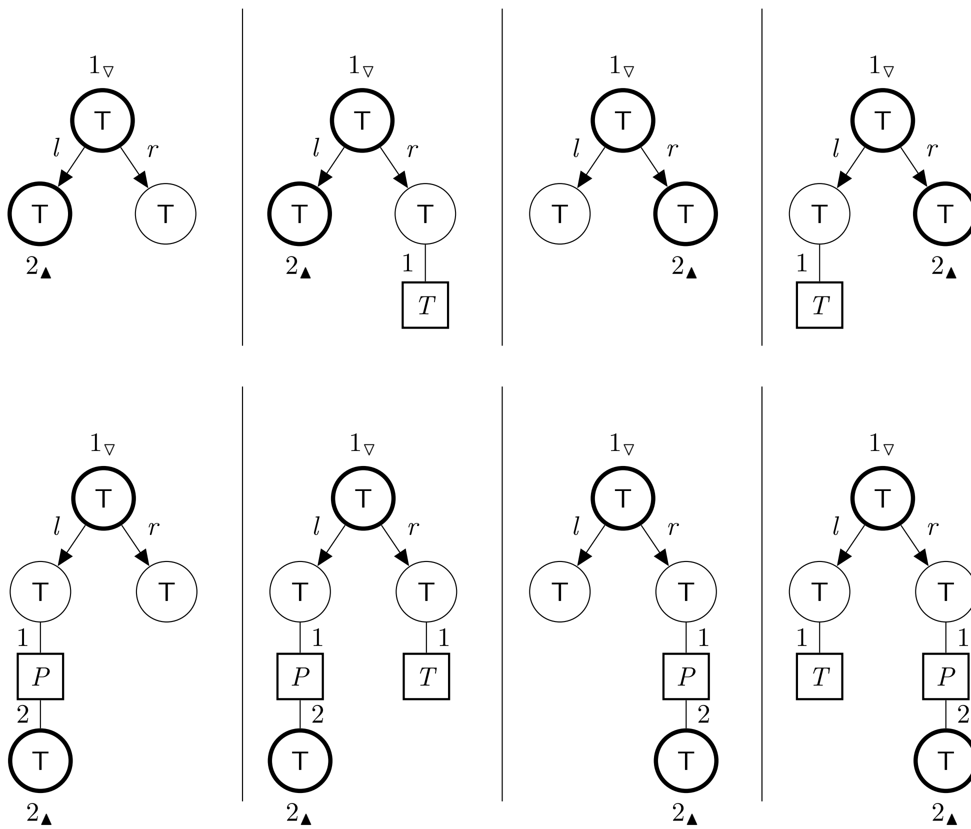


Figure 4.16: DSG for trees and tree-paths.

5 Chapter 5
Abstract JVM

In the previous chapter we considered heap structures of Java programs that were modelled by heap configurations. The state of a Java program, however, is not just determined by the heap. In this chapter we extend the heap representation towards a whole-state representation. The main part of the extension covers the method stack that keeps track of the (possibly recursive) method invocations. Other aspects handled in this chapter are local and global variables, boolean values and *null*-pointers. The second part of this chapter covers the semantics of Java programs and will lead to a transition relation for the hypergraph representations. Instead of analysing Java programs we consider the equivalent but easier to handle Java bytecode. Java programs are compiled to so called class files that contain bytecode that is executed by a *Java Virtual Machine* (JVM).

In Section 5.1 we shortly introduce the parts of the JVM specification that are needed for the understanding of this chapter. In Section 5.2 we then introduce the formal model of the JVM as defined by Stärk *et. al.* [SSB01]. Based on this model we discuss how to model states of a JVM by heap configurations in Section 5.3. Section 5.4 covers the bytecode instructions. We first give a concrete semantics of the instruction in form of transition rules for heap configurations. Based on Theorem 3.6 the concrete semantics is then extended towards an abstract semantics in Section 5.4.3. We close the chapter with some experimental results regarding Lindstrom's algorithm.

5.1 The Java Virtual Machine

The *Java Virtual Machine (JVM)* is an abstract computation machine with a proper set of instructions. This abstract model allows Java programs to work hardware and operating system-independent. There exists several implementations of the JVM for various devices and operating systems. The JVM is specified in the "Java Virtual Machine Specification" [LY99]. This specification is the base of this section.

Programs for the JVM are given in a particular binary format called the class file format. A class file (not necessarily a file) contains JVM instructions (bytecode) and additional information needed for the execution.

The instructions of a JVM consist of an *operand code* (opcode) and a possible empty sequence of parameters. *Opcodes* are represented by a one-byte value allowing a theoretical maximum of 256 opcodes. The opcodes 203 to 253 are not used. In Section 5.4.2 we give the semantic of the instructions in form of transition rules for heap configurations. Instead of the complex set of 205 opcodes, we use a set of *abstract instructions* defined by Stärk *et. al.* [SSB01], that covers the whole instruction set of a JVM. These abstract instruction mainly abstract from the type information that is encoded in most of the instructions. Types supported by the JVM are the ones known from Java: *boolean*, *byte*, *char*, *short*, *int*, *float* and *reference*.

During runtime the JVM has accesses to different data areas. We consider here the *heap* and the *Java Virtual Machine Stack*. Other data areas are used to store the instruction lists and the value of static variables.

A *Java Virtual Machine Stack* is created for each thread (we only consider single threaded programs). The Java Virtual Machine Stack is the analogous of the stack of a conventional language and stores *frames*. Each frame represents a method invocation and holds local variables, partial results and other informations necessary for the execution of the methods. The local variables are stored within an array of variables that is contained in each frame. Depending on their type variables occupy either one or two succeeding entries of the array. We abstract from the array and represent the array as *registers* where each register represents one local variable. The *operand stack* is a LIFO stack that at creation of the frame is empty and is used during the execution to safe intermediate results. Depending on the instruction one or more operands are pop from the operand stack, operate on them and the result is pushed back to the operand stack. The maximum size of the array (i.e. the number of registers) and the maximum depth of the operand stack are determined at compile-time and stored in the class file. Frames are removed from the Java Virtual Machine Stack when the method invocation completes.

Objects as well as their relations are stored in the *heap*. The heap is shared between all Java Virtual Machine Stacks (and their methods). Objects are not explicitly deallocated but removed from the heap by a *garbage collector* if they are no longer accessible.

5.2 A Formal Definition of the JVM

In this section we introduce a formal model of Java bytecode and the JVM based on the model $\mathrm{JVM}_{\mathcal{O}}$ by Stärk et al. [SSB01]. The rest of the chapter will be based on this formal model. Whereas $\mathrm{JVM}_{\mathcal{O}}$ treats primitive data types as integers, floats, and characters, we do not consider them here as we focus our interest on the impact of heap

structures only. Stärk et al. extend their model to one that takes exception handling into account [SSB01]. For the sake of simplicity we do not deal with exceptions here, however, our approach could easily be extended to handle exceptions.

Given a JVM we differentiate between the static environment and the dynamic states of the JVM. Both are introduced in this section.

5.2.1 Static environment of a JVM

A JVM executes programs with respect to a static environment $cEnv : \mathit{Class} \cup \mathit{Interface} \rightarrow \mathit{ClassFile}$ [SSB01]. For each class of a Java program (top-level, inner or anonymous) a separate class file is compiled.

Definition 5.1 (Java class file [SSB01]):
A Java class file $cf = (name, isInterface, modifiers, super, implements, fields, methods)$ *is a tuple, with*

$name \in \mathit{Class} \cup \mathit{Interface}$	*unique identifier of the class or interface,*
$isInterface \in \{true, false\}$	*determines if it defines an interface or a class,*
$modifiers \in \mathcal{P}(\mathit{Modifier})$	*the modifiers (static, private, public, ...),*
$super \in \mathit{Class}$	*the super class,*
$implements \in \mathcal{P}(\mathit{Interface})$	*the implemented interfaces,*
$fields : \mathit{Field} \rightarrow \mathcal{P}(\mathit{Modifier}) \times \mathit{Type}$	*with* $\mathit{Type} = \mathit{Class} \cup \mathit{Interface} \cup \{\mathit{boolean}\}$ *defining the fields, modifiers and types,s*
$methods : \mathit{MSig} \rightarrow \mathit{MDec}$	*the methods of the class.*

where MSig *is the set of method signatures* $\mathit{MSig} = \mathit{Meth} \times \mathit{Type}^\star$ *with set of method identifier* Meth *and a sequence determining the parameter types.* MDec *is the set of method declarations as defined in Definition 5.2.*

$\mathit{ClassFile}$ *denotes the set of all class files of a given Java Bytecode program.*

The sets Class and $\mathit{Interface}$ contain the identifiers of the classes and interfaces respectively, distinguished by $isInterface$: $\mathit{Class} = \{name_{cf} \mid cf \in \mathit{ClassFile} \wedge \neg isInterface_{cf}\}$ and $\mathit{Interface} = \{name_{cf} \mid cf \in \mathit{ClassFile} \wedge isInterface_{cf}\}$. We define the sets of available fields $\mathit{Class/Field} = \{name_{cf}.field \mid cf \in \mathit{ClassFile} \wedge field \in \mathit{Field}_{cf}\}$, which are uniquely identified by the combination of class and field name. We use $fieldType : \mathit{Class/Field} \rightarrow \mathit{Type}$, mapping fields to their type as defined by the mapping *fields* of the corresponding class file. We distinguish between *static* and *object fields*, where object fields are member variables accessible through the corresponding object, i.e. what we referred to by fields

so far, while static fields are *global variables* accessible in any context. In the following we refer to object fields if we use field and we use *static variable* for static fields. We denote the set of static variables by $\mathit{Class/Field}_S$, i.e. fields for which the modifier static is set, and denote the set of object fields by $\mathit{Class/Field}_O$. The set of methods is denoted by $\mathit{Class/MSig} = \{name_{cf}.mSig \mid cf \in \mathit{ClassFile} \land mSig \in \mathit{MSig}_{cf}\}$, where each method is uniquely defined by class name and method signature.

Definition 5.2 (Method declaration [SSB01]):
A method declaration $md = (modifiers, returnType, code, maxReg, maxOpd)$ *is a tuple consisting of*

$modifiers \in \mathcal{P}(\mathit{Modifier})$	*a set of modifiers,*
$returnType \in \mathit{Type} \cup \{\mathit{void}\}$	*the return type of the method,*
$code \in \mathit{Instruction}^\star$	*a sequence of instructions (Instruction is defined in Section 5.4),*
$maxOpd$	*the maximum size of the operand stack,*
$maxReg$	*the highest register used by the method.*

The set of all method declarations of a given Java program is denoted by MDec.

The function *method*: $\mathit{Class} \times \mathit{MSig} \to \mathit{Class/MSig}$ maps a class and a method signature to the corresponding implementation, i.e. $method(c, mSig)$ returns $c'.mSig \in \mathit{Class/MSig}$ where c' is the class in which the corresponding method is declared, i.e. returns the method of the given signature inherited from c. The function *method* is defined implicitly by *ClassFile*.

5.2.2 State of a JVM

The state of a JVM is determined by the state of the heap and method stack. The formal definitions, based on the ones given by Stärk *et. al.* [SSB01], are given below.

Definition 5.3 (Heap [SSB01]):
A heap is a function $heap : Ref \to Heap$*, where Ref is a set of references uniquely denoting objects and* $Heap = \mathit{Class} \times \mathit{Class/Field} \to \mathit{Val}$*, is a set of objects defined by the type and evaluation of references.*

In a general setting, *Val* is the set of possible values of a variable, including primitive data types. Here we define $\mathit{Val} = Ref \cup \{null\} \cup \{\mathsf{true}, \mathsf{false}\}$. This definition restricts

the variables to be pointers to objects, restricting variables of primitive datatypes to be a boolean. The above defined heap has a one-to-one correspondence with the heap configuration as defined in Section 4.1 on page 74, except that null pointers and boolean values are now permitted.

Definition 5.4 (Method stack [SSB01]):
A method stack is a sequence of frames, stack : Frame^* *where a Frame is a tuple* $(pc, reg, opd, method)$ *with:*

$pc \in \mathbb{N}$	*the program counter defining the current position in the method,*
$reg \in \mathbb{N} \to \mathsf{Val}$	*the values of the registers, used to store the local variable information,*
$opd \in \mathsf{Val}^*$	*the operand stack used to store intermediate results of calculations,*
$method \in \mathsf{Class}/\mathsf{MSig}$	*the corresponding method.*

The top frame of the stack defines the state of the active method.

Val is defined as above. As in Definition 5.2, the number of available (used) registers as well as the length of the operand stack is bounded for any method. In the next section we show how to model an entire state of a JVM, i.e. the heap as well as the method stack, by a heap configuration.

5.3 Modelling JVM States

Our aim is to model (abstract) states of the JVM by heap configurations. Starting with a basic model, representing only the heap and restricting programs to classes and fields, we will step by step extend this representation.

We consider programs in the most basic case where each class file defines a class and all fields are member variables, i.e. $\mathsf{Interface} = \emptyset$. There are no static variables yet. States represent heap configurations over the heap alphabet Σ with $\mathbb{T}_\Sigma = \mathsf{Class}$, $\mathbb{F}_\Sigma = \mathsf{Class}/\mathsf{Field}$ and $types_\Sigma(c.f) = c_\triangledown t_\blacktriangle$, where $t = fieldType(c.f)$.

5.3.1 Interfaces and *null*

Java distinguishes classes and interfaces. Interfaces cannot be instantiated, i.e. there are no objects of an interface type and *null* can be referenced but does not reflect a proper type.

We extend the heap alphabet Σ to $\mathbb{T} = \mathit{Class} \cup \mathit{Interface} \cup \{\bot\}$, where $\bot$ represents *null*. For all $t \in \mathbb{T}$ we let $\bot \preceq t$, i.e. $\bot$ is the least element in the type hierarchy. Other elements of $\mathbb{T}$ (including interfaces) are ordered as defined by *super* elements and the *implements* sets of the Java class files (see Definition 5.1).

We model *null* as an external ▲-vertex, i.e. a vertex that can be referenced but is not part of the heap. We consider heap configurations $(V, E, \mathrm{lab}, \mathrm{type}, \mathrm{att}, \mathbf{null}_{\blacktriangle})$ over Σ, where $\bot \in \mathbb{T}_\Sigma$ and $\{v \in V \mid \mathrm{type}(v) = \bot\} = \{\mathbf{null}\}$, i.e. the null reference is unique. This is important for comparisons of null pointers. As $\bot$ is the least element in the type hierarchy, any ▲-tentacle can be attached to **null**.

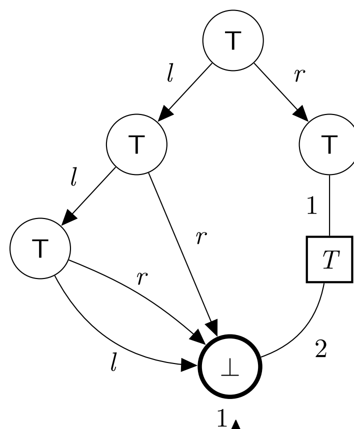


Figure 5.1: Heap configuration representing a tree

Example 5.1:
Consider the (abstract) heap configuration representing a binary tree in Figure 5.1. Every vertex is of the type Tree*:* ***class*** *Tree{Tree left , Tree right;} denoted by* Tree*. Leaves are* Tree *objects, where both pointers point to null. The single external vertex (ext =* $\mathbf{null}_{\blacktriangle}$*) is of type $\bot$ to realise pointers to null. As $\bot \preceq$* Tree *the l- and r-pointers can also point to a $\bot$-vertex. As* **null** *is a ▲-node, it has no outgoing edges. Note that otherwise it should have one outgoing edge for any defined field in $\mathbb{F}$, as $\bot$ is the least element of $(\mathbb{T}, \preceq)$. A grammar for the nonterminal T can be found in Figure 5.2.*

Vertices of type $\bot$ can also be used within grammar rules. However, vertices of type $\bot$ have to be external ▲-vertices. Otherwise for each field in $\mathbb{F}$ a corresponding outgoing edge must be attached. If the vertex would be internal we could derive additional vertices of type $\bot$ along **null**, which would violate the uniqueness of the *null*-reference. In Figure 5.2 a data structure grammar for arbitrary binary trees is given, where an external $\bot$-node is used to model *null*-references.

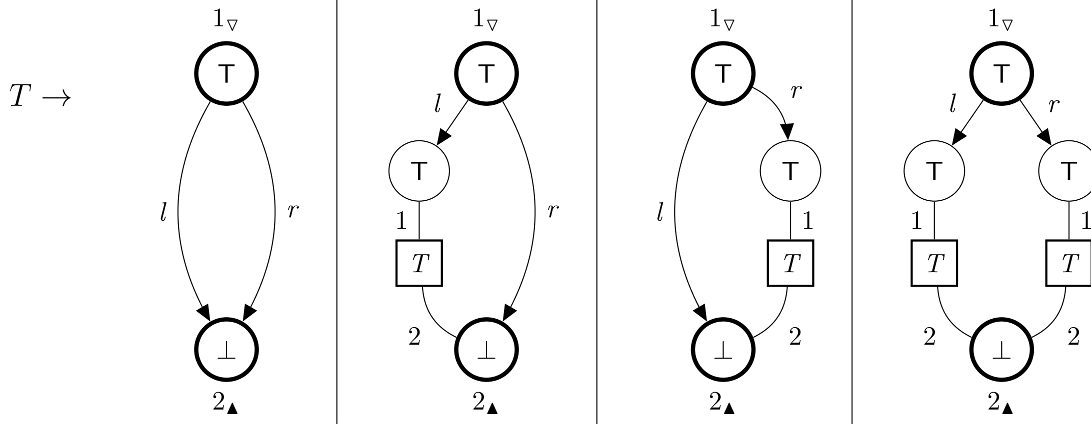


Figure 5.2: Data structure grammar for binary trees.

5.3.2 Static Variables, Literals and Boolean Values

Static variables are not linked to an object and are accessible from any context. We make them accessible through an external vertex $\boldsymbol{static}$ of (a new) type $\mathsf{static} \in \mathbb{T}$. For any static field $f \in \mathit{Class/Field}$ we define $\mathit{types}(c.f) = \mathsf{static}_\triangledown t_\blacktriangle$ ($t = \mathit{fieldType}(c.f)$), i.e. type static has one outgoing pointer for each static variable. We extend the sequence of external vertices by a vertex $\boldsymbol{static}$ and get $(V, E, \text{lab}, \text{type}, \text{att}, \boldsymbol{null}_\blacktriangle \boldsymbol{static}_\triangledown)$. We require $\boldsymbol{static}$ to be the unique static-node, i.e. $\{v \in V \mid \text{type}(v) = \mathsf{static}\} = \{\boldsymbol{static}\}$. The vertex $\boldsymbol{static}$ is an $\triangledown$-node of the heap.

Literals are a special case of static variables, whose values are implicitly used within a Java program. As we do not consider general data values, the only possible literals are the Boolean values false and true, represented in Java bytecode as integer values *zero* and *one*. In order to model Boolean values we add two vertices of (a new) type $\mathsf{int} \in \mathbb{T}$ representing integer value zero and one, accessible through $\boldsymbol{static}$ by edges labelled *int(0)* and *int(1)*, i.e. $\mathit{types}(\mathit{int}(0)) = \mathit{types}(\mathit{int}(1)) = \mathsf{static}_\triangledown \mathsf{int}_\blacktriangle$.

5.3.3 Complete Heap Representation

Given a set *ClassFile* of class files, we use heap configurations over the alphabet Σ with $\mathbb{T} = \mathit{Class} \cup \mathit{Interface} \cup \{\mathsf{static}, \mathsf{int}, \bot\}$, $\mathbb{F} = \mathit{Class/Field} \cup \{\mathit{int(0)}, \mathit{int(1)}\}$ and

$$\begin{aligned}\mathit{types} &= \{c.f \mapsto c_\triangledown t_\blacktriangle \mid c.f \in \mathit{Class/Field}_o \wedge \mathit{fieldType}(c.f) = t\} \\ &\cup \{c.f \mapsto \mathsf{static}_\triangledown t_\blacktriangle \mid c.f \in \mathit{Class/Field}_s \wedge \mathit{fieldType}(c.f) = t\} \\ &\cup \{\mathit{int}(0) \mapsto \mathsf{static}_\triangledown \mathsf{int}_\blacktriangle, \mathit{int}(1) \mapsto \mathsf{static}_\triangledown \mathsf{int}_\blacktriangle\}.\end{aligned}$$

We use a heap configuration of the form $H = (V, E, \text{lab}, \text{type}, \text{att}, \boldsymbol{static}_\triangledown \boldsymbol{null}_\blacktriangle)$, where none of the vertices is of interface type $\{v \in V \mid \text{type}(v) \in \mathit{Interface}\} = \emptyset$, $\boldsymbol{null}$ is the only vertex of type $\bot$ and $\boldsymbol{static}$ the only one of type static. The only two

int-nodes $\{v \in V \mid \text{type}(v) = \textsf{int}\} = \{v_{int(0)}, v_{int(1)}\}$ are successors of the vertex ***static***: $\exists e_0, e_1 \in \nabla(\textbf{\textit{static}}).\text{lab}(e_i) = int(i) \wedge \text{att}(e_i)[2] = v_{int(i)}$.

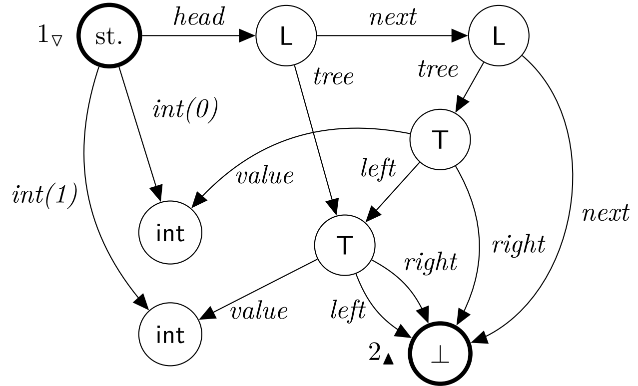
Example 5.2:

*Figure 5.3 presents (a) Java class definitions for list of trees and (b) a corresponding heap representation. The static variable List.head points to the first element of the list. Each tree node has a boolean value represented as pointer to one of the int-vertices for the values zero and one representing true respectively false. The static variable as well as the int-vertices are accessible through **static**.*

```
class Tree{
  Tree    left;
  Tree    right;
  boolean value;
}

class List{
  static List head;
  List next;
  Tree tree;
}
```

(a) Java class definition



(b) Heap representation

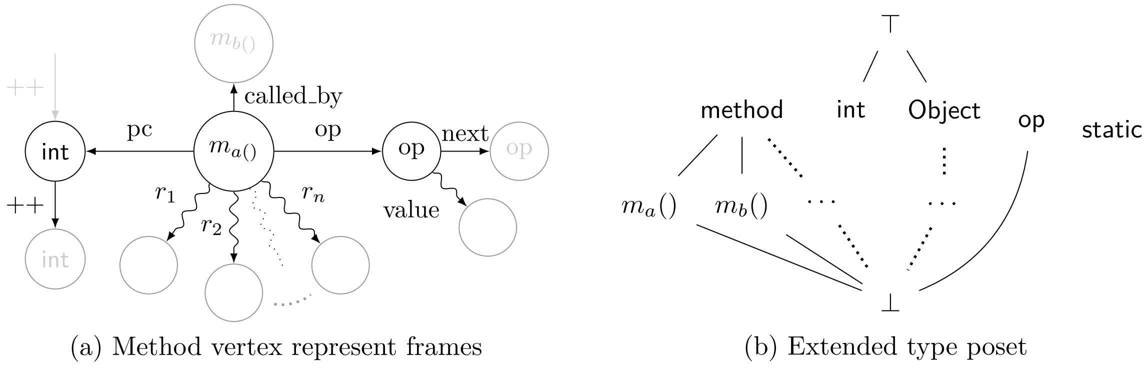
Figure 5.3: List of trees.

5.3.4 Method Stack

So far we only considered the heap component of a JVM state. Now we will model the method stack and its components within the same HC by representing each stack frame as a vertex of a special method type. This allows us to include the stack in the abstraction and therefore to handle recursive functions with unbounded method stack size. It is preferable to model both parts in one representation as abstracting heap and stack independently would imply losing their relation.

We model each frame $(pc, reg, opd, method)$ by a vertex. For each method $c.m \in$ *Class*/*MSig* we define a proper type c.m, reflecting the *method* component of the frame. Each method type is a subtype of a general method type method $\in \mathbb{T}$ (see Figure 5.4(b)).

For the program counter we add one int-node for each possible value, i.e. we add additional int-vertices $\{v_{int(i)} \mid 1 < i \leq max(\{|code(c.m)| \mid c.m \in \mathit{Class}/\mathit{MSig}\})\}$ to the configuration. In addition we add fields *int(i)* as pointers from static to int for each $0 \leq i \leq max(\{|code(c.m)| \mid c.m \in \mathit{Class}/\mathit{MSig}\})$. Further we add the field *++* with



(a) Method vertex represent frames

(b) Extended type poset

Figure 5.4: Method vertex represent frames of the method stack.

types$(++ = \mathsf{int}_\triangledown \mathsf{int}_\blacktriangle)$ representing the successor relation between int-nodes. The program counter is modelled as a pointer $method.pc \in \mathbb{F}$ to the corresponding int-node, i.e. $types(method.pc) = \mathsf{method}_\triangledown \mathsf{int}_\blacktriangle$. For the operand stack we add an op-type for stack elements with *next* and *value* successors, $types(op.next) = \mathsf{op}_\triangledown \mathsf{op}_\blacktriangle$ and $types(op.value) = \mathsf{op}_\triangledown \top_\blacktriangle$, where $\top \in \mathbb{T}$ with int and Object as subtypes (see Figure 5.4(b)), i.e. *op.value* can reference int- and Object-nodes. We add a pointer to each method-node $op \in \mathbb{F}$ to the operand stack ($types(op) = \mathsf{method}_\triangledown \mathsf{op}_\blacktriangle$).

As registers offer random access we model each register i by a pointer r_i. The number of registers depends on the method, therefore we define the r_i-pointer for each specification of the method-type. Given $c.m \in \mathit{Class/MSig}$ we define for each $1 \leq i \leq maxReg_{c.m}$: $types(r_i) = \mathsf{c.m}_\triangledown \top_\blacktriangle$ (see Figure 5.4(a)).

We model the method stack itself by an additional field $called_by \in \mathbb{F}$ with $types(called_by) = \mathsf{method}_\triangledown \mathsf{method}_\blacktriangle$ referencing the predecessor of the vertex within the stack, where the least element in the stack *called_by* points to ***null***.

The method at the top of the stack is the active method. The corresponding vertex contains the information that can be currently modified. Thus the top method vertex can access the heap and is modelled as an external $\triangledown$-node. We get heap configurations of the following form: $(V, E, \mathrm{lab}, \mathrm{type}, \mathrm{att}, \mathbf{method}_\triangledown \mathbf{static}_\triangledown \mathbf{null}_\blacktriangle)$. We call heap configurations of this form *JVM states*.

Given a JVM state $H \in \mathrm{HC}_\Sigma$ we use the shortcuts $\mathbf{method}_H = \mathrm{ext}_H[1]$, $\mathbf{static}_H = \mathrm{ext}_H[2]$, and $\mathbf{null}_H = \mathrm{ext}_H[3]$ for better readability.

Example 5.3:

Figure 5.5(a) presents the standard recursive tree traversal algorithm as Java program and (b) the corresponding Java bytecode for the trav-method (see Section 5.4 for details on Java bytecode).

*Figure 5.6 presents a state of this program. As the method trav(Tree t) consists of nine instructions (0 - 8) there are nine int-vertices. Each is accessible via a corresponding edge from **static**. In addition there are ++-edges between int-vertices*

```
public class Tree{
  Tree left, right;
  static void trav(Tree t){
    if(t != null){
      trav(t.left);
      trav(t.right);
    }
  }
}
```

(a) Java Class Definition

```
0 Load(Tree, 0)
1 Cond(ifNull, 8)
2 Load(Tree, 0)
3 GetField(Tree, Tree.left)
4 InvokeStatic(void, Tree.trav(Tree))
5 Load(Tree, 0)
6 GetField(Tree, Tree.right)
7 InvokeStatic(void, Tree.trav(Tree))
8 Return(void)
```

(b) Java Bytecode: trav(Tree t)

Figure 5.5: Recursive tree traversal.

that represent successive values. The m-labelled vertices are the method vertices, each representing a trav(Tree t)-method call. Three method calls are concrete, others are abstracted by the nonterminal edge X. The first external vertex represents the active method. The program counter is set to zero. The different methods were all called in instruction five: trav(t.left), as all up to the active one, have there program counters point set to $i(5)$. Note that in X method as well as tree vertices are abstracted.

5.3.5 Terminal States

If the execution of a JVM halts this is either because the executed program terminated properly or because some exception occurred. We do not consider exceptions in general but handle null-pointer and check-cast exceptions.

Proper Termination

The execution of a JVM is terminated properly, if the last frame is removed from the method stack. Note that within such a JVM state the first and third external vertex would be the same (***method*** = ***null***), which is against the definition of hypergraphs (Definition 2.1 on page 17). Instead we keep the last frame and set the pointer for the program counter to the vertex ***null***. Doing so also keeps the state of the local variables, which would otherwise be removed with the frame.

Exceptions

We do not model exception handling in general. However, there are two kinds of exceptions that could occur on runtime and yield an improper termination. These are *null-pointer exceptions* that occur if the JVM tries to dereference a pointer of a *null*-reference. And *check-cast exceptions* that occur if the JVM tries to set a reference to an object of an inaccurate type. We model the improper termination via special, not further specified JVM states that we name *NullPointer* and *CheckCastException*.

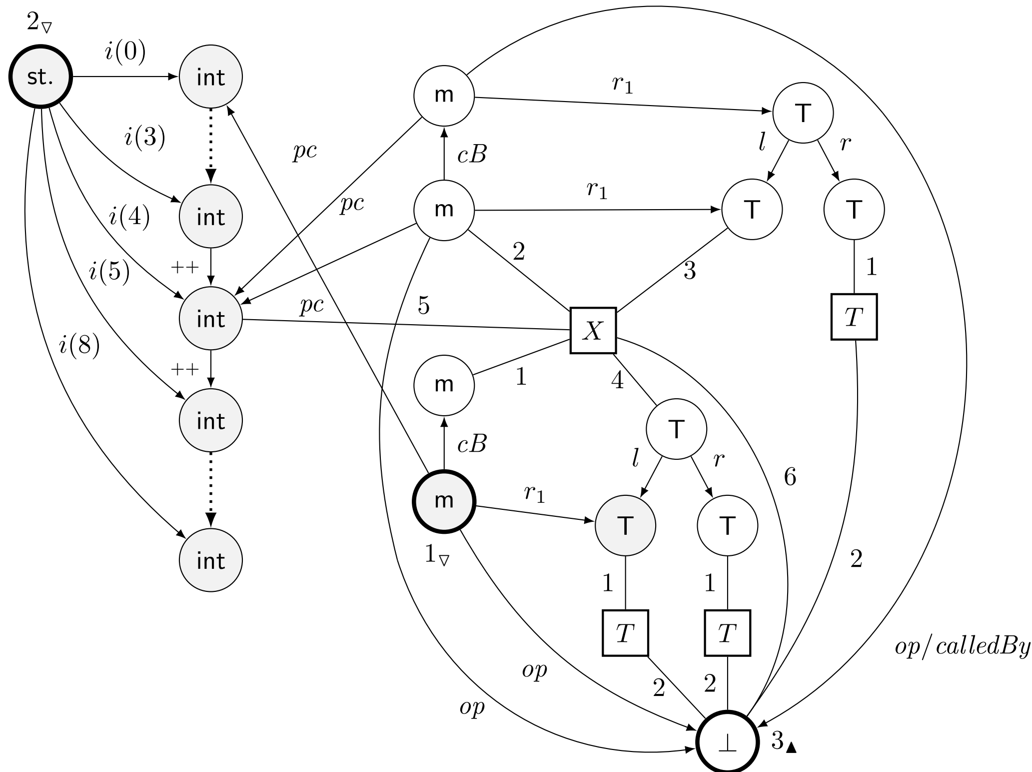


Figure 5.6: State of the abstract JVM.

5.4 Java Bytecode Execution

Our aim is to analyse Java bytecode programs. To do so we define in Section 5.4.2 a concrete transition relation for JVM states (see page 64). In Section 5.4.3, we then define an abstract semantics based on the result from Theorem 3.6 on page 68. Instead of the complex set of 205 opcodes we use the following set of *abstract instructions* defined by Stärk *et. al.* [SSB01], which covers the whole instruction set of a JVM.

Prim(PrimOp)	Dupx()	Pop()
Load(Type, RegNo)	Store(Type, RegNo)	Goto(LineNumber)
Cond(PrimOp, LineNumber)		
GetStatic(Type, Class/Field)	PutStatic(Type, Class/Field)	InvokeStatic(Type, Class/MSig)
Return(Type)		
New(Class)	Return(Type)	InstanceOf(Type)
GetField(Type, Class/Field)	PutField(Type, Class/Field)	Checkcast(Type)
InvokeSpecial(Type, Class/MSig)	InvokeVirtual(Type, Class/MSig)	

In Section 5.4.2 we describe the above abstract instructions and give their semantics in form of transition rules. A translation table for opcodes and abstract instructions can be found in Appendix C.8 of [SSB01]. The Type information of the instructions can

be used to check for type safeness but is ignored by the JVM. We thus omit the type information in abstract instructions. Focusing on heap structures we do not consider primitive data values. This results in a notable cutback of primary operations (PrimOp). The complete set of primary operations can be found in Stärk *et. al.* [SSB01]. We consider:

if_acmpeq	if_acmpne	if_icmpeq	if_icmpne
iconst_0	iconst_1	iand	ior

The primary if-operations, used by the Cond instruction, are realised by comparing the corresponding vertices referred by the stack. The operations iconst_0 and iconst_1 push the corresponding int-vertices on the stack. The operations iand and ior can be defined explicitly for the four possible inputs.

In Section 5.4.2 we describe the instructions and give their transition rules. In the following section we define some basic graph modifiers that we will use for the definition of the transition rules.

5.4.1 Basic Modifier

To simplify the notation of the concrete semantics (Section 5.4.2), we first define the impact of some commonly used operations (such as *push*, *pop*, etc.). These *basic modifiers* are then used to define the semantics of the JVM statements in a more comfortable way.

new$(H, t) = (H', v_{new})$

The basic modifier *new* generates a new object of type $t \in \mathbb{T}$. Given $H \in \mathrm{HC}_\Sigma$ the new vertex v_{new} of type t is added to the heap configuration H:

$$H' = (V_H \uplus \{v_{new}\}, E_H, lab_H, \mathrm{type}_H \cup \{v_{new} \mapsto t\}, \mathrm{att}_H, ext_H).$$

suc$(H, v, f) = v'$

The modifier *suc* is used for pointer dereferencing. For the field $f \in \mathbb{F}$ it returns the corresponding successor of the vertex $v \in V_H$:

$$\{v'\} = \mathrm{att}_H(\{e \in \nabla_H(v) \mid \mathrm{lab}_H(e) = f\})[2].$$

setSuc$(H, v, f, v') = H'$

The modifier *setSuc* is used to set a pointer in $H \in \mathrm{HC}_\Sigma$ to a new target. This is done by changing the attachment of the corresponding edge within H:

$$H' = (V_H, E_H, \mathrm{lab}_H, \mathrm{type}_h, \mathrm{att}_h[e \mapsto v\ v'], \mathrm{ext}_h),$$

where e is the single element of $\{e \in \nabla_H(v) \mid \mathrm{lab}(e) = f\} = \{e\}$ A schematic representation of the modification is given in Figure 5.7.

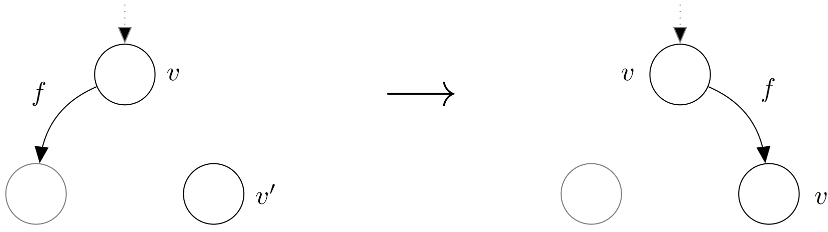


Figure 5.7: The modifier *setSuc* (H, v, f, v')

push$(H, v) = H'$

The modifier *push* pushes a reference onto the operand stack. A schematic representation of the *push*-modifier can be found in Figure 5.8.

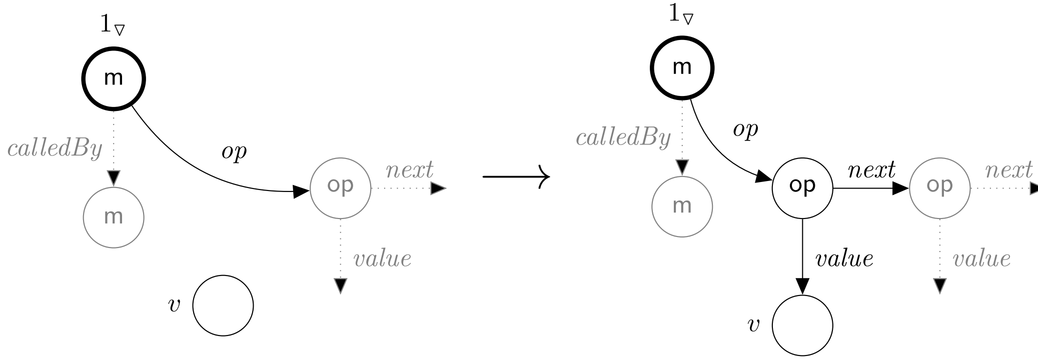


Figure 5.8: The modifier $push(H, v)$

Given heap configuration $H \in \mathrm{HC}_\Sigma$ and vertex $v \in V_H$ we proceed as follows. First we generate a new operand stack element using the new modifier and we get $(H_{new}, v_{op}) = new(H, \mathsf{op})$. Then we set the successors of the new vertex v_{op}. The *value*-successor is set to the referenced vertex v using the *setSuc* modifier. Within the resulting heap configuration we then set the *next*-successor to the top vertex of the operand stack $v_{top} = suc(H, \texttt{method}_H, op)$ and finally we update the reference to the operand stack.

$$H' = setSuc(\ setSuc(\ setSuc(H_{new}, v_{op}, value, v), v_{op}, next, v_{top}), ext_H[1], op, v_{op}).$$

pop$(H) = (H', v)$

The modifier *pop* removes the top-element of the operand stack and returns its reference. Given a heap configuration $H \in \mathrm{HC}_\Sigma$ the vertex representing the top element of the operand stack is $v_{top} = suc(H, \texttt{method}_H, op)$. The corresponding referenced vertex is $v = suc(H, v_{top}, value)$. We remove the top element by by altering the *op*-edge to the next stack element.

$$H' = setSuc(H, \texttt{method}_H, op, \ suc(H, v_{top}, next)).$$

$\boldsymbol{peek}(H, i) = v$

The modifier *peek* returns an element from the operand stack without removing it. Given $H \in \mathrm{HC}_\Sigma$ and $i \in \mathbb{N}$, the ith element of the operand stack is returned. We define suc^n recursively as $suc^n(H, v, f) = suc^{n-1}(H, suc(H, v, f), f)$ for $n > 0$ and $suc^0(H, v, f) = v$, i.e. suc^n returns the nth f-successor of a vertex.

$$v = suc(H, suc^i(H, suc(H, \mathtt{method}_H, op), next), value)$$

$\boldsymbol{incPc}(H) = H'$

The modifier *incPc* increments the program counter. The modification is schematic depicted in Figure 5.9. The current value of the program counter is represented by the vertex $v_{pc} = suc(H, \mathtt{method}_H, pc)$. We increment the program counter by altering it to the successor vertex of v_{pc} and we get the heap configuration

$$H' = setSuc(H, \mathtt{method}_H, pc, suc(H, v_{pc}, ++)).$$

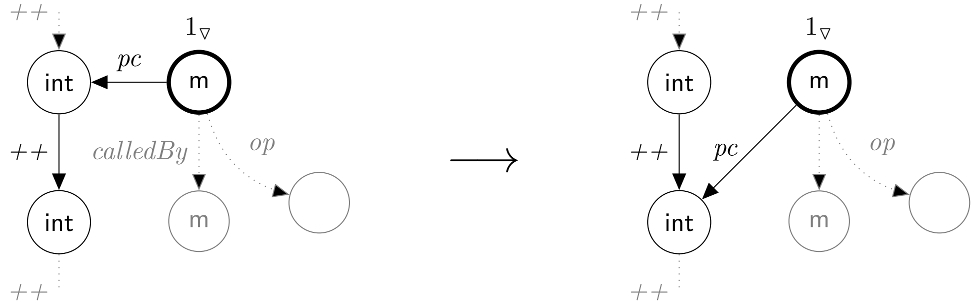


Figure 5.9: The modifier *incPc*(H)

$\boldsymbol{inst}(H) = op$

The basic instruction *inst* returns the current abstract instruction. Referring to the static environment *cEnv* of the program, the instruction is determined by the active method and the current program counter. Given a JVM state $H \in \mathrm{HC}_\Sigma$ the active method can be determined by the type of the vertex $\mathtt{method}_H$ and the program counter by its *pc*-successor. Given a heap configuration $H \in \mathrm{HC}_\Sigma$ We determine the integer values that corresponds to a int-vertex by the function $intValue_H : V_H \to \mathbb{N}$ with $intValue(v) = i$, where $suc(H, \mathtt{static}_H, int(i)) = v$.

$$op = \boldsymbol{code}(\mathrm{type}_H(\mathtt{method}_H))[\, intValue_H(\, suc(H, \mathtt{method}_H, pc))].$$

If H is not a proper JVM State (e.g. a terminal state or the state *NullPointer*), then $inst(H) = \bot$.

5.4.2 Concrete Semantics

In this section we give for the abstract instructions the (concrete) semantics in form of transition rules for the JVM. The transition rules describes the transition relation between concrete states of the JVM. Each rule is of the form:

$$\frac{\mathit{inst}(H) = \mathit{op} \qquad \mathit{property}(H, \ldots)}{H \to H'},$$

stating that there is a transition from $H \in \mathrm{HC}_{T_\Sigma}$ to $H' \in \mathrm{HC}_{T_\Sigma}$ if the active instruction of H is op and H, potential with additional parameter determining H', fulfil the property *property*. Some transition rules contain more than one property, that the all must be fulfilled. All concrete transition rules are deterministic.

Prim(PrimOp)

The instruction Prim(PrimOp) executes the primary operation PrimOp. All primary operations have a boolean result which is pushed onto the operand stack. Given $H \in \mathrm{HC}_\Sigma$ the pushes are realised via:

$$\mathit{pushTrue}(H) = \ \mathit{incPc}(\ \mathit{push}(\ H,\ \mathit{suc}(H, \mathtt{static}_H, \mathit{int(1)})\,)\,)$$

$$\mathit{pushFalse}(H) = \ \mathit{incPc}(\ \mathit{push}(\ H,\ \mathit{suc}(H, \mathtt{static}_H, \mathit{int(0)})\,)\,)$$

Prim(if_acmpeq) / Prim(if_icmpeq)

Checks if the two topmost entries of the operand stack are equal. Java distinguishes between references to objects and primitive data values. We represent both by vertices. Therefore the rules for Prim(if_icmpeq) are the same as the ones for Prim(if_acmpeq), which are given below:

$$\frac{\mathit{inst}(H) = \mathsf{PrimOp(if_acmpeq)} \qquad \mathit{pop}(H) = (H', v_1) \qquad \mathit{pop}(H') = (H'', v_2) \qquad v_1 = v_2}{H \to \mathit{pushTrue}(H'')}$$

$$\frac{\mathit{inst}(H) = \mathsf{PrimOp(if_acmpeq)} \qquad \mathit{pop}(H) = (H', v_1) \qquad \mathit{pop}(H') = (H'', v_2) \qquad v_1 \neq v_2}{H \to \mathit{pushFalse}(H'')}$$

Prim(if_acmpneq) / Prim(if_icmpneq)

These instructions are the counterpart to the previous one. They check if two references are *not* equal.

$$\frac{\mathit{inst}(H) = \mathsf{PrimOp(if_acmpneq)} \qquad \mathit{pop}(H) = (H', v_1) \qquad \mathit{pop}(H') = (H'', v_2) \qquad v_1 = v_2}{H \to \mathit{pushFalse}(H'')}$$

$$\frac{\mathit{inst}(H) = \mathsf{PrimOp(if_acmpneq)} \qquad \mathit{pop}(H) = (H', v_1) \qquad \mathit{pop}(H') = (H'', v_2) \qquad v_1 \neq v_2}{H \to \mathit{pushTrue}(H'')}$$

Prim(iconst_0) / Prim(iconst_1)

The instructions Prim(iconst_0) and Prim(iconst_1) push a reference to the integer value *zero* respectively *one* to the operand stack.

$$\frac{inst(H) = \mathsf{PrimOp(iconst_0)}}{H \rightarrow pushFalse(H)}$$

$$\frac{inst(H) = \mathsf{PrimOp(iconst_1)}}{H \rightarrow pushTrue(H'')}$$

Prim(iand) / Prim(ior)

The instructions Prim(iand) and Prim(ior) perform a logical *and* respectively *or* on the boolean values, that are represented by the two integer vertices referred by the topmost operand stack entries. Given a heap configuration $H \in \mathrm{HC}_\Sigma$ and two vertices $v_1, v_2 \in V_H$ the property $and_H(v_1, v_2)$ is fulfilled if $intValue_H(v_1) = 1$ and $intValue_H(v_2) = 1$, else it is not fulfilled. Analogous property $or_H(v_1, v_2)$ is fulfilled if and only if $intValue_H(v_1) = 1$ or $intValue_H(v_2) = 1$. We get for Prim(iand) the transition rules

$$\frac{inst(H) = \mathsf{PrimOp(iand)} \qquad pop(H) = (H', v_1) \qquad pop(H') = (H'', v_2) \qquad and_H(v_1, v_2)}{H \rightarrow pushTrue(H'')}$$

$$\frac{inst(H) = \mathsf{PrimOp(iand)} \qquad pop(H) = (H', v_1) \qquad pop(H') = (H'', v_2) \qquad \neg and_H(v_1, v_2)}{H \rightarrow pushFalse(H'')}$$

Analogous we get for Prim(ior) the transition rules

$$\frac{inst(H) = \mathsf{PrimOp(ior)} \qquad pop(H) = (H', v_1) \qquad pop(H') = (H'', v_2) \qquad or_H(v_1, v_2)}{H \rightarrow pushTrue(H'')}$$

$$\frac{inst(H) = \mathsf{PrimOp(ior)} \qquad pop(H) = (H', v_1) \qquad pop(H') = (H'', v_2) \qquad \neg or_H(v_1, v_2)}{H \rightarrow pushFalse(H'')}$$

Dupx()

The instruction Dupx() duplicates the topmost element of the operand stack. This instruction is used for constructor calls and shortcut assignments (as a += b).

In the abstract JVM we realise Dupx by peeking the reference of the topmost element of the operand stack and then pushing this element once more to the stack:

$$\frac{inst(H) = \mathsf{Dupx()}}{H \rightarrow\ incPc(\ push(\ H,\ peek(H, 1)\ )\ )}$$

Pop()

The abstract instruction Pop() has a one-to-one correspondence to the basic instruction *pop*, that removes the top most element from the operand stack. However, in the abstract instruction the removed element is not further considered.

For the abstract JVM we execute a *pop* instruction and we get a transition to the resulting heap for which we have to increment the program counter.

$$\frac{\mathit{inst}(H) = \mathsf{Pop}() \qquad \mathit{pop}(H) = (H', v)}{H \rightarrow \mathit{incPc}(H')}$$

Load(RegNo)

The Load(RegNo) instruction reads a reference from the Register RegNo and pushes it to the operand stack for further computations.

We determine the vertex corresponding to the value of the register and push it to the stack. Within the heap configuration the active method is represented by the first external vertex. The value of the register i of the active method is the r_i-successor of this vertex.

$$\frac{\mathit{inst}(H) = \mathsf{Load(i)}}{H \rightarrow \mathit{incPc}(\ \mathit{push}(H,\ \mathit{suc}(H, \mathtt{method}_{H'}, r_i))\ )}$$

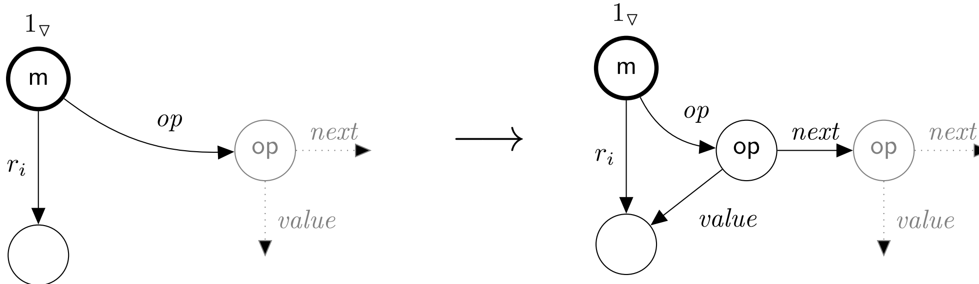


Figure 5.10: A transition induced by Load(i)

Store(RegNo)

The Store(RegNo) instruction is the counterpart of the Load instruction. It pops the topmost reference from the operand stackand stores it into the register RegNo.

To realise the execution of the Store instructions we pop the vertex on the top of the operand stack and update the register entry with its value.

$$\frac{\mathit{inst}(H) = \mathsf{Store(i)} \qquad \mathit{pop}(H) = (H', v)}{H \rightarrow \mathit{incPc}(\ \mathit{setSuc}(H', \mathtt{method}_{H'}, r_i, v)\ )}$$

Goto(LineNumber)

The Goto(LineNumber) instruction represents an unconditional jump. The only impact of the instruction is a change of the program counter for the active stack frame.

In the JVM we realise the change of the program counter by altering the attachment information of the *pc*-edge connecting the vertex representing the active method with a vertex representing the value of the program counter. The vertex representing the new value i of the program counter is the $int(i)$-successor of the vertex $\mathtt{static}_H$.

$$\frac{inst(H) = \mathsf{Goto(i)}}{H \rightarrow \; setSuc(\, H, \mathtt{method}_H, pc, \; suc(H, \mathtt{static}_H, int(i))\,)}$$

Cond(PrimOp, LineNumber)

The instruction Cond(PrimOp, LineNumber) represents a conditional jump. The primary operation PrimOp determines the condition. The available primary operations are if_acmpeq and if_acmpne that compare two references, where the former is true iff the references are equal, while the latter is true iff the references are not equal, and if_icmpeq and if_icmpne for the comparision of integer values. Any of the four checks removes the compared values from the stack. More complex conditions including conditions combined by *and* and *or* are transformed into a sequence of instructions by the Java compiler. Typically such a sequence consists of several primary operations, building up the complex condition, concluded by a push of the value *one* representing the boolean value true, and a check for equality if_icmpeq.

In an abstract state integer values as well as objects are represented by vertices. Therefore we can handle them equally and we use if_eq and if_ne as placeholders for the corresponding primary operations. Depending on the condition and if the two topmost stack entries are equal or not we get one of the following four rules.

If the condition is if_acmpeq or if_icmpeq and the entries are equal we get:

$$\frac{inst(H) = \mathsf{Cond(if_eq, i)} \qquad pop(H) = (H', v') \qquad pop(H') = (H'', v'') \qquad v' = v''}{H \rightarrow \; setSuc(\, H'', \mathtt{method}_{H''}, pc, \; suc(H'', \mathtt{static}_{H''}, int(i))\,)}$$

otherwise if the entries are different we get:

$$\frac{inst(H) = \mathsf{Cond(if_eq, i)} \qquad pop(H) = (H', v') \qquad pop(H') = (H'', v'') \qquad v' \neq v''}{H \rightarrow \; incPc(H'')}$$

If the condition if_acmpne or if_icmpne and the both top elements of the stack are equal:

$$\frac{inst(H) = \mathsf{Cond(if_ne, i)} \qquad pop(H) = (H', v') \qquad pop(H') = (H'', v'') \qquad v' = v''}{H \rightarrow \; incPc(H'')}$$

and if they are not equal:

$$\frac{inst(H) = \mathsf{Cond(if_ne, i)} \qquad pop(H) = (H', v') \qquad pop(H') = (H'', v'') \qquad v' \neq v''}{H \rightarrow \; setSuc(\, H'', \mathtt{method}_{H''}, pc, \; suc(H'', \mathtt{static}_{H''}, int(i))\,)}$$

GetStatic(Class/Field)

The instruction GetStatic(Class/Field) reads the value of a static variable and pushes the value onto the top of the operand stack.

In the abstract JVM the vertices representing the values of static variables are connected with the vertex $\boldsymbol{static}_H$ through an edge labelled with the corresponding variable name. The resulting vertex is pushed to the operand stack.

$$\frac{\mathit{inst}(H) = \mathsf{GetStatic(c.f)}}{H \rightarrow\ \mathit{incPc}(\ \mathit{push}(H,\ \mathit{suc}(H, \boldsymbol{static}_H, c.f))}$$

PutStatic(Class/Field)

The abstract instruction PutStatic(Class/Field) writes the topmost entry of the operand stack to the given static variable Class/Field. By writing it to the variable the stack entry is removed.

We update the value of the static variable by setting the corresponding successor of the vertex $\boldsymbol{static}_H$ to the vertex representing the value on the top of the stack. By doing so we remove the top element from the stack.

$$\frac{\mathit{inst}(H) = \mathsf{PutStatic(c.f)} \qquad \mathit{pop}(H) = (H', v)}{H \rightarrow\ \mathit{incPc}(\ \mathit{setSuc}(H', \boldsymbol{static}_{H'}, c.f, v)\,)}$$

InvokeStatic(Class/MSig)

Static methods are called via the instruction InvokeStatic(Class/MSig). A new method stack frame is created with an empty operand stack, the program counter set to the first instruction and the registers corresponding to the parameters of the method are initialised with references and values from the operand stack. The InvokeStatic instruction is depicted in Figure 5.11.

Figure 5.11: A transition induced by Invoke(c.m(A, B))

Given an JVM state $H \in \mathrm{HC}_\Sigma$ we add a new vertex of the corresponding method type and initialise its successors. The *calledBy* successor is set to the active method, represented by $\boldsymbol{\mathtt{method}}_H$. The program counter is initialised to zero by adding an edge to the $int(0)$-successor of the vertex $\boldsymbol{\mathtt{static}}_H$. The initial operand stack is empty, thus the op-successor of the new method vertex is $\boldsymbol{\mathtt{null}}_H$. The register edges r_1 to r_n are set to the n-topmost entries of the operand stack, which are removed from the stack.

$$\frac{inst(H) = \mathsf{InvokeStatic(c.msig)}}{H \to call(H, method_{cEnv}(c.msig))},$$

where $call(\,H, c.m(p)) = \;setSuc(K, \boldsymbol{\mathtt{method}}_H, op,\; peek(H, |p|)\,)$ with:

$$\begin{array}{lcl}
V_K & = & V_H \uplus \{v_m\} \\
E_K & = & E_H \uplus \{e_{calledBy}, e_{op}, e_{pc}\} \uplus \{e_{r_i} \mid 1 \leq i \leq \mathrm{maxReg}_{c.m(p)}]\} \\
\mathrm{lab}_K & = & \mathrm{lab}_H \uplus \{e_x \mapsto x \mid e_x \in E_K \setminus E_H\} \\
\mathrm{type}_K & = & \mathrm{type}_H \cup \{v_m \mapsto c.m(p)\} \\
\mathrm{att}_K & = & \mathrm{att}_H \\
 & \cup & \{e_{calledBy} \mapsto v_m\; \boldsymbol{\mathtt{method}}_H, \quad e_{op} \mapsto v_m\; \boldsymbol{\mathtt{null}}_H, \} \\
 & \cup & \{e_{pc} \mapsto v_m \quad suc(H, \boldsymbol{\mathtt{static}}_H, int(0))\} \\
 & \cup & \{e_{r_i} \mapsto v_m \quad peek(i) \mid 1 \leq i \leq |p|\} \\
 & \cup & \{e_{r_i} \mapsto v_m\; \boldsymbol{\mathtt{null}}_H \mid |p| \leq i \leq maxReg_{c.m(p)}\} \\
\mathrm{ext}_K & = & v_{m\triangledown}\; \boldsymbol{\mathtt{static}}_{H\triangledown}\; \boldsymbol{\mathtt{null}}_{H\blacktriangle}
\end{array}$$

Return()

The instruction Return() terminates the active method and returns to the invoking method, which execution is continued after the instruction that invoke the terminated method. If the corresponding Java method has a return value, then the return value is the topmost element of the operand stack. Therefore the top element of the stack is removed and pushed to the stack of the invoking method, such that it could be used for further computations. Additional elements on the operand stack are discarded. In Figure 5.12 the result of a Return() with return type A is given.

The realisation depends on whether the terminating method has a return value. Given a JVM state $H \in \mathrm{HC}_\Sigma$ the return type of the active method is $rT(H) = returnType(\,\mathrm{type}_H(\boldsymbol{\mathtt{method}}_H)\,)$. The type is void if the method does not return a result.

In any case we replace the first external vertex by its *calledBy*-successor to remove the terminated method from the stack. Given a heap configuration $H \in \mathrm{HC}_\Sigma$ the removal of the topmost method from the method stack, is realised by the function

$$removeTop(H) = (V_H, E_H, \mathrm{lab}_H, \mathrm{type}_H, \mathrm{att}_H, \boldsymbol{\mathtt{method}}^2_H\; \boldsymbol{\mathtt{static}}_H\; \boldsymbol{\mathtt{null}}_H),$$

where $\boldsymbol{\mathtt{method}}^2_H = \; suc(H, \boldsymbol{\mathtt{method}}_H, calledBy)$ is the second vertex on the method stack. If the terminated method has no return value we use the transition rule

$$\frac{inst(H) = \mathsf{Return()} \qquad \boldsymbol{\mathtt{method}}^2_H \neq \boldsymbol{\mathtt{null}}_H \qquad rT(H) = \mathsf{void}}{H \to \;incPc(\,removeTop(K)\,)}$$

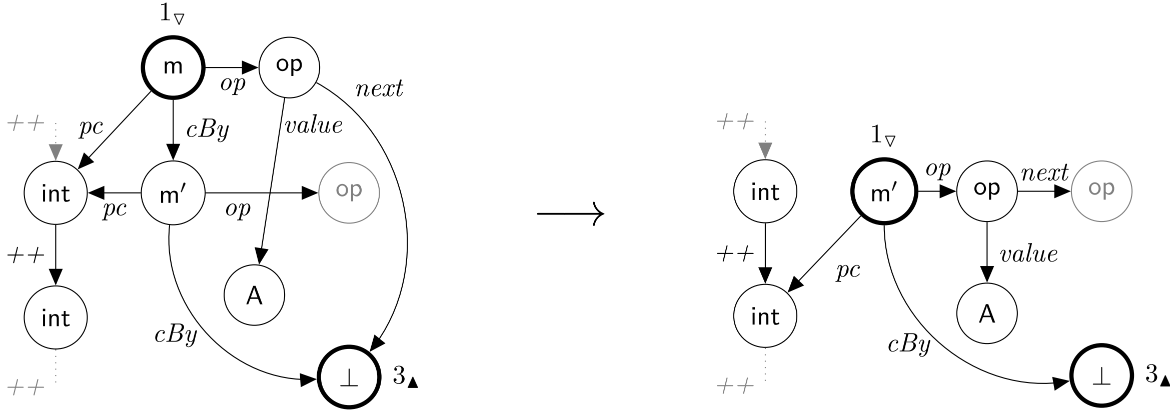


Figure 5.12: A transition induced by Return()

If the method has a return value we pop the topmost element from the operand stack before we remove the topmost method frame, and push it to the operand stack of the new topmost method-vertex afterwards.

$$\frac{inst(H) = \mathsf{Return()} \qquad \boldsymbol{\mathtt{method}}^2_H \neq \boldsymbol{\mathtt{null}}_H \qquad rT(H) \neq \mathsf{void} \qquad pop(H) = (H', v)}{H \rightarrow \; incPc(\; push(\, removeTop(K), \; v \,) \,)}$$

Note that in the above transition rules we need the *calledBy* successor of the active method to be different from $\boldsymbol{\mathtt{null}}_H$, otherwise the Return() terminates the execution and we move to a terminal state as defined in Section 5.3.5. That is the pointer for the program counter is set to the vertex $\boldsymbol{\mathtt{null}}_H$. We do not distinguish methods with and without return value as in any case there is no calling method to which we could return a value.

$$\frac{inst(H) = \mathsf{Return()} \qquad \boldsymbol{\mathtt{method}}^2_H = \boldsymbol{\mathtt{null}}_H}{H \rightarrow \; setSuc(\, H, \boldsymbol{\mathtt{method}}_H, pc, \boldsymbol{\mathtt{null}}_H \,)}$$

New(Class)

The abstraction instruction New(Class) generates a new object of the type Class. A reference to the newly generated object is pushed to the operand stack. The newly generated object is not initialised. For the initialisation a constructor has to be called explicitly. If the bytecode results from the compilation of a Java program then the call of a constructor is ensured.

In the abstract JVM we add a new vertex of the type, that represents the class Class, to the corresponding heap configuration and push it to the operand stack.

$$\frac{inst(H) = \mathsf{new(t)}}{H \rightarrow \; incPc(\; push(H, \; new(t)) \,)}$$

InstanceOf(Type)

The InstanceOf(Type) instruction checks if the top most element of the stack (that should be a reference) is referencing to an object of the type (or a subtype of) Type. If this is the case the integer value *one*, representing the boolean value true, is pushed to the operand stack, the integer value zero otherwise. The instruction removes the checked reference from the operand stack.

Within the abstract JVM we pop the topmost element of the operand stack and check if the type of the referred vertex is equal or a subtype of the type corresponding to Type. If this is the case we use the basic push-instruction to add a reference to the integer vertex representing the value *one*, which we get as the $int(1)$ successor of the static vertex. Analogous if it is not a subtype we push a reference to the integer vertex representing the value *zero*.

For the case that the topmost stack entry is of a subtype we get:

$$\frac{inst(H) = \mathsf{InstanceOf(t)} \qquad pop(H) = (H', v) \qquad \mathrm{type}_H(v) \preceq t}{H \rightarrow \ incPc(\ push(H',\ suc(H', \boldsymbol{static}_H, \mathsf{true}))\,)}$$

Otherwise if the stack entry is not a subtype:

$$\frac{inst(H) = \mathsf{InstanceOf(t)} \qquad pop(H) = (H', v) \qquad \mathrm{type}_H(v) \not\preceq t}{H \rightarrow \ incPc(\ push(H',\ suc(H', \boldsymbol{static}_H, \mathsf{false}))\,)}$$

GetField(Class/Field)

The abstract instruction GetField(Class/Field) reads a value from the field given by Class/Field of the object that is referred by the topmost element of the operand stack. The data value or reference is pushed to the operand stack.

Given a JVM state we model the instruction in the following steps. First we pop the topmost element from the operand stack. Next we get the vertex representing the value of the field by reading the corresponding successor of the vertex. The successor vertex then is pushed to the operand stack.

$$\frac{inst(H) = \mathsf{GetField(c.f)} \qquad pop(H) = (H', v) \qquad v \neq \boldsymbol{null}_{H'}}{H \rightarrow \ incPc(\ push(H',\ suc(H', v, c.f))\,)}$$

Note that we need the topmost entry of the operand stack to be different from *null*, as the execution would result in a Null-Pointer-Exception otherwise. That is if the topmost entry is *null* we add a transition into the *NullPointer*-state.

$$\frac{inst(H) = \mathsf{GetField(c.f)} \qquad pop(H) = (H', v) \qquad v = \boldsymbol{null}_{H'}}{H \rightarrow NullPointer}$$

PutField(Class/Field)

The instruction PutField(Class/Field) writes a value to a field of an object. The value that should be written to the field is the topmost element of the operand stack, while the succeeding element on the stack is the object for which the field should be updated.

Both references are removed from the stack during the execution of the instruction. The operation is destructive as the previous value of the field is overwritten.

Given a heap configuration representing an abstract state of the JVM we determine the vertex representing the value as well as the vertex representing the object for which the field should be updated by two succeeding *pop* operations. The first *pop* operation gives us the target vertex. The second *pop* operation gives us the vertex for which we update the corresponding successor to point to the target vertex.

$$\frac{\mathit{inst}(H) = \mathsf{PutField(f)} \qquad \mathit{pop}(H) = (H', v) \qquad \mathit{pop}(H') = (H'', v') \qquad v' \neq \mathtt{null}_{H''}}{H \rightarrow \mathit{incPc}(\mathit{setSuc}(H'', v', f, v))}$$

As for GetField we need the object reference to be not *null*, as otherwise the execution results in a Null-Pointer-Exception.

$$\frac{\mathit{inst}(H) = \mathsf{PutField(f)} \qquad \mathit{pop}(H) = (H', v) \qquad \mathit{pop}(H') = (H'', v') \qquad v' = \mathtt{null}_{H''}}{H \rightarrow \mathit{NullPointer}}$$

Checkcast(Type)

The Checkcast(Type) instruction is similar to the Instanceof(Type) instruction. It is used to check if a cast could be done, thus to check if an object is of the type or of a subtype of the type Type, which is the target type. If the cast is possible the execution succeeds without any further manipulations. Otherwise if Checkcast(Type) fails an ClassCastException is thrown.

The realisation is straight forward. If the Checkcast is successful we succeed the execution. Otherwise we move to the state *CheckCastException*.

$$\frac{\mathit{inst}(H) = \mathsf{Checkcast(t)} \qquad \mathit{pop}(H) = (H', v) \qquad \mathrm{type}_H(v) \preceq t}{H \rightarrow \mathit{incPc} H'}$$

$$\frac{\mathit{inst}(H) = \mathsf{Checkcast(t)} \qquad \mathit{pop}(H) = (H', v) \qquad \mathrm{type}_H(v) \not\preceq t}{H \rightarrow \mathit{CheckCastException}}$$

InvokeSpecial(Class/MSig)

In Java bytecode we distinguish two types of member method calls. InvokeSpecial are method calls where the called method can be determined statically. This kind of invokes are for example used for constructor calls. On the other hand there are InvokeVirtual method calls, where the method must be determined dynamically for each execution of the statement.

To model an InvokeSpecial instruction we add an additional vertex to the method stack, which is of the type of the corresponding calling method. We get the method using the method signature and defining class given as parameter, from the static environment by $\mathit{method}_{cEnv}(c.msig)$. We initialise the successors of the new method vertex. The *calledBy*-successor is set to vertex $\mathtt{method}_H$, the *pc*-successor is set to the integer vertex

representing the value *zero*. The operand stack is empty at the beginning of the execution of the method, thus the *op*-successor is set to $\boldsymbol{null}_H$. The register successors are set to $\boldsymbol{null}_H$, up to the ones representing parameters of the method, which are initialised by the topmost values from the operand stack of the invoking method (which are removed).

$$\frac{inst(H) = \mathsf{InvokeSpecial(c.msig)}}{H \to call(H', method_{cEnv}(c.msig))}$$

where $call(H, c.m(p)) = \; setSuc(K, op, \; peek(H, |p| + 1))$ with:

$$\begin{array}{lcl}
V_K & = & V_H \uplus \{v_m\} \\
E_K & = & E_H \uplus \{e_{calledBy}, e_{op}, e_{pc}\} \uplus \{e_{r_i} \mid 1 \le i \le maxReg_{c.m(p)}\} \\
\mathrm{lab}_K & = & \mathrm{lab}_H \uplus \{e_x \mapsto x \mid e_x \in E_K \setminus E_H\} \\
\mathrm{type}_K & = & \mathrm{type}_H \cup \{v_m \mapsto c.m(p)\} \\
\mathrm{att}_K & = & \mathrm{att}_K \\
 & \cup & \{e_{calledBy} \mapsto \boldsymbol{method}_H, e_{op} \mapsto \boldsymbol{null}_H, e_{pc} \mapsto \; suc(H, \boldsymbol{static}_H, int(0))\} \\
 & \cup & \{e_{r_i} \mapsto \; peek(i) \mid 1 \le i \le |p| + 1\} \\
 & \cup & \{e_{r_i} \mapsto \boldsymbol{null}_H \mid |p| + 2 \le i \le maxReg_{c.m(p)}\} \\
\mathrm{ext}_K & = & v_{m\triangledown} \; \boldsymbol{static}_H \; \boldsymbol{null}_H
\end{array}$$

InvokeVirtual(Class/MSig)

The instruction InvokeVirtual is a method invocation, where the corresponding method must be determined dynamically. The object referenced by the top most entry of the operand stack, together with the given message signature determines the method that has to be invoked. The searched method is the method, that overrides or implements the method corresponding to the method signature.

Within the abstract JVM a virtual call does not differ much from a special or static one, up to the fact that we pop the topmost element of the operand stack first, that then is used to determine the method-type for the newly introduced method-vertex. The method-type can be determined from the static environment *cEnv* of the program by $method_{cEnv}(\mathrm{type}(v), c.msig))$.

$$\frac{inst(H) = \mathsf{InvokeVirtual(c.msig)} \qquad pop(H) = (H', v)}{H \to call(H', method_{cEnv}(\mathrm{type}(v), c.msig))}$$

where as before $call(H, c.m(p)) = \; setSuc(K, op, \; peek(H, |p| + 1))$ with K as above.

Terminal States

Recall that transition relations are always total (Definition 3.12 on page 63). Thus all states need to have outgoing transitions (including the terminal ones). Therefore we add a self loop to all the states not covered by the above transition rules.

$$\frac{inst(H) = \bot}{H \to H}$$

In the following we denote the concrete transition relation defined by the above set of transition rules by $\rhd$.

5.4.3 Abstract Semantics

The definition of the abstract semantics is based on Theorem 3.6 on page 68. That is we get an over-approximation of the concrete transition function $\rhd$ by combining it with an concretisation and abstraction function. The used concretisation and abstraction are presented in this section.

Abstract JVM states

In Chapter 3 we used nonterminals to represent abstract parts of a heap. In Example 5.3 on page 101 we already adapted this abstraction to include parts of the method stack, i.e. we extended the technique towards abstraction of JVM states. To represent an abstract JVM state, we consider a set of nonterminals N as defined before, where we restrict *types* over N to $\mathit{types} : N \to (\mathsf{Class}_{\triangledown\blacktriangle} \cup \mathsf{Class/MSig}_{\triangledown\blacktriangle} \cup \mathsf{Interface}_{\blacktriangle} \cup \{\bot_{\blacktriangle}, \top_{\blacktriangle}\})^{\star}$, i.e. only class- and method-nodes can be connected to $\triangledown$-tentacles. This restriction ensures that interfaces, $\bot$, and $\top$ can be used as types for references but no objects of these types are generated, as for any $H \in \mathrm{HC}_\Sigma$ it holds that $\mathrm{type}_H(v) \in \mathsf{Interface} \cup \{\top, \bot\} \Rightarrow v_{\blacktriangle} \in [\mathrm{ext}]$.

For concretisation and abstraction of JVM states we use the functions presented in Section 4.3. For the abstraction we restrict the set of abstractable vertices to keep some information concrete that is important for the actual execution. In the following we discuss which vertices we want to keep concrete.

Restricted abstraction

The short-term behaviour of a JVM primarily depends on the state of the active method frame and static variables. Given a JVM state $H \in \mathrm{HC}_\Sigma$, static variables are represented by the successors of the vertex $\boldsymbol{static}_H$:

$$V_H^{static} = \{\mathrm{att}(e)[2] \mid e \in \triangledown_H(\boldsymbol{static}_H)\} \cup \{\boldsymbol{static}_H\}.$$

We keep a method-frame concrete by keeping the local variables as well as its complete operand stack concrete. Given a method-vertex v_m ($\mathrm{type}(v_m) \preceq \mathsf{method}$) the local variables of the represented method are represented by the set

$$\mathit{variables}(H, v_m) = \{\ \mathit{suc}(H, v_m, r_i) \mid r_i \in \mathit{fields}(\ \mathrm{type}_H(v_m)\ )\}.$$

For v_m the vertex $v_f = \ \mathit{suc}(v_m, \mathit{op})$ represents the top of the operand stack. The set of all vertices representing the operand stack can be described as

$$\mathit{opStack}(H, v_m) = \{\ \mathit{suc}_H^i(v_f, \mathit{next}) \mid 0 \le i < k,\ \mathit{with}\ \ \mathit{suc}_H^k(v_f, \mathit{next}) = \boldsymbol{null}_H\}.$$

Also the objects references stored in the operand stack should be kept concrete. The set of vertices representing referenced objects can be described as

$$opStackEntries(H, v_m) = \{\ suc(v, \mathsf{value}) \mid v \in opStack(H, v_m)\}.$$

Given a JVM state $H \in \mathrm{HC}_\Sigma$, the set of all vertices that we should keep concrete for some method represented by method-vertex $v_m \in V_H$ is the set

$$methodVertices(H, v_m) = variables(H, v_m) \cup opStack(H, v_m) \cup opStackEntries(H, v_m).$$

If we do not have recursive methods we could keep any method-frame concrete, as the number of method-vertices is then finite. For recursive methods we keep for each method only the most recent method-frame concrete. Using the shortcut $m(H, i) = suc^i(H, \mathtt{method}\ calledBy)$ to refer to the ith frame on the method stack we define the set of concrete methods as

$$keepMethod(H) = \{v \in V_H \mid \exists i \in \mathbb{N}.\ v = m(H, i) \wedge \forall j < i.\ \mathrm{type}(v) \neq \mathrm{type}(m(H, j))\}.$$

Consequently the set of vertices of a JVM state $H \in \mathrm{HC}_\Sigma$ that should kept concrete because of its affiliation to an abstractable method is

$$V_H^{\mathtt{method}} = \{v \in methodVertices(H, v_m) \mid v_m \in keepMethod(H)\}.$$

Based on the above discussion we define for data structure grammar $G \in \mathrm{DSG}_\Sigma$ and abstract JVM state $H \in \mathrm{HC}_\Sigma$ the abstraction function

$$stateAbstr_G(H) = \ maxAbstr_G(H,\ V_H^{\mathtt{static}} \cup V_H^{\mathtt{method}}).$$

In Figure 5.6 on page 103 the vertices that should be kept concrete are light grey. The set $stateAbstr_G(H)$ contains all the white vertices. Note that one of the grey vertices is abstract. This vertex can be included in an abstraction, as long as it is used as reduction vertex (see Section 4.3.2).

Concretisation

Let $K \subseteq \mathrm{HG}_\Sigma$ be the set of all (abstract) JVM states, for which the external vertices, as well as the operand stack of the active method are concrete. Up to the instructions GetField, PutField and Return all instructions (as defined in Section 5.4.2) exclusively manipulate the successors of the external vertices and the operand stack. That is, give a state $H \in K$ any state H' with $H \rhd H'$ is also in K. Further as the manipulations involved by the transition rules only affect the concrete part these transitions are safe under K (see page 68). We adapt the transition rules for GetField, PutField and Return such that these are also safe under K.

Consider GetField and PutField first. Both instructions access a pointer of the object referenced by the topmost entry of the operand stack. The problem that occurs here is, that for $H \in K$ it is not guaranteed, that the corresponding vertex is concrete. If this is not the case we concretise the vertex, resulting in a set off concretisations, which we manipulate corresponding to the transition rule.

$$\frac{\mathit{inst}(H) = \mathsf{GetField(c.f)} \qquad \mathit{pop}(H) = (H', v) \qquad K \in \ \mathit{conc}(H', v)}{H \to \ \mathit{incPc}(\ \mathit{push}(K, \ \mathit{suc}(K, v, c.f)))}$$

$$\frac{\mathit{inst}(H) = \mathsf{PutField(f)} \qquad \mathit{pop}(H) = (H', v) \qquad \mathit{pop}(H') = (H'', v') \qquad K \in \ \mathit{conc}(H'', v)}{H \to \mathrm{gc}(\ \mathit{incPc}(\ \mathit{setSuc}(K, v', f, v)))}$$

Note that for concrete vertex $v \in V_H$ it holds that $\mathit{conc}(H, v) = \{H\}$, i.e. if the corresponding object is concrete the modified transition rule behaves equal to the original one. Following Lemma 3.9 on page 67 this gives us that the corresponding transition relation is a safe over-approximation.

The modification involved in the transition rule for Return changes the first external vertex from the active method to the one that called the method (that thereby becomes the active one). If this method is recursive it could be abstracted, such that for the result the first external vertex is not concrete anymore. We solve this by prior concretisation of the corresponding method vertex.

$$\frac{\mathit{inst}(H) = \mathsf{Return()} \qquad \mathtt{method}^2_H \neq \mathtt{null}_H \qquad rT(H) = \mathsf{void} \qquad K \in \ \mathit{conc}(H, \mathtt{method}^2_H)}{H \to \ \mathit{incPc}(\mathit{removeTop}(K))}$$

$$\frac{\mathit{inst}(H) = \mathsf{Return()} \qquad \mathtt{method}^2_H \neq \mathtt{null}_H \qquad rT(H) \neq \mathsf{void} \qquad \mathit{pop}(H) = (H', v) \qquad K \in \ \mathit{conc}(H, \mathtt{method}^2_H)}{H \to \ \mathit{incPc}(\ \mathit{push}(\mathit{removeTop}(K), v))}$$

The method $\mathit{removeTop}$, $\mathtt{method}^2_H$, and method rT are defined on page 112 Replacing the corresponding four rules from $\rhd$ with the above ones results in a transition relation $\rhd'$ that is safe and closed under K. Thus given a set of initial states $I \subseteq K$ the resulting transition system $\mathcal{T}_{I,\rhd'}$ is a safe over-approximation of the concrete transition system $\mathcal{T}_{L_G(I),\rhd}$.

Abstraction

Given an transition system $\mathcal{T}_{I,\rhd'}$, we aim in aggregating (infinitely) many states from $S_\mathcal{T}$ to obtain a finite state space. If a (concrete) state space is infinite this is either because of a loop caused by a Goto or because of recursive method calls. Therefore our abstraction strategy is to abstract only at the beginning of loop iterations and after the recursive invocation of methods. This yields that during one loop iteration and the execution of one recursive method any explored information remains concrete, without loosing the effective aggregation of states. Note that states can only be aggregated if the active methods as well as its program counters are equal, as both are kept concrete during abstraction. We combine the corresponding transition rules with the abstraction function $\mathit{stateAbstr}$ as defined above.

For the Goto we perform a $\mathit{stateAbstr}$ if the target line number is smaller than the program counter, as this (potentially) invokes a new iteration of a loop. Depending on

the line number we get the following two (abstract) transition rules that replace the prior (concrete) ones.

$$\frac{inst(H) = \mathsf{Goto(i)} \qquad i > \ intValue_H(\ suc(H, \texttt{method}_H, pc))}{H \rightarrow \ \ setSuc(\ H, \texttt{method}_H, pc,\ \ suc(H, \texttt{static}_H, int(i))\ )}$$

$$\frac{inst(H) = \mathsf{Goto(i)} \qquad i \leq \ intValue_H(\ suc(H, \texttt{method}_H, pc))}{H \rightarrow \ \ stateAbstr(\ \ setSuc(\ H, \texttt{method}_H, pc,\ \ suc(H, \texttt{static}_H, int(i))\ )\ )}$$

For method calls we have to check if the method call is recursive. We use *recursiveCall*, that checks if a corresponding method is already on the method stack.

$$recursiveCall(H, c.msig) \Leftrightarrow \exists\, v \in V_H.\ \mathrm{type}(v) = method_{cEnv}(c.msig)$$

If this is the case we assume the method to be recursive and we call *stateAbstr* on the resulting state. We get the following transition rules.

$$\frac{inst(H) = \mathsf{InvokeStatic(c.msig)} \qquad recursiveCall(H, m.sig)}{H \rightarrow \ \ stateAbstr(call(H, method_{cEnv}(c.msig)))}$$

$$\frac{inst(H) = \mathsf{InvokeStatic(c.msig)} \qquad \neg recursiveCall(H, m.sig)}{H \rightarrow call(H, method_{cEnv}(c.msig))}$$

$$\frac{inst(H) = \mathsf{InvokeSpecial(c.msig)} \qquad recursiveCall(H, m.sig)}{H \rightarrow \ \ stateAbstr(call(H', method_{cEnv}(c.msig)))}$$

$$\frac{inst(H) = \mathsf{InvokeSpecial(c.msig)} \qquad \neg\, recursiveCall(H, m.sig)}{H \rightarrow call(H', method_{cEnv}(c.msig))}$$

$$\frac{inst(H) = \mathsf{InvokeVirtual(c.msig)} \qquad pop(H) = (H', v) \qquad recursiveCall(H, m.sig)}{H \rightarrow \ \ stateAbstr(call(H', method_{cEnv}(\mathrm{type}(v), c.msig)))}$$

$$\frac{inst(H) = \mathsf{InvokeVirtual(c.msig)} \qquad pop(H) = (H', v) \qquad \neg\, recursiveCall(H, m.sig)}{H \rightarrow call(H', method_{cEnv}(\mathrm{type}(v), c.msig))}$$

The method *call* used here is defined in Section 5.4.2 on page 116.

Let $\blacktriangleright$ be the transition relation defined by the transition rules for $\rhd'$ with replaced transitions rules for jumps and method calls as defined above. Note that the new transition rules results from prior ones combined with a succeeding abstraction. As given by Lemma 3.8 on page 66 this results in a safe approximation of $\rhd'$ and thus also a safe approximation of $\rhd$.

5.4.4 Garbage Collection

Due to destructive updates during the execution of Java bytecode programs, some objects could become unreachable. These unreachable objects are considered to be garbage, as they cannot influence the future behaviour of the program. In a JVM these objects are removed by the *garbage collector*. We model the behaviour of the garbage collector by the method gc.

Definition 5.5 (Garbage Collection):
Given a JVM state $H \in HC_\Sigma$, and vertices $n_1, n_2 \in V_H$, we denote by $n_1 \rightsquigarrow n_2$ that n_2 is reachable from n_1, that is

$$n_1 \rightsquigarrow n_2 \quad \textit{iff} \quad \exists e \in \triangledown(n_1), n' \in att(e).\ n' = n_2 \vee n' \rightsquigarrow n_2.$$

Garbage collection removes any vertex from H that is not reachable from one of the external vertices. The remaining vertices are $V' = \{v \in V_H \mid \exists v' \in [ext_H] \rightsquigarrow v\}$ and the set of edges is restricted to $E' = \triangledown(V')$, the set of edges attached to non-garbage vertices via $\triangledown$-tentacles. We denote the result of garbage collection by

$$gc(H) = (V', E', \mathrm{lab}{\restriction}E', \mathrm{type}{\restriction}V', \mathrm{type}{\restriction}E', \mathrm{ext}).$$

Examples considered so far were garbage free. Note that gc over-approximates connectivity if nonterminals are considered. From this it follows that $L_G(\mathrm{gc}(H)) = \mathrm{gc}(L_G(H))$ does not hold in general.

In Figure 5.14 an example for this impreciseness of gc is given. Consider the heap configuration in Figure 5.14(b). None of the vertices is considered to be garbage as they are reachable via $\triangledown$-tentacles from the external vertex. However, given the single grammar rule for X from Figure 5.14(a), the vertices on the right of the nonterminal are garbage.

Nevertheless it holds that $\mathrm{gc}(L_G(\mathrm{gc}(H))) = \mathrm{gc}(L_G(H))$ and as garbage has no influence on the behaviour of a program this is sufficient for correctness. Garbage collection ensures that we get no infinite state space due to garbage vertices. We involve garbage collection into the transition relation by calling gc on each result of a transition.

5.5 Experiments: Lindstrom's Algorithm

We implemented a state space generator for Java bytecode based on the techniques and transition rules presented in this chapter. The tool is called Juggrnaut [HNR10] and expects an abstraction grammar, a program in Java bytecode and a start JVM state

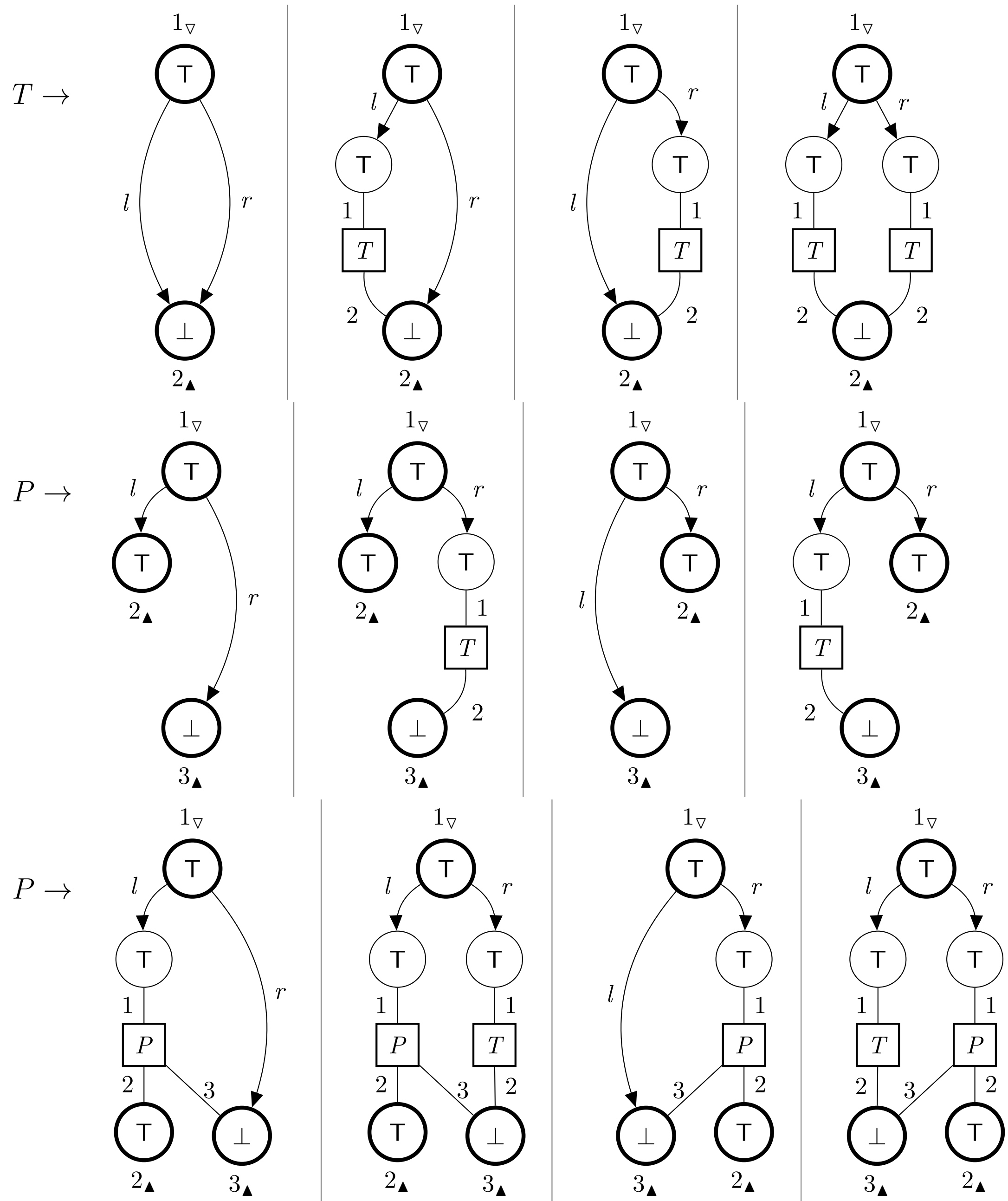
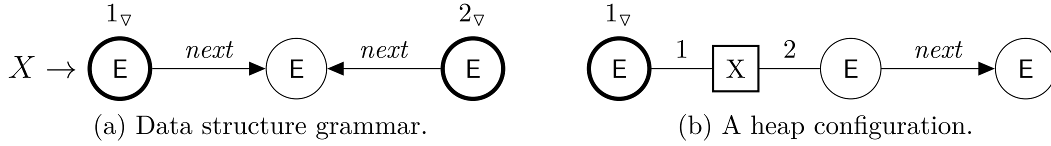


Figure 5.13: Data structure for trees and tree paths.



(a) Data structure grammar.

(b) A heap configuration.

Figure 5.14: A grammar for which gc is not precise.

as input and produces the abstract state space of all JVM states reachable from the given start configuration. We used the tool to generate the state space for Lindstrom's algorithm on binary trees [HNR10]. In this section we consider the used inputs and we discuss some aspects of the output.

In Section 4.4 we considered a DSG for trees and tree paths which are fully branched. The restriction to fully branched trees was motivated by the enormous amount of HRG rules necessary to represent arbitrary branched trees. In Section 3.6 we noted that the increase in rules origins from the variety of different vertex types imposed by the implicit representation of *null*-pointers. Now that we represent *null* explicitly we can adapt the grammar such that it represents arbitrary trees without increasing the rule count. The adapted grammar is given in Figure 5.13. Note that in order to establish backward confluence an additional $P \rightarrow T$ rule would be necessary (see Section 3.6).

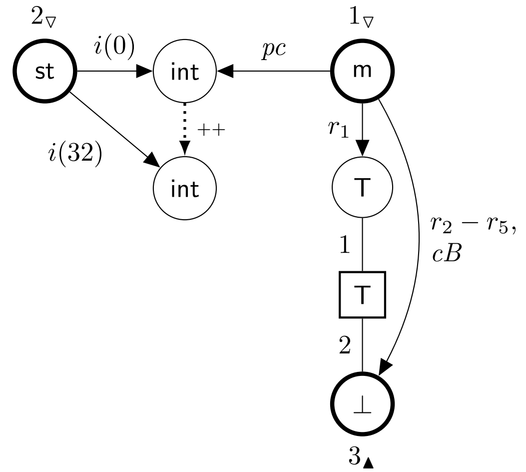
The Lindstrom algorithm from Figure 1.1a compiled to Java bytecode is given in Figure 5.15(a). The local registers (one to five) correspond to the variables *root*, *sen*, *prev*, *cur* and *next*, in this order. The first four lines correspond to the first line of the methods Java code: sen = **new** Tree();. Observe the method invocation in the third line, where the Tree constructor is called. In line four to seven the variables *sen*, *prev* and *cur* are initialised. The remaining lines up to line 32, which is the exit point of the method, contain the while-loop. Here lines eight through eleven realise the check of the while condition, while the remaining implements the pointer rotation.

In order to complete the input for the state space generation we determine a start configuration. Figure 5.15(b) depicts a JVM state after the invocation of method traverse(tree), where the parameter for *root* refers to the root of an arbitrary tree. The variable *root*, represented by the first register, is set to that trees root. As the method was just invoked the program counter is set to the integer vertex representing zero and the local variables (register two to five) are set to *null*.

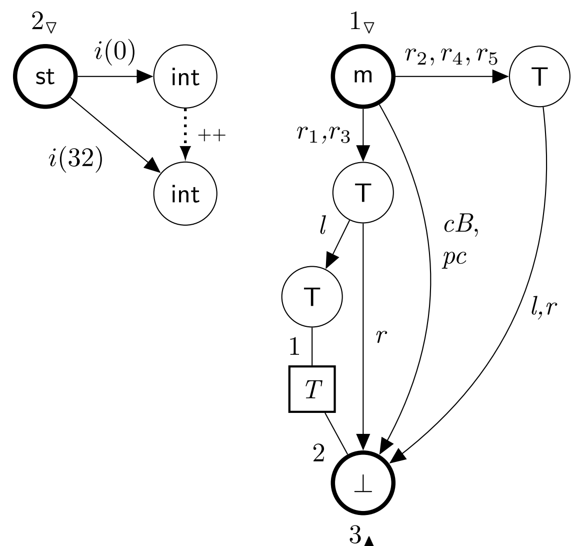
From this start configuration we can reach 4521 configurations from which four are terminal. The four terminal configurations are given in Figure 5.15(c) and Figure 5.16(a) - (c). These terminal configurations are equal up to the tree rooted at variable *root* (register r_1). This tree is either a single leaf (Figure 5.16(a)), has only a left (Figure 5.15(c)), respectively right subtree (Figure 5.16(b)) or a right and a left subtree (Figure 5.16(c)). Note that we omit the representation of the static-vertex and its int-fields in Figure 5.16a - (c), as those are the same for all terminal configurations.

```
1  New(Tree)
2  Dupx()
3  InvokeVirtual(Tree.<init>())
4  Store(2)
5  Load(2)
6  Store(3)
7  Load(1)
8  Store(4)
9  Load(4)
10 Load(2)
11 Cond(if_eq, 32)
12 Load(4)
13 GetField(Tree.left)
14 Store(5)
15 Load(4)
16 Load(4)
17 GetField(Tree.right)
18 PutField(Tree.left)
19 Load(4)
20 Load(3)
21 PutField(Tree.right)
22 Load(4)
23 Store(2)
24 Load(5)
25 Store(4)
26 Load(4)
27 Cond(if_neq_null, 31)
28 Load(3)
29 Store(4)
30 Push(null)
31 Store(3)
32 Goto(8)
33 Return()
```

(a) Java bytecode



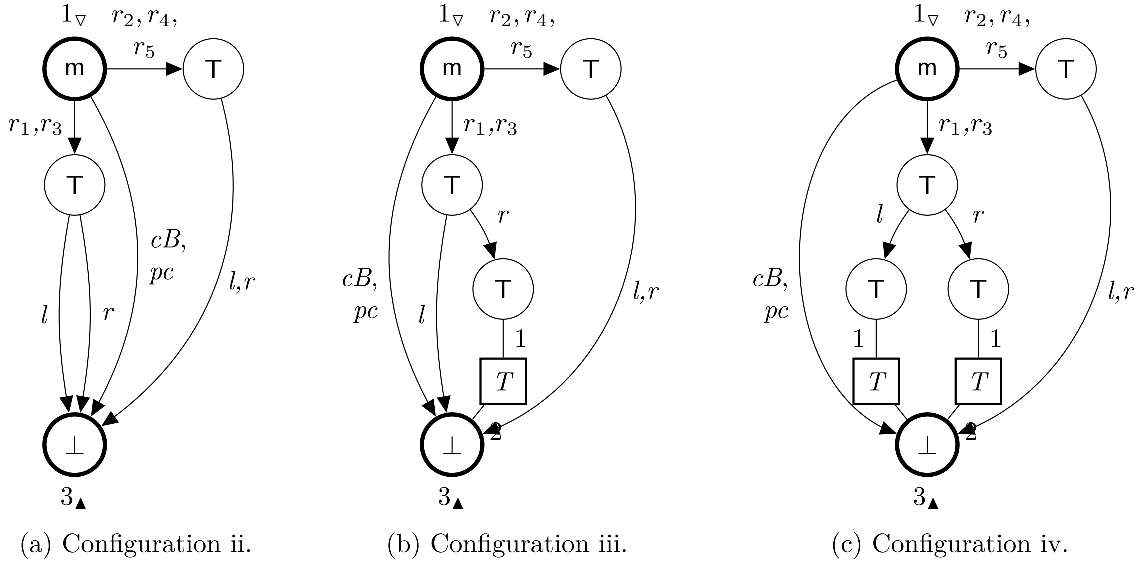
(b) Start configuration



(c) Terminal configuration i

Figure 5.15: Analysis of Lindstrom's algorithm.

If we would abstract the trees at *root* we would get a single T-edge for each terminal state, i.e. all terminal states would be equal. However, the corresponding sub-heap is not abstracted due to the abstraction restrictions introduced in Section 5.4.3 on page 117, where we excluded those vertices from the abstraction which are referred to by active registers. If we would relax the restriction also the newly generated leaf refereed by *sen* (register r_2) would be abstracted to an T-edge. This would not have a negative influence to the analysis, but would at least be inconvenient.



(a) Configuration ii. (b) Configuration iii. (c) Configuration iv.

Figure 5.16: Terminal configurations for Lindstrom's algorithm.

Our tool can generate the 4521 reachable states in less then $400ms$ using $15Mb$ of RAM. We verified the absence of *null*-pointer deferences, as all terminal states are proper. From the shapes of the terminal configurations we can manually conclude that the resulting structure is a tree (shape preservation).

The performance compares quite well to its competitors. We are aware of two other automated proofs of Lindstrom's algorithm. One by regular tree model checking [Bou+06] consuming three minutes and 14 seconds and another by TVLA [Bog+07] taking 8.21 seconds. As mentioned in Section 1.4 for the latter a set of 24 manually provided instrumentation predicates were needed which encode a deeper knowledge about the algorithm, while for our approach a general tree and tree-path grammar is sufficient. Both referenced experiments verified the absence of null pointer dereferencing, absence of memory leaks and the shape invariance, i.e. the treeness of the heap. Note that as we analyse Java bytecode we do not consider memory leaks, but we could verify that no garbage is collected (Section 5.4.4) without additional costs. In [Bog+07] Lindstrom's algorithm is modified to ensure the treeness of all intermediate states to reduce the complexity of the analysis. We keep the original version and verify treeness for the terminal states only. In this chapter the treeness of the terminal states, however, is

concluded manually from the resulting abstract hypergraphs. In the next chapter we will consider how to check shape properties automatically. Further we defined additional properties of Lindstrom's algorithm in Section 1.1, namely termination, completeness and that the resulting tree is exactly the same as the input tree. To verify these properties we need to check temporal properties and describe properties of object identities, which we consider in Chapter 7.

5.6 Conclusions

In this chapter we extended the foundations and techniques introduced in earlier chapters towards an analysis technique for Java bytecode programs. For this purpose we developed two major ingredients. First we extended the typed heap configurations as introduced in Chapter 4 towards models for complete states of a JVM (in Section 5.3) including the representation of *null*, static and local variables and the method stack as well as boolean variables. The second ingredient are the transition rules that model the Java bytecode semantics (Section 5.4). We first developed a concrete semantics for each Java byecode instruction, which then along the lines of Section 3.5 can be combined with an abstraction grammar to obtain an abstract semantics.

Finally we reconsidered Lindstrom's algorithm (Section 1.1 on page 2) in Section 5.5, where we presented the Java bytecode of the algorithm, an abstraction grammar for trees and tree-path and a start configuration suitable for the analysis of the algorithm on arbitrary binary trees. We also give the experimental results that we got from a prototype implementation of the approach and compared those to results from regular tree model checking [Bou+06] and TVLA [Bog+07].

Chapter 6
Logics over Hypergraphs

So far we considered HRGs to describe sets of hypergraphs. Like sets of strings, sets of hypergraphs cannot only be described via grammars but also by logic formulae. In this chapter we consider two logics over hypergraphs, that are also well-known for strings. The first one is *first order logic* (FO) allowing quantification over vertices and edges, the second one is an extension of FO called *monadic second order logic* (MSO) allowing quantification over sets of vertices and edges.

Our aim is to describe properties of JVM states by logical formulae, i.e. properties of concrete as well as of abstract heap configurations. Note that an abstract state represents a set of graphs. Thus we have to check for each represented graph (potentially infinitly many) whether the property of interest is fulfilled. As not necessarily all graphs fulfil or reject a property we want to know if *all*, *some* or *none* of the graphs fulfil the property.

A well-known theorem by Courcelle [Cou90] states that for a HRG and an MSO-formula it is decidable whether the language of the grammar contains graphs fulfilling the property described by the formula. This theoretical result has not been implemented so far, as the underlying construction is not suitable for practical purpose. In this chapter we present an alternative proof for Courcelles theorem, based on a construction that proved to be suitable for various practical problems.

This chapter is organised as follows. In Section 6.1, we define the logic FO and its semantics. In Section 6.2, we introduce an algorithm that for a HRG and an FO-property determines a new grammar that can be used to define the set of graphs from the HRG fulfilling the property, as well as the ones not fulfilling the property. This algorithm gives us a procedure to decide whether one, some or all of the graphs fulfil the property. In Section 6.3, we define MSO and adapt the FO-algorithm from Section 6.2 towards MSO.

The algorithm for FO and MSO is introduced for general (labelled) hypergraphs, i.e. we consider graphs over ranked alphabets, not considering any typing of nodes. However, the general approach can easily be adapted to heap configurations. In Section 6.5, we

provide these adaptions and we discuss how the algorithm can be simplified for heap graphs.

6.1 First Order Logic

First order logic essentially is an extension of propositional logic by existential and universal quantification over elements. In the case of graphs the elements are the edges and vertices.

Definition 6.1 (First order logic formulae over graphs):
Given a (ranked) alphabet Σ, a set of free vertex variables X_V, and a set of free edge variables X_E, a first order formula $\varphi(X_V, X_E)$ *over alphabet Σ is recursively defined as follows:*

$$\begin{aligned}\varphi(X_V, X_E) \;:=\; & \exists_V\, x.\ \varphi(X_V \uplus \{x\}, X_E) \mid \forall_V\, x.\ \varphi(X_V \uplus \{x\}, X_E) \mid \\ & \exists_E\, e.\ \varphi(X_V, X_E \uplus \{e\}) \mid \forall_E\, e.\ \varphi(X_V, X_E \uplus \{e\}) \mid \\ & \varphi(X_V, X_E) \wedge \varphi(X_V, X_E) \mid \varphi(X_V, X_E) \vee \varphi(X_V, X_E) \mid \\ & \neg\varphi(X_V, X_E) \mid \pi(X_V, X_E)\end{aligned}$$

where $\pi(X_V, X_E)$ is a predicate over vertex variables X_V and edge variables X_E, with $x, y \in X_V$, $e, f \in X_E$, $a \in \Sigma$ and $\bar{x} \in X_V^\star$, where all entries of $\bar{x}$ are pairwise distinct, defined as follows:

$$\pi(X_V, X_E) \;:=\; x = y \mid e = f \mid \mathrm{att}(e) = \bar{x} \mid \mathrm{lab}(e) = a$$

We denote the set of all first order formulae *over a ranked alphabet Σ by FO_Σ. The set of edge and vertex variables occurring in a formula $\varphi \in FO_\Sigma$ is denoted by Var_φ and the ordered set of predicates used in φ by $Pred\varphi$.*
A formula $\varphi(X_V, X_E) \in FO_\Sigma$ without free variables, i.e. $X_V = X_E = \emptyset$, is called a sentence *and abbreviated by φ. A formula without quantifiers, i.e. generated without the use of the first two rows, is said to be* quantifier-free.

Note that we allow quantification over vertices as well as over edges. For general hypergraphs, where edges are proper objects, this is somehow natural. In the literature this is referred to as FO_2 while FO_1 allows quantification over vertices only.

We write $\forall_V\, x, y.\ \varphi$ instead of $\forall_V\, x.\, \forall_V\, y.\ \varphi$, and for $\exists_V, \forall_E$ and $\exists_E$ accordingly.

We give the semantics of FO-formulae as a model relation, defined recursively over the structure of formulae. Given a hypergraph $H \in \mathrm{HG}_\Sigma$, a first order formula $\varphi(X_V, X_E) \in$

FO_Σ over the same alphabet Σ and a variable interpretation $\alpha : (X_V \to V_H) \cup (X_E \to E_H^T)$, we say the graph H under the interpretation α is a model of the formula $\varphi(X_V, X_E)$, denoted by $(H, \alpha) \models \varphi(X_V, X_E)$, if:

$(H,\alpha) \models x = y$	iff $\alpha(x) = \alpha(y)$
$(H,\alpha) \models e = f$	iff $\alpha(e) = \alpha(f)$
$(H,\alpha) \models \mathrm{att}(e) = \bar{x}$	iff $\mathrm{att}_H(\alpha(e)) = \alpha\bar{x}$
$(H,\alpha) \models \mathrm{lab}(e) = a$	iff $\mathrm{lab}_H(\alpha(e)) = a$
$(H,\alpha) \models \neg\varphi(X_V, X_E)$	iff *not* $(H,\alpha) \models \varphi(X_V, X_E)$
$(H,\alpha) \models \varphi_1(X_V, X_E) \wedge \varphi_2(X_V, X_E)$	iff $(H,\alpha) \models \varphi_1(X_V, X_E)$ *and* $(H,\alpha) \models \varphi_2(X_V, X_E)$
$(H,\alpha) \models \varphi_1(X_V, X_E) \vee \varphi_2(X_V, X_E)$	iff $(H,\alpha) \models \varphi_1(X_V, X_E)$ *or* $(H,\alpha) \models \varphi_2(X_V, X_E)$
$(H,\alpha) \models \exists_V\, x.\, \varphi(X_V \uplus \{x\}, X_E)$	iff there exists a vertex $v \in V_H$ such that $(H, \alpha[x \mapsto v]) \models \varphi(X_V \uplus \{x\}, X_E)$.
$(H,\alpha) \models \forall_V\, x.\, \varphi(X_V \uplus \{x\}, X_E)$	iff for any vertex $v \in V_H$ it holds that $(H, \alpha[x \mapsto v]) \models \varphi(X_V \uplus \{x\}, X_E)$.
$(H,\alpha) \models \exists_E\, e.\, \varphi(X_V, X_E \uplus \{e\})$	iff there is an edge $f \in E_H^T$ such that $(H, \alpha[e \mapsto f]) \models \varphi(X_V, X_E \uplus \{e\})$.
$(H,\alpha) \models \forall_E\, e.\, \varphi(X_V, X_E \uplus \{e\})$	iff for any edge $f \in E_H^T$ it holds that $(H, \alpha[e \mapsto f]) \models \varphi(X_V, X_E \uplus \{e\})$.

For $\varphi, \psi \in \mathrm{FO}_\Sigma$ we use $\varphi \to \psi$ (implication) as shortcut for $\neg\varphi \vee \psi$ and $\varphi \leftrightarrow \psi$ (equivalence) as shortcut for $(\varphi \wedge \psi) \vee (\neg\varphi \wedge \neg\psi)$. Instead of $\neg(x \in X)$ we write $x \notin X$. For $\varphi \in \mathrm{FO}_\Sigma$ we write $H \models \varphi$ instead of $(H, \emptyset) \models \varphi(\emptyset, \emptyset)$.

Note that we quantify over terminal edges only. That is because nonterminal edges are replaced and not contained by any graph of the resulting language. Correspondingly the language defined by a FO-sentence is a set of terminal graphs.

Definition 6.2 (FO-Language):
The language defined by $\varphi \in FO_\Sigma$ is the set

$$L(\varphi) = \{H \in HG_{T_\Sigma} \mid H \models \varphi\}.$$

Definition 6.3 (Logical equivalence):
Formulae $\varphi(X_V, X_E), \psi(X_V, X_E) \in FO_\Sigma$ are logically equivalent iff any model of φ is also a model of ψ and vice versa, i.e. $\forall H \in HG_\Sigma$ and $\forall \alpha \in X_V \to V_H \cup X_E \to E_H^T$

$$(H,\alpha) \models \varphi(X_V, X_E) \quad \Leftrightarrow \quad (H,\alpha) \models \psi(X_V, X_E).$$

Example 6.1: FO-formula
A doubly linked list is a list where two succeeding elements are connected by a next

(n) edge, from the prior to the latter node, as well as a previous *(p) edge from the latter to the prior. The following FO-formula expresses that any two nodes that are connected by a* next *edge are also connected by a* previous *edge with reverse direction and vice versa:*

$$\varphi_{dll} := \forall_V\, x, y.\ (\,\exists_E\, e.\ (\mathrm{lab}(e) = n \wedge \mathrm{att}(e) = xy) \leftrightarrow \exists_E\, e.\ (\mathrm{lab}(e) = p \wedge \mathrm{att}(e) = yx)\,)$$

We can rewrite the formula in the following logically equivalent one:

$$\begin{array}{lll} \varphi_{dll'} & := & \forall_V\, x, y. \\ & & (\ \ \exists_E\, e.\ (\mathrm{lab}(e) = n \wedge \mathrm{att}(e) = xy) \wedge \ \ \exists_E\, e.\ (\mathrm{lab}(e) = p \wedge \mathrm{att}(e) = yx)) \\ & \vee & (\neg\, \exists_E\, e.\ (\mathrm{lab}(e) = n \wedge \mathrm{att}(e) = xy) \wedge \neg\, \exists_E\, e.\ (\mathrm{lab}(e) = p \wedge \mathrm{att}(e) = yx)) \end{array}$$

6.1.1 Prenex Normal Form

In Section 6.2 we present a strategy for evaluating FO-formulae over terminal hypergraphs. This strategy is based on formulae in prenex normal form. The definition of prenex normal form is given below.

Definition 6.4 (Prenex normal form [BM77]):
A FO-formula $\varphi \in FO_\Sigma$ is prenex *if it has the form*

$$\mathcal{Q}_1 x_1 . \mathcal{Q}_2 x_2 . \ldots \mathcal{Q}_k x_k .\ \beta,$$

where $k \geq 0$ and for each $1 \leq i \leq k$, $\mathcal{Q}_i \in \{\forall_V, \exists_V, \forall_E, \exists_E\}$, and β is quantifier-free. We call $\mathcal{Q}_1 x_1, \mathcal{Q}_2 x_2, \ldots, \mathcal{Q}_k x_k$ the prefix *and β the* matrix*.*

A prenex normal form is a prenex formula such that the variables $x_1, \ldots, x_k$ are pairwise distinct, and all of them occur within the matrix β.

Given an arbitrary FO-formula $\varphi \in FO_\Sigma$, a prenex form *for φ is a prenex normal form formula logically equivalent to φ.*

Example 6.2:
The following formula is in prenex normal form and describes the property that we considered before in Example 6.1.

$$\begin{array}{lll} \varphi_{dll} & := & \forall_V\, x, y.\ \exists_E\, e_1, e_2.\ \forall_E\, e_3, e_4. \\ & & (\ \ \ (\mathrm{lab}(e_1) = n \wedge \mathrm{att}(e_1) = xy) \wedge \ \ \ (\mathrm{lab}(e_2) = p \wedge \mathrm{att}(e_2) = yx)\,) \\ & \vee & (\neg\, (\mathrm{lab}(e_3) = n \wedge \mathrm{att}(e_3) = xy) \wedge \neg\, (\mathrm{lab}(e_4) = p \wedge \mathrm{att}(e_4) = yx)\,) \end{array}$$

Note that there are more variables involved as in the non-prenex case. That is because it is not possible to reuse variables as we did before for the variable e in Example 6.1. Here variable e is divided into the four variables e_1, e_2, e_3, and e_4, one for each use of e in the non-prenex formula.

In the above example we give a logical equivalent formula in prenex normal form, for the formula from Example 6.1. Indeed we can give an equivalent formula in prenex normal form for any FO-formula, as the following theorem states.

Theorem 6.1 (Existence of prenex normal form [BM77]):
For any FO-formula there exists a logically equivalent FO-formula in prenex normal form.

A construction for the prenex normal form, starting at an arbitrary formula, can be found in [BM77].

6.2 FO and Graph Grammars

A FO-formula $\varphi \in \mathrm{FO}$ describes a properties of terminal hypergraphs. It thus define a set of hypergraphs, namely the set $L(\varphi)$ of hypergraphs fulfilling the described property. Given $G \in \mathrm{HRG}_\Sigma$ and $H \in \mathrm{HG}_\Sigma$, we want to know which graphs in the set $L_G(H)$ fulfil the property φ. That is, we want to determine the intersection of the language defined by the FO-formula and the language defined by hyperedge replacement grammar, i.e. $L(\varphi) \cap L_G(H)$. For monadic second order logic (MSO), an extension of first order logic and strictly more expressive, there is a theorem by Courcelle [Cou90], that states that the intersection of an MSO-formula defined hypergraph set and an HRG defined set is HRG-definable. The following theorem is an adaption of Courcelle's theorem to FO and to our notations.

Theorem 6.2 (Intersection of FO and HRG [Cou90]):
Given $G \in HRG_\Sigma$, $H \in HG_\Sigma$ and a sentence $\varphi \in FO_\Sigma$, there exists $G' \in HRG_{\Sigma'}$ and $H' \in HG_{\Sigma'}$ with $L_{G'}(H') = L(\varphi) \cap L_G(H)$.

As the emptiness problem is decidable for HRGs [Hab92], we can decide if the intersection is empty, i.e. if *none* of the graphs in $L_G(H)$ fulfil the property φ. If the intersection is not empty, there is at least one graph in $L_G(H)$ fulfilling the property. As FO is closed under negation we can also determine the set of graphs *not* fulfilling the property as intersection of $L_G(H)$ and $L(\neg\varphi)$. In the case that $L_G(H) \cap L(\neg\varphi)$ is empty, *all* graphs from $L_G(H)$ fulfil the property φ.

Corollary 6.1 (Decidability of FO properties over HRGs)
Given $G \in HRG_\Sigma$, $H \in HG_\Sigma$ and a sentence $\varphi \in FO_\Sigma$, it is decidable if none, some *or* all *graphs from $L_G(H)$ fulfil the property φ.*

Courcelle's proof of Theorem 6.2 consists of a transformation of the FO formula into a kind of automaton over graphs, that then can be intersected with an equation system that is obtained from the HRG. Thus from a theoretical point of view Courcelle gives an algorithm to decide if all the terminal graphs described by a nonterminal graph fulfil an FO-property. However, the given algorithm is not suitable in practice, as the size of the corresponding automaton in not elementary in the size of the formula.

In this section we develop an algorithm for evaluating FO-formulae over sets of terminal graphs described by an abstract hypergraph and a corresponding grammar. The algorithm avoids the automaton construction (and as we will see in Section 6.4) has practical relevance. The algorithm is based on *evaluation trees*, a recursive representation of all possible variable interpretations. In Section 6.2.2, we introduce evaluation trees for terminal graphs, which are a direct implementation of the FO-semantics. In Section 6.2.4, we consider hyperedge replacements as defined in Definition 2.6 on page 19. We extend the definition of evaluation trees such that for any hyperedge replacement we can compose the evaluation tree of the result from the evaluation trees of the involved graphs. In Section 6.2.8, we will use the concept of evaluation trees to evaluate HRGs. The result is an algorithm that given a grammar $G \in \mathrm{HRG}_\Sigma$ and a hypergraph $H \in \mathrm{HG}_\Sigma$ determines if either *each*, *some* or *all* of the graphs from $L_G(H)$ fulfil the property given by some FO-formula.

As byproduct the algorithm yields a new grammar which can be used to describe the set of hypergraphs of $L_G(H)$ fulfilling the formula as well as the set of graphs refuting the formula. That is, we provide an alternative proof for Theorem 6.2 as well as a construction for the intersection.

6.2.1 Grammar Adjustment

In this chapter we consider only graphs with nonterminal edges that are attached to pairwise distinct vertices. We impose this assumption, as it significantly simplifies the algorithms and proofs while the restrictions are only syntactically, as shown by the following result.

Lemma 6.1 (Pairwise distinct attachments):
For any $G \in HRG_\Sigma$ exists a $G' \in HRG_{\Sigma'}$ such that $G \approx G'$ and for each $X \to R \in G$, $e \in E_R^N$ it holds that

$$\mathrm{att}(e)[i] = \mathrm{att}(e)[j] \quad \Longrightarrow \quad i = j \qquad \forall 1 \leq i, j \leq |\mathrm{att}(e)|.$$

Proof. We prove Lemma 6.1 by constructing grammar $G' \in \mathrm{HRG}_\Sigma$ from G.

Nonterminal edges attached to a single vertex via multiple tentacles, induce a merging of external vertices during their replacement. The idea behind the construction is to realise

this merging in advance within the corresponding rule graphs. To do so we introduce new nonterminals from which graphs are derivable that equal the ones derivable from the original ones, but with merged external vertices.

Let $\mathbb{N}^{\leq k} = \{n \in \mathbb{N} \mid n \leq k\}$ represent a set of k tentacles. We consider any possible partition of tentacles. The set of partitions for $\mathbb{N}^{\leq n}$ is defined as

$$\mathit{Partition}(n) = \{P \subset \mathcal{P}(\mathbb{N}^{\leq n}) \mid \forall p_1, p_2 \in P.\ p_1 \cap p_2 = \emptyset \wedge \bigcup_{p \in P} p = \mathbb{N}^{\leq n}\}$$

Given a ranked alphabet Σ we introduce new nonterminals N' as combinations of nonterminals from N_Σ and partitions:

$$N' = \{(X, T) \mid X \in N_\Sigma \wedge T \in \mathit{Partition}(\mathrm{rk}(X))\}$$

Each block in T represents one tentacle of the nonterminal (X, T), thus $\mathrm{rk}((X, T)) = |T|$. Given a graph $H \in \mathrm{HG}_\Sigma$ we replace the nonterminal edges in H such that multiple tentacles of a nonterminal edge - attached to the same vertex - are represented by a single one. For an edge $e \in E_H$ we get the partition of its tentacles, that corresponds to its attachment by

$$T(e) = \{\{i \in \mathbb{N} \mid \mathrm{att}(e)[i] = v\} \mid v \in \mathrm{att}_R(e)\},$$

i.e. two tentacles are attached to the same vertex if and only if their numbers are in the same block. We let the function $\gamma_{G \to G'}(H) = (V_H, E_H, \mathrm{att}', \mathrm{lab}', \mathrm{ext}_H)$ with

$$\begin{array}{lcl} \mathrm{lab}'(e) & = & (\,\mathrm{lab}(e), T(e)\,) \\ \mathrm{att}'(e)[i] & = & \mathrm{att}(e)[j],\ \mathit{with}\ j \in \mathit{order}(T(e))[i], \end{array}$$

where $\mathit{order}(T(e))$ returns the linearly ordered sequence of the elements in $T(e)$.

We construct the quasi-equivalent grammar G' by generating a new rule for each combination of a rule from G and a new nonterminal from N'. We get the new rule graphs by first merging external vertices corresponding to T, and then replacing nonterminal edges with their correspondence from N'. We get $G' = \{(X, T) \to \gamma_{G \to G'}(\mathit{merge}(R, T)) \mid X \to R \in G\}$ with $\mathit{merge}(R, T) = (V_R \setminus [\mathrm{ext}_R] \cup T, E_R, \mathrm{lab}_R, \mathrm{att}, \mathit{order}(T))$, where

$$\mathrm{att}(v) = \begin{cases} t \in T & ,\ \mathit{if}\ \exists i \in \mathbb{N}^{\leq \mathrm{rk}(X)}.\ v = \mathrm{ext}_R[i] \wedge i \in T \\ v & ,\ \mathit{otherwise} \end{cases}$$

The function $\gamma_{G \to G'}$ ensures that for any nonterminal edge the sequence of attached vertices is pairwise distinct. The construction yields a grammar where external vertices, which previously where merged via hyperedge replacements are merged in advanced within the rule graphs. Thus the language of the grammar does not change and we get $G \approx G'$.

□

Example 6.3:
Figure 6.1(a) depicts the production rules of $G \in HRG_\Sigma$ for nonterminal X. Between the two tentacles of a X-labelled edge arbitrary many a-edges can be derived. The grammar $G' \in HRG_{\Sigma'}$, resulting from applying the construction from the proof of Lemma 6.1, is given in Figure 6.1(b). Note that there are a-edges for which both tentacles are attached to the same vertex. That is admissible as the restriction is only imposed on nonterminal edges. For graph H depicted in Figure 6.1(c) any derivable graph has two vertices, and arbitrarily many a-edges in-between. Additionally there are arbitrary a-edges attached with both tentacles at the second vertex. Note that the left X-edge in H does not comply with the requirements. Figure 6.1(d) depicts the graph $\gamma_{G\to G'}(H)$, that fulfils the requirements and it holds that $L_G(H) = L'_G(\gamma_{G\to G'}(H))$.

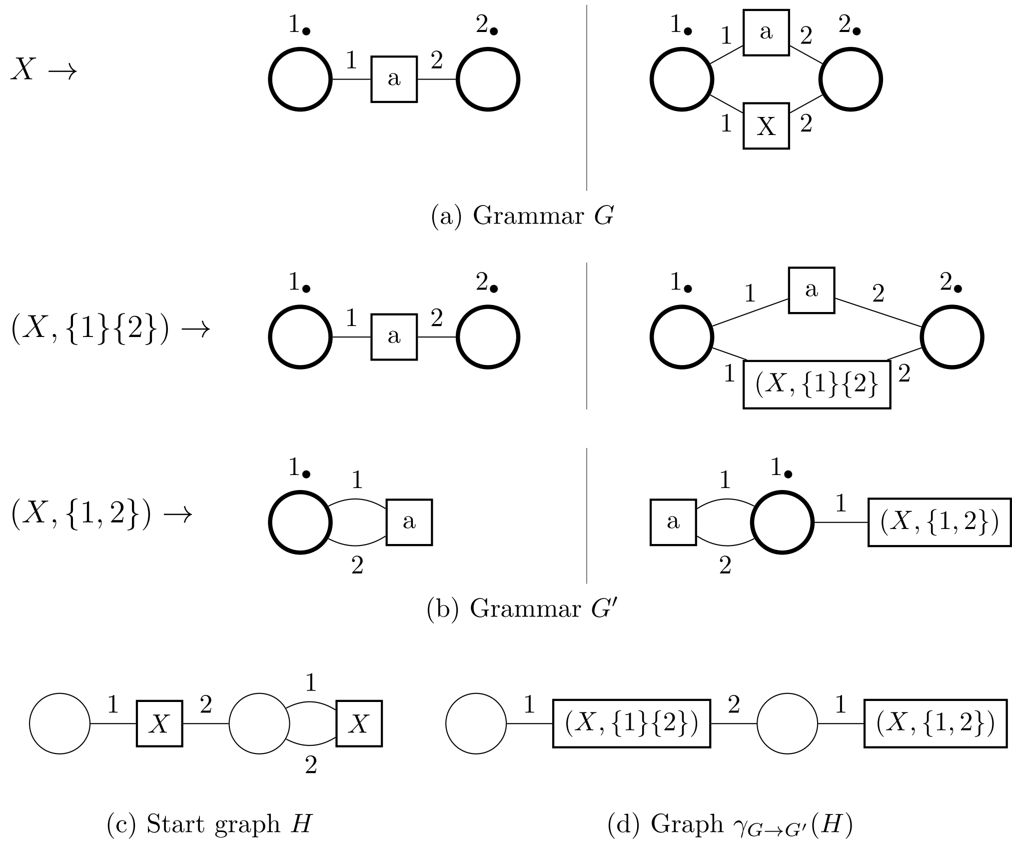


Figure 6.1: Grammar Adjustment

In the remaining of this chapter we consider any nonterminal edge to be attached to pairwise distinct vertices.

6.2.2 Evaluation Trees for Terminal Graphs

An *evaluation tree* is a directed labelled graph used to check whether $(H, \alpha) \models \varphi(X_V, X_E)$ for $H \in \mathrm{HG}_{T_\Sigma}$, $\varphi(X_V, X_E) \in \mathrm{FO}_\Sigma$ in prenex normal form and variable interpretation $\alpha : X_V \to V_H \times X_E \to E_H^T$. Any path from the root of the evaluation tree to a leaf represents an assignment for all variables from the prefix of φ to elements of H. An evaluation tree is similar to a binary decision diagram (BDD) [Knu08; BK08], where the possible variable interpretations are not just true or false but (depending on the variable) vertices or edges of the graph.

Definition 6.5 (Evaluation tree):
Given $H \in HG_\Sigma$, FO-sentence $\varphi(X_V, X_E) \in FO_\Sigma$ in prenex normal form and a variable interpretation $\alpha : (X_V \to V_H) \cup (X_E \to E_H^T)$. An evaluation tree *is a tuple $t = (N, L, quant, var, eval, \delta, root)$, with*

N	*a set of (inner) nodes,*
L	*a set of leaves with $N \cap L = \emptyset$,*
$quant : N \to \{\forall, \exists\}$	*a function assigning each node a quantifier,*
$var : N \to Var_\varphi$	*a function assigning each node a variable,*
$eval : L \to \{true, false\}$	*a function assigning each leaf a boolean value,*
$\delta \in N \times (V_H \cup E_H) \times (N \cup L)$	*a transition relation,*
$root \in N \cup L$	*the root node of the evaluation tree.*

We call a node $n \in N$ universal *if $quant(n) = \forall$,* existential *otherwise ($quant(n) = \exists$). We denote the set of all universal nodes of an evaluation tree t by $N_t^\forall$, the set of all existential nodes by $N_t^\exists$. We denote the set of all evaluation trees by Tree, the set of all trees for FO-formula φ by $Tree_\varphi$.*

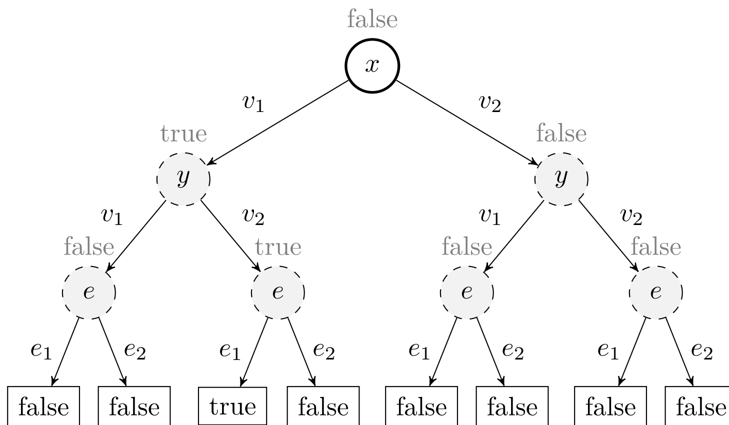


Figure 6.2: Evaluation tree $evaltree(H, \emptyset, \varphi_{suc})$.

Construction of an Evaluation Tree

Given a hypergraph $H \in \mathrm{HG}_\Sigma$, an FO-formula $\varphi(X_V, X_E) \in \mathrm{FO}_\Sigma$ in prenex normal form and a variable interpretation $\alpha : (X_V \to V_H) \cup (X_E \to E_H^T)$ we construct the corresponding evaluation tree $\mathit{evaltree}(H, \alpha, \varphi)$ as follows:

If φ is quantifier-free, the evaluation tree consists of a single leaf, that depending on the evaluation is either true or false.

$$\mathit{evaltree}(H, \alpha, \varphi) = \begin{cases} (\emptyset, \{l\}, \emptyset, \emptyset, [l \mapsto \text{true}], \emptyset, l) & , \mathit{if}\,(H, \alpha) \models \varphi \\ (\emptyset, \{l\}, \emptyset, \emptyset, [l \mapsto \text{false}], \emptyset, l) & , \mathit{otherwise}. \end{cases}$$

If $\varphi = \mathcal{Q}_E\, x.\, \varphi'$ ($\mathcal{Q} = \{\exists, \forall\}$) then we construct for each $e \in E_H^T$ the evaluation tree $t_e = \mathit{evaltree}(H, \alpha[x \mapsto e], \varphi')$, and we let $\mathit{evaltree}(H, \alpha, \varphi) = (N, L, \mathit{quant}, \mathit{var}, \mathit{eval}, \delta, \mathit{root})$, with

$$\begin{array}{lcl} N & = & \biguplus_{e \in E_H} N_{t_e} \uplus \{n\}, \\ L & = & \biguplus_{e \in E_H} L_{t_e}, \\ \mathit{quant} & = & \biguplus_{e \in E_H} \mathit{quant}_{t_e} \uplus [n \mapsto \mathcal{Q}], \\ \mathit{var} & = & \biguplus_{e \in E_H} \mathit{var}_{t_e} \uplus [n \mapsto x], \\ \mathit{eval} & = & \biguplus_{e \in E_H} \mathit{eval}_{t_e}, \\ \delta & = & \biguplus_{e \in E_H} (\delta_{t_e} \uplus \{(n, e, \mathit{root}_{t_e})\}), \\ \mathit{root} & = & n \end{array}$$

We call the node n an *edge node*, as the assigned variable quantifies over edges.

If $\varphi = \mathcal{Q}_V\, x.\, \varphi'$ then we construct an evaluation tree t_v for each $v \in V_H$ with $t_v = \mathit{evaltree}(H, \alpha[x \mapsto v], \varphi')$ and we let $\mathit{evaltree}(H, \alpha, \varphi) = (N, L, \mathit{quant}, \mathit{var}, \mathit{eval}, \delta, \mathit{root})$, with

$$\begin{array}{lcl} N & = & \biguplus_{v \in V_H} N_{t_v} \uplus \{n\}, \\ L & = & \biguplus_{v \in V_H} L_{t_v}, \\ \mathit{quant} & = & \biguplus_{v \in V_H} \mathit{quant}_{t_v} \uplus [n \mapsto \mathcal{Q}], \\ \mathit{var} & = & \biguplus_{v \in V_H} \mathit{var}_{t_v} \uplus [n \mapsto x], \\ \mathit{eval} & = & \biguplus_{v \in V_H} \mathit{eval}_{t_v}, \\ \delta & = & \biguplus_{v \in V_H} (\delta_{t_v} \uplus \{(n, v, \mathit{root}_{t_e})\}), \\ \mathit{root} & = & n \end{array}$$

We call the node n a *vertex node*, as the assigned variable quantifies over vertices.

Example 6.4: Evaluation tree

The property that each vertex of a graph has a successor vertex, connected through a next(n)-edge can be formalised by the FO-formula

$$\varphi_{suc} := \forall_V\, x.\ \exists_V\, y.\ \exists_E\, e.\ (\,\mathrm{lab}(e) = n \land \mathrm{att}(e) = x\ y\,).$$

Given the hypergraph $H \in HG_\Sigma$ from Figure 6.3 we recursively build up the evaluation tree $\mathit{evaltree}(H, \emptyset, \varphi_{suc})$, given in Figure 6.2. Leaves are depicted as rectangular boxes

labelled with their evaluation true *or* false*. Universal nodes are depicted as white cycles with a solid border* (x)*, while existential nodes are depicted as grey nodes with dashed borders* (x)*. The assigned variables are given as labels inside nodes. The transition relation is given as labelled, directed edges between nodes.*

In order to fulfil β, the matrix of φ, we have to set the variable e to e_1, as e_1 is the only edge in H with $\text{lab}(e_1) = n$. Further we have to set variable x to the vertex v_1 and y to v_2, as $\text{att}(e_1) = v_1\ v_2$. Under any other assignment β evaluates to false. Therefore the only true-labelled leaf is the third one, reachable from the root via $v_1\ v_2\ e_1$, while all other leaves are labelled by false.

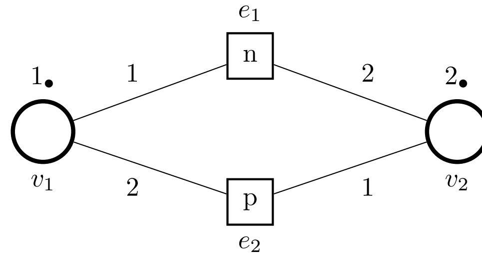


Figure 6.3: Hypergraph H not a model of φ_{suc}

Given an evaluation tree $t \in \mathit{Tree}$, the level of a node $n \in N_t$ is its distance from the root node. That is, the root node has level one ($level_t(root_t) = 1$), while for a node $n \in N_t$ at level k ($level_t(n) = k$) all successors $\{n' \in N_t \mid (n, _, n') \in \delta_t\}$ have the level $k+1$. Note that, due to the construction of an evaluation tree, all nodes at a given level share the same assigned variable. Actually given the FO-formula $\varphi = \mathcal{Q}_1 x_1. \ldots \mathcal{Q}_l x_l.\ \beta$ in prenex normal form, for any evaluation tree t for φ (independent from the given graph and variable interpretation) it holds that all nodes $n \in N_t$ at level $1 \leq k \leq l$ are assigned the variable x_k ($var_t(n) = x_k$). The leaves are all at level $l+1$. See Figure 6.4 for an example visualisation.

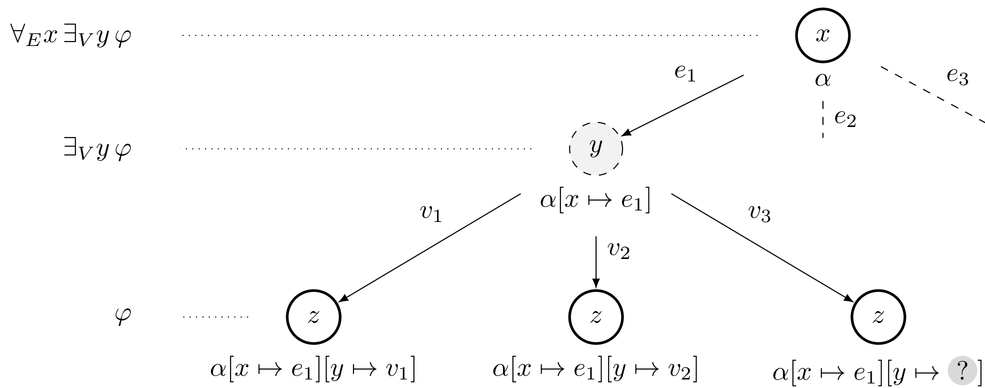


Figure 6.4: An example evaluation tree.

Given an evaluation tree $t = \mathit{evaltree}(H, \alpha, \varphi)$ for graph $H \in \mathrm{HG}_\Sigma$, a FO-formula in prenex normal form $\varphi = \mathcal{Q}_1 x_1. \dots \mathcal{Q}_l x_l.\ \beta$ and a variable interpretation α, the tree starting at some node $n \in N_t$ at level $k \leq l+1$ is an evaluation tree for the sub-formula $\varphi_{k+} = \mathcal{Q}_k x_k. \dots \mathcal{Q}_l x_l.\ \beta$. Thus each level of the tree corresponds to a sub-formula of φ. In Figure 6.4, the corresponding sub-formulae are indicated on the left of the tree. Given an evaluation tree $t \in \mathit{Tree}$ by $\mathit{succ}(t)$ we refer to the set of evaluation trees starting in the nodes at level two (the successors of the root node), while $\mathit{succ}(t, y)$ refers to the single tree that starts at the y-successor ($n \in N_t$ with $(\mathit{root}_t, y, n) \in \delta_t$). Note that by construction $\mathit{succ}(t, y) = \mathit{evaltree}(H, \alpha[x_i \mapsto y], \varphi_{2+})$.

Evaluation of an Evaluation Tree

Given $H \in \mathrm{HG}_\Sigma$, $\varphi \in \mathrm{FO}_\Sigma$, and variable interpretation α, we use the evaluation tree $t = \mathit{evaltree}(H, \alpha, \varphi)$ to determine if $(H, \alpha) \models \varphi$ holds. We define the evaluation function $\mathit{eval}\colon \mathit{Tree} \to \{\text{true}, \text{false}\}$ defined as follows:

$$\mathit{eval}(t) = \begin{cases} \text{true} & \textit{, if } \mathit{root}_t \in L_t \wedge \mathit{eval}_t(\mathit{root}_t) = \text{true}, \\ & \textit{ or } \mathit{root}_t \in N_t^{\exists} \textit{ and there exists } t' \in \mathit{succ}(t) \textit{ with } \mathit{eval}(t') = \text{true}, \\ & \textit{ or } \mathit{root}_t \in N_t^{\forall} \textit{ and for all } t' \in \mathit{succ}(t) \textit{ it holds that } \mathit{eval}(t') = \text{true}, \\ \text{false} & \textit{, otherwise.} \end{cases}$$

Lemma 6.2 (Evaluation trees):
Given $H \in HG_{T_\Sigma}$, $\varphi(X_V, X_E) \in FO_\Sigma$ in prenex normal form, and a variable interpretation $\alpha : X_V \to V_H \cup X_E \to E_H$, it holds that $(H, \alpha) \models \varphi$ if and only if the evaluation tree $t = \mathit{evaltree}(H, \alpha, \varphi)$ is evaluated to true, i.e. $\mathit{eval}(t) = \mathit{true}$.

Proof. We prove Lemma 6.2 by induction over the number $k \in \mathbb{N}_0$ of quantifiers in the prefix of the FO-formula $\varphi \in \mathrm{FO}_\Sigma$.

Induction Base $k = 0$:
If a formula $\varphi(X_V, X_E)$ has no quantifiers, i.e. is quantifier-free, the corresponding evaluation, by construction, only consists of a root node. The root node is a leaf l, with $\mathit{eval}_t(l) = \text{true}$ if $(H, \alpha) \models \varphi(X_V, X_E)$ and $\mathit{eval}_t(l) = \text{false}$ otherwise, thus the lemma is trivially fulfilled for any quantifier-free FO-formula.

Induction Hypothesis:
For any $H \in \mathrm{HG}_{T_\Sigma}$, $\alpha : (X_V \to V_H) \cup (X_E \to E_H^T)$, and $\varphi(X_V, X_E) \in \mathrm{FO}_\Sigma$ in prenex normal form with k quantifiers it holds that $(H, \alpha) \models \varphi$ if and only if for $t = \mathit{evaltree}(H, \alpha, \varphi)$ it holds that $\mathit{eval}(t) = \text{true}$.

Induction Step $k \to k+1$:
Let $\varphi(X_V, X_E) \in \mathrm{FO}_\Sigma$ in prenex normal form with $k+1$ quantifiers. Let $H \in \mathrm{HG}_{T_\Sigma}$,

and $\alpha : (X_V \to V_H) \cup (X_E \to E_H^T)$ and $t = \mathit{evaltree}(H, \alpha, \varphi(X_V, X_E))$. As $\varphi(X_V, X_E)$ is in prenex normal form with at least one quantifier it has to start with a quantifier. We distinguish four cases, as given by the Definition 6.1 of FO-formulae:

Case 1: $\varphi(X_V, X_E) = \exists_V\, x.\ \varphi'(X_V \uplus \{v\}, X_E)$

By definition of the $\models$-relation, $(H, \alpha) \models \exists_V\, x.\ \varphi'(X_V \uplus \{x\}, X_E)$ if and only if there exists some vertex $v_x \in V_H$ such that $(H, \alpha[x \mapsto v_x]) \models \varphi'(X_V \uplus \{x\}, X_E)$. Assume that there is such a vertex $v_x \in V_H$. The node root_t of t is an existential node. As *root* is a vertex node ($\mathit{var}(\mathit{root}_t) = x$), it has one successor for each vertex in V_H, each being the root of an evaluation tree. The successor reachable via the v_x-transition is the root of the evaluation tree $\mathit{succ}(t, v_x) = \mathit{evaltree}(H, \alpha[x \mapsto v_x], \varphi'(X_V \uplus \{x\}, X_E))$. Note that as φ has $k+1$ quantifiers, φ' has k quantifiers. By assumption $(H, \alpha[x \mapsto v_x]) \models \varphi'(X_V \uplus \{x\}, X_E)$ and therefore by the induction hypothesis $\mathit{eval}(\mathit{succ}(t, v_x)) = $ true. Thus one of the successors of the existential node root_t evaluates to true and therefore $\mathit{eval}(t)$ yields true.

If there is no vertex $v_x \in V_H$ with $(H, \alpha[x \mapsto v_x]) \models \varphi'(X_V \uplus \{x\}, X_E)$, then by induction hypothesis it follows that all successors are evaluated to false and therefore also $\mathit{eval}(t)$ returns false.

Case 2: $\varphi(X_V, X_E) = \forall_V\, x.\ \varphi'(X_V \uplus \{x\}, X_E)$

This case can be proven analogous to the first case. As in this case $(H, \alpha) \models \varphi(X_V, X_E)$ if and only if for all $v_x \in V_H$ it holds that $(H, \alpha[x \mapsto v_x]) \models \varphi(X_V \uplus \{x\}, X_E)$, all successors of t are evaluated to true. The node root_t is a universal node and thus $\mathit{eval}(t)$ yields true if and only if all its successors are true.

Case 3: $\varphi(X_V, X_E) = \exists_E\, x.\ \varphi'(X_V, X_E \uplus \{x\})$

This case can be proven analogous to the first case. Instead of selecting one of the vertices from V_H we select an edge from E_H^T. The node root_t is a edge node and its outgoing transitions are therefore labelled by edges from E_H^T. An $e_x \in E_H$ labelled transition leads from root_t to the root of the evaluation tree $\mathit{evaltree}(H, \alpha[x \mapsto e_x], \varphi'(X_V, X_E \uplus \{x\}))$, which by induction evaluates to true.

Case 4: $\varphi(X_V, X_E) = \forall_E\, x.\ \varphi'(X_V, X_E \uplus \{x\})$

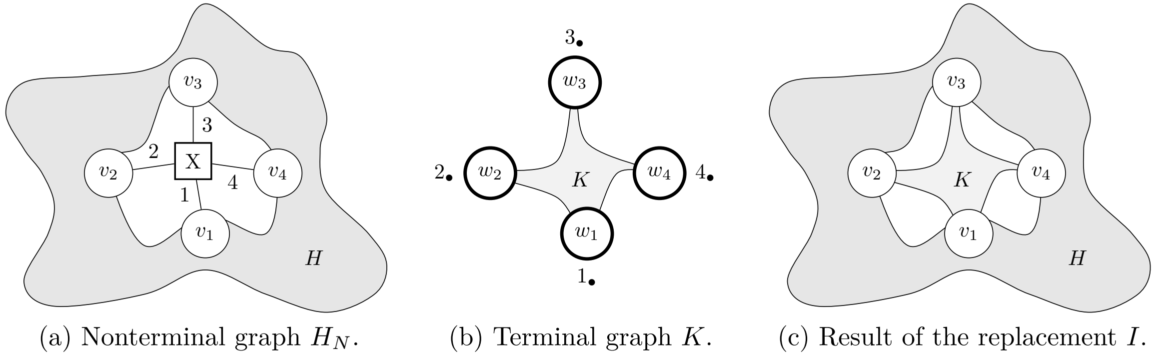
This case can be proven analogous to the second case, where as in the third case we consider the edges $x \in E_H$. Also the branching in the *root* of evaluation tree t is over the edges from E_H.

In any case $\mathit{evaltree}(H, \alpha, \varphi)$ is evaluated to true if and only if $(H, \alpha) \models \varphi$. □

In the following section we extend the evaluation approach such that it can be used to evaluate FO-formulae over HRGs also.

6.2.3 Evaluation Trees for Graphs in Context

Reconsider hyperedge replacements as defined in Section 2.3 (Definition 2.6 on page 19). Given $H, K \in \mathrm{HG}_\Sigma$ and nonterminal edge $e \in E_H^N$, with $|\mathrm{att}_H(e)| = |\mathrm{ext}_K|$, we replace edge e in H by graph K, yielding $I = H[K/e]$. A schematic representation is given in Figure 6.5. Figure 6.5(a) depicts graph H with X-labelled edge e. Terminal hypergraph K is depicted in (b). The number of external vertices of K is equal to the rank of the edge e. The result $I = H[K/e]$ is given in Figure 6.5(c). Note that I consists of the vertices and edges of H and K, excluding edge e and the external nodes of K that are merged with those nodes from H which are attached to the edge e. Any vertex $v \in V_I$ has its origin either in H or K, or in both if the vertex originated from merging vertices.



(a) Nonterminal graph H_N. (b) Terminal graph K. (c) Result of the replacement I.

Figure 6.5: Hyperedge replacement $I = H[K/e]$

Given $\varphi(X_V, X_E) \in \mathrm{FO}_\Sigma$ and $\alpha \in (X_V \to V_{H[K/e]}) \cup (X_E \to E^T_{H[K/e]})$ we want to construct $t = \mathit{evaltree}(H[K/e], \alpha, \varphi)$ in a compositional way from the evaluation trees for H and K. Note that the variable interpretation α maps variables to vertices and edges from H as well as from K. In order to construct the evaluation tree for H and K, we split the interpretation α into one for H and one for K. We consider the graphs to be embedded in an additional context, i.e. the context of H is graph K and vice versa. We map variables to a new symbol ? to represent references to the context of a graph.

Consider Figure 6.6 where a graph and its context are given. Inner and external vertices (e.g. v_1, v_3), as well as the edges (e.g. e_1) of the graph are visible and referenced by their names, while edges and vertices from the context are referenced by ?. The question mark states that the element is unknown. In general it is even not ensured that the referenced element exists. Considering this extension we obtain from α the following variable interpretations for H and K:

$$\alpha_H(x) = \begin{cases} \alpha(x) & \text{, } \textit{if } \alpha(x) \in V_H \cup E_H \\ ? & \text{, } \textit{otherwise} \end{cases}$$

$$\alpha_K(x) = \begin{cases} \alpha(x) & \text{, } \textit{if } \alpha(x) \in V_K \cup E_K \\ \mathrm{ext}_K[i] & \text{, } \textit{if } \alpha(x) = \mathrm{att}_H(e)[i] \\ ? & \text{, } \textit{otherwise} \end{cases}$$

That is, we obtain an extended variable interpretation where variables beside vertices and edges are also mapped to a placeholder. For an arbitrary graph $H \in \mathrm{HRG}_\Sigma$ and $\varphi(X_V, X_E) \in \mathrm{FO}_\Sigma$ we denote the set of extended variable interpretations by $\mathcal{A}_{\varphi,H} = (X_V \to V_H \cup \{?\}) \times (X_E \to E_H \cup \{?\})$.

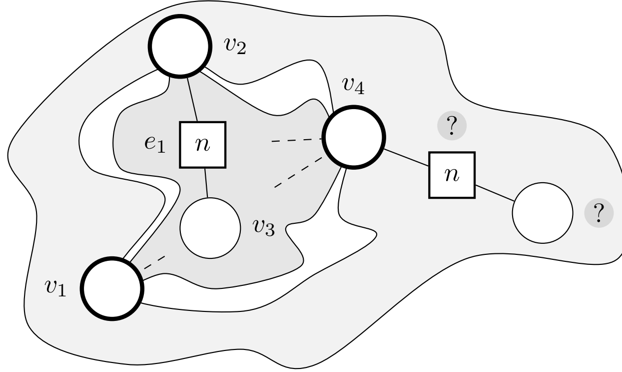


Figure 6.6: Hypergraph K and its outer context.

Locality of Predicates

Note that given an extended variable interpretation as α_H or α_K we cannot evaluate all FO-formula, as additional information about the context is needed. However, any predicate fulfilled in $H[K/e]$ is also fulfilled locally either in H or in K.

Lemma 6.3 (Locality of predicates):
Given $K, H \in HG_\Sigma$, and $e \in E_H^N$ with $|\mathrm{att}_H(e)| = |\mathrm{ext}_K|$, predicate $\pi(X_V, X_E) \in FO_\Sigma$ and $\alpha \in \mathcal{A}_{\varphi,H[K/e]}$ it holds that

$$(H[K/e], \alpha) \models \pi \iff (H, \alpha_H) \models \pi \quad \vee \quad (K, \alpha_K) \models \pi.$$

Proof. Let $I = H[K/e]$. We prove Lemma 6.3 for any type of predicate:

Predicate $\pi := e = f$:

By definition of $\models$ it holds that $(H, \alpha) \models \pi \Leftrightarrow \alpha(e) = \alpha(f)$. As $E_I^T = E_H^T \cup E_K^T$ it holds for any edge variable x that either $\alpha(x) = \alpha_H(x)$ or $\alpha(x) = \alpha_K(x)$. If $\alpha(e)$ and $\alpha(f)$ are both neither from E_H nor from E_K, then $\alpha(e) \neq \alpha(f)$ and thus $(I, \alpha) \not\models \pi$. It also holds that $(H, \alpha_H) \not\models \pi$ and $(K, \alpha_K) \not\models \pi$, as in both cases one of the edges is mapped to ? while the other is mapped to some edge of the corresponding graph. If both are from the same set E_H (or E_K) then it holds for $\alpha' = \alpha_H$ (or $\alpha' = \alpha_K$) that $\alpha(e) = \alpha'(e)$ and $\alpha(f) = \alpha'(f)$, and thus that $\alpha(e) = \alpha(f) \Leftrightarrow \alpha'(e) = \alpha'(f)$. From which directly follows that $(H, \alpha) \models \pi$ if and only if either $(H, \alpha_H) \models \pi$ or $(K, \alpha_K) \models \pi$.

Predicate $\pi := x = y$:

If $\alpha(x) = \alpha(y) \neq$?, then it holds that $\alpha(x), \alpha(y) \in V_H$ (or $\alpha(x), \alpha(y) \in V_K$). For $\alpha' = \alpha_H$ (or $\alpha' = \alpha_K$) it holds that $\alpha(x) = \alpha'(x)$ and $\alpha(y) = \alpha'(y)$, and thus that $\alpha(x) = \alpha(y) \Rightarrow \alpha'(x) = \alpha'(y)$. For $\alpha_H(x) = \alpha_H(y) \in V_H \subseteq V_I$ and $\alpha_K(x) = \alpha_K(y) \in V_K \subseteq V_I$ also the opposite direction holds. Note that in case $\alpha(x) = \alpha(y) \in [\text{att}(e)]$, the predicate is fulfilled in H as well as in K.

Predicate $\pi := \text{lab}(e) = a$:

Variable e is mapped to some edge $x \in E_H$. Note that $E_I = E_H \cup E_K$, as well as $\text{lab}_I = \text{lab}_H \cup \text{lab}_K$. It either holds that $x \in E_K$ and that $\text{lab}_I(x) = \text{lab}_K(x)$ or that $x \in E_H$ and that $\text{lab}_I(x) = \text{lab}_H(x)$. Thus either $(H, \alpha_H) \models \pi$ or $(K, \alpha_K) \models \pi$.

Predicate $\pi := \text{att}(x) = \bar{s}$:

For $\pi := \text{att}(x) = \bar{s}$, with edge variable x and $\bar{s}$ a sequence of vertex variables we have to prove that

$$\alpha(\bar{s}) \;=\; \text{att}_I(\alpha(x)) \quad \Longleftrightarrow \quad \alpha_H(\bar{s}) \;=\; \text{att}_H(\alpha_H(x)) \quad \vee \quad \alpha_K(\bar{s}) \;=\; \text{att}_K(\alpha_K(x)).$$

Recall that $E_I^T = E_H^T \uplus E_K^T$, thus it holds for $\alpha(x) = f$ that either $f \in E_H^T$ or $f \in E_K^T$. Reconsider the definition of att_I from Definition 2.6 for $f \in E_I^T$:

$$\text{att}_I(f)[i] = \begin{cases} \text{att}_H(e)[j] & \text{,}\textit{if } f \in E_K \;\wedge\; \text{att}_K(f)[i] = \text{ext}_K[j] \\ \text{att}_K(f)[i] & \text{,}\textit{if } f \in E_K \;\wedge\; \text{att}_K(f)[i] \notin [\text{ext}_K] \\ \text{att}_H(f)[i] & \text{,}\textit{otherwise } (f \in E_H) \end{cases}$$

If $f \in E_H$ then $\text{att}_I(f) = \text{att}_H(f)$ and thus $\alpha(\bar{s}) \subseteq V_H$:

$$\alpha_H(\bar{s}) \quad = \quad \alpha(\bar{s}) \quad = \quad \text{att}_I(\alpha(x)) \quad = \quad \text{att}_H(\alpha_H(x)).$$

If $f \in E_K$ then $\text{att}_I(f)$ is a sequence of inner vertices of K and vertices from H, which are attached to e, i.e. $[\text{att}_I(f)] \subseteq (V_K \setminus ext_K) \cup [\text{att}_H(e)]$. By definition of att_I it holds that $\text{att}_I(e)[i] = \text{att}_K(e)[i]$ whenever $\text{att}_I(e)[i] \in V_K$. As $\alpha(y) = \alpha_K(y)$, for all $\alpha(y) \in V_K$ it holds that

$$\alpha_K(\bar{s}[i]) \quad = \quad \alpha(\bar{s}[i]) \quad = \quad \text{att}_I(\alpha(x))[i] \quad = \quad \text{att}_K(\alpha_K(x))[i],$$

for any i with $\mathrm{att}_H(f)[i] \in V_K$. It remains to show that for i with $\alpha(\bar{s}[i]) \in V_{H_N}$ it holds that

$$\mathrm{att}_K(\alpha_K(x))[i] \;=\; \alpha_K(\bar{s}[i]) \quad\Longleftrightarrow\quad \mathrm{att}_I(\alpha(x))[i] \;=\; \alpha(\bar{s}[i]).$$

By definition of α_K it holds for any $1 \leq j \leq \mathrm{rk}(e)$, and vertex variable x_V that (1)

$$\alpha_K(x_V) = \mathrm{ext}_K[j] \quad\Longleftrightarrow\quad \alpha(x_V) = \mathrm{att}_H(e)[j],$$

further by definition of att_I it holds that (2)

$$\mathrm{att}_K(f)[i] = \mathrm{ext}_K[j] \quad\Longleftrightarrow\quad \mathrm{att}_I(f)[i] = \mathrm{att}_H(e)[j].$$

If we combine (1) and (2) we get

$$\mathrm{att}_K(f)[i] \;=\; \alpha_K(x_V) \quad\Longleftrightarrow\quad \mathrm{att}_I(f)[i] \;=\; \alpha(x_V).$$

Note that $f = \alpha(x) = \alpha_K(x)$, thus if we let $x_V = \bar{s}[i]$ the above combination gives us directly the remaining part of the proof. □

Extended Evaluation Trees

To realise *partial* evaluation trees for H and K we extend the evaluation trees from Definition 6.5 on page 135 by ?-transitions. Based on Lemma 6.3, we replace the evaluations in the leaves by one evaluation for each predicate occurring within the formula, i.e. a sequence over $\{\mathtt{t}, \mathtt{?}\}$, where `t` represents a predicate that is locally fulfilled, while `?` states that the predicate is not fulfilled locally but may be fulfilled within the context. This leads to extended evaluation trees defined as follows:

Definition 6.6 (Extended evaluation tree):
Let $H \in HG_\Sigma$, $\varphi(X_V, X_E) \in FO_\Sigma$ in prenex normal form and $\alpha \in \mathcal{A}_{\varphi,H}$. An evaluation tree *is a tuple $t = (N, L, quant, var, eval, \delta, root)$, with N, $quant$, var and $root$ as in Definition 6.5 and*

$eval : L \to \{\mathtt{t}, \mathtt{?}\}^\star$	*a predicate semi evaluation for the leaves.*
$\delta \in N \times (V_H \cup E_H \cup \{\,?\,\}) \times (N \cup L)$	*an extended transition relation.*

Given $H \in \mathrm{HG}_\Sigma$, $\varphi(X_V, X_E) \in \mathrm{FO}_\Sigma$ in prenex normal form and $\alpha \in \mathcal{A}_{\varphi,H}$ we construct the corresponding (extended) evaluation tree $evaltree(H, \alpha, \varphi)$ as follows:

If φ is quantifier-free we construct an evaluation tree that consists of a singe leaf and assign this leaf an evaluation sequence corresponding to the evaluations of the predicates, i.e. $evaltree(H, \alpha, \varphi) = (\emptyset, \{l\}, \emptyset, \emptyset, [l \mapsto p], \emptyset, l)$, where $p \in \{\mathtt{t}, \mathtt{?}\}^\star$ with $p[i] = \mathtt{t}$ if $(H, \alpha) \models \pi_i$ and `?` otherwise for all $\pi_i \in Pred\varphi$.

If $\varphi = \mathcal{Q}\, x.\, \varphi'$, depending on $\mathcal{Q}$ we consider any $y \in Y = (V_H \cup \{\,?\,\})$ if $\mathcal{Q}$ is $\exists_V$ or $\forall_V$, or $y \in Y = (E_H^T \cup \{\,?\,\})$ if $\mathcal{Q}$ is $\exists_E$ or $\forall_E$. For each y, we construct an evaluation tree $t_y = \mathit{evaltree}(H, \alpha[x \mapsto y], \varphi')$. Then $\mathit{evaltree}(H, \alpha, \varphi) = (N, L, \mathit{quant}, \mathit{var}, \mathit{eval}, \delta, \mathit{root})$, with

$$\begin{array}{lcl}
N & = & \biguplus_{y \in Y} N_{t_y} \uplus \{n\}, \\
L & = & \biguplus_{y \in Y} L_{t_y}, \\
\mathit{quant} & = & \biguplus_{y \in Y} q_{t_y} \uplus [n \mapsto \mathcal{Q}], \\
\mathit{var} & = & \biguplus_{y \in Y} v_{t_y} \uplus [n \mapsto x], \\
\mathit{eval} & = & \biguplus_{y \in Y} e_{t_y}, \\
\delta & = & \biguplus_{y \in Y} \left(\delta_{t_e} \uplus \{(n, y, \mathit{root}_{t_y})\} \right), \\
\mathit{root} & = & n
\end{array} .$$

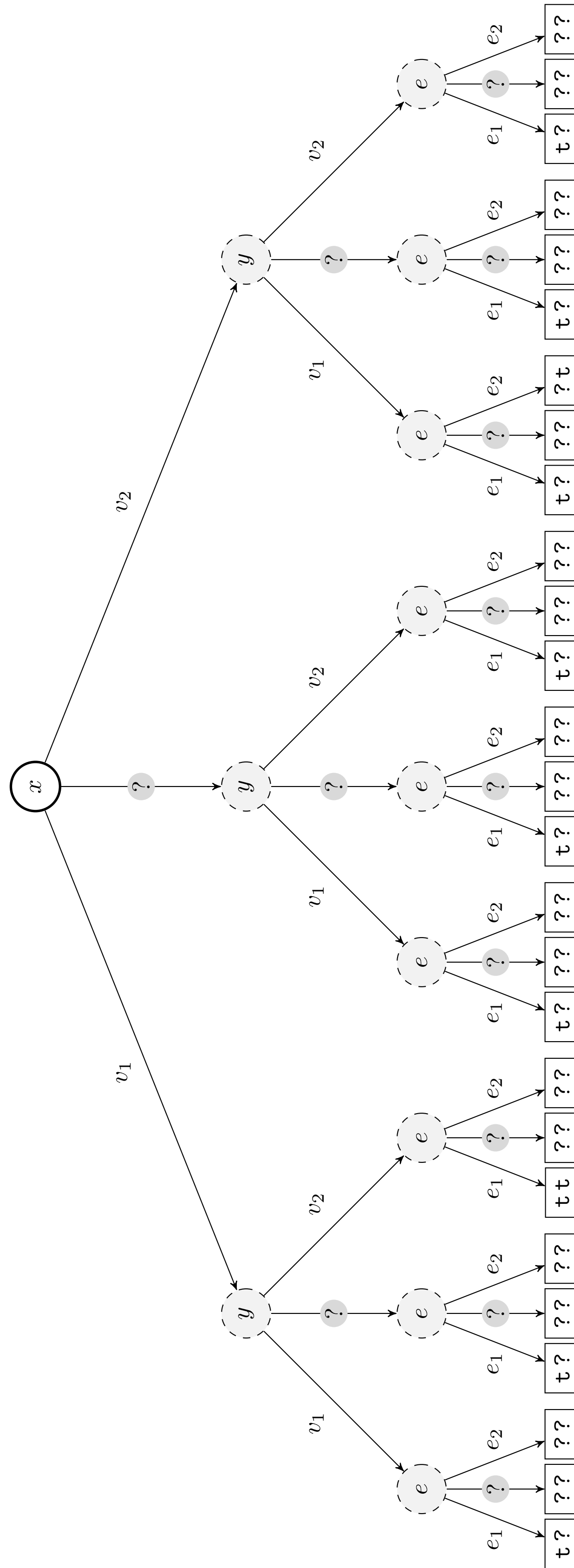


Figure 6.7: Extended evaluation tree for $H \models \varphi_{suc}$.

Example 6.5:
In Example 6.4, we consider for the graph H depicted in Figure 6.3 and FO-formula

$$\varphi_{suc} := \forall_V\, x.\ \exists_V\, y.\ \exists_E\, e.\ (\,\mathrm{lab}(e) = n \wedge \mathrm{att}(e) = xy\,),$$

the evaluation tree (without ?-extension) given in Figure 6.2 on page 135. Figure 6.7 on page 145 depicts the (extended) evaluation tree. The leafs hold the evaluations of the predicates $\pi_1 := \mathrm{lab}(e) = n$ and $\pi_2 := \mathrm{att}(e) = x\,y$.

*Consider the leaf reachable via $x \mapsto v_1$, $y \mapsto v_2$, $e \mapsto e_1$. As the edge e_1 is labelled by n, predicate π_1 is fulfilled. The edge e_1 attaches vertex v_1 at the first tentacle and v_2 at the second one, thus π_2 is also fulfilled. Therefore the corresponding leaf is labelled by [**t t**]. If we consider $x \mapsto v_2$, $y \mapsto v_1$, $e \mapsto e_2$, the second predicates π_2 becomes true, as $\mathrm{att}_H(e_2) = v_2\, v_1$, but as the label of e_2 is p the first predicate π_1 does not hold, thus we get the leaf [**? t**]. Whenever we select the edge e to be in the context, i.e. $e \mapsto$? we cannot determine its label. As v_1 and v_2 are external vertices we can neither preclude that they are attached to one or both of the tentacles of e if the latter is from the context. Therefore whenever e is mapped to ? we evaluate both predicates to* **?**.

In the following we will only consider extended evaluation trees as defined above. Therefore we refer to them just as evaluation tree.

6.2.4 Composition of Evaluation Trees

Given a hyperedge replacement $I = H[K/e]$, with $H, K \in \mathrm{HG}_\Sigma$, $e \in E_H$, $\varphi(X_V, X_E) \in \mathrm{FO}_\Sigma$ and $\Sigma \in \mathcal{A}_{\varphi,H[K/e]}$, our aim is to determine the evaluation tree $t = \mathit{evaltree}(H[K/e], \alpha, \varphi)$ as composition of the evaluation trees $t_H = \mathit{evaltree}(H, \alpha_H, \varphi)$ and $t_K = \mathit{evaltree}(K, \alpha_K, \varphi)$. The composition is realised via the merging operator $\bowtie$, that we will introduce shortly. Let us first have a look at the given trees and the idea behind the merging.

Note that t_H, t_K, as well as t are evaluation trees for the FO-formula φ. They thus have the same height as well as the formulae that are represented by the different levels of the tree. As a result two nodes from the same level are assigned the same variable as well as the same quantifier, no matter if they belong to the same or to different trees. That is t, t_H, t_K differ only in the number and labels of transitions as well as in the predicate evaluations in the leaves. Note that regarding the labels of the trees, transitions in t_H are labelled by vertices and edges from H ($V_H \cup E_H^T$), while transitions in t_K are labelled from $V_K \cup E_K^T$. The labels in t are from ($V_H \cup V_K \setminus [\mathrm{ext}_K]\ \cup\ E_H^T \cup E_K^T$). That is, the labels in t are a subset of the labels from t_H and t_K. The idea is to compose the evaluation tree t from t_H and t_K recursively. To get the target node of a transition, we merge the targets of the corresponding transitions in t_H and t_K. E.g. to get the successor for $e \in E_H$ in t we merge the corresponding successors of t_H and t_K. As $e \in E_H$ there is also a e-successor in t_H. In t_K there is no successor labelled by $e \in E_H$ but as e is part of

the context of K we choose the ?-successor. Correspondingly we merge the e-successor from t_H with the ?-successors from t_K to get the desired e-successor of t. Note that in order to get the successor for a vertex $v = \mathrm{att}_H(e)[i]$, we have to merge the v-successor from t_H with the corresponding $\mathrm{ext}_K[i]$-successors from t_K. There is exactly one such i, as we restricted nonterminal edges to be attached to pairwise distinct vertices. To reflect the last observation and to simplify the definition of the merging operator we relabel t_K to $t_K^r = (N_{t_K}, L_{t_K}, quant_{t_K}, var_{t_K}, eval_{t_K}, \delta', root_{t_K})$ with

$$\begin{array}{rcl} \delta' & = & \{(n, x, n') \in \delta_{t_T} \mid x \notin [ext_{H_T}]\} \\ & \cup & \{(n, \mathrm{att}_{H_N}(e)[i], n') \mid (n, \mathrm{ext}_{H_T}[i], n') \in \delta_{t_T}\}. \end{array}$$

Given a FO-formula φ we define $\bowtie$: $Tree_\varphi \times Tree_\varphi \rightarrow Tree_\varphi$, as the merging operator where the result of $t_H \bowtie_e t_K^r$ is as follows. If t_H consists of a single leaf ($N_{t_H} = \emptyset$), then φ is quantifier-free, and therefore also t_K consists of a single leaf that at the same time is the root of the corresponding tree. Following Lemma 6.3 we define the merging $t_H \bowtie t_K = (\emptyset, \{l\}, \emptyset, \emptyset, [l \mapsto p], \emptyset, l)$, where for all $1 \leq i \leq |Pred_\varphi|$ it holds that:

$$p[i] = \begin{cases} \mathtt{t} & \text{, if } e_{t_H}(root_{t_H})[i] = \mathtt{t} \ \vee \ e_{t_K}(root_{t_K})[i] = \mathtt{t} \\ \mathtt{?} & \text{, otherwise} \end{cases}$$

If t contains inner nodes, i.e. $N_t \neq \emptyset$, then this also holds for the tree t_K, as both are based on the same formula. As the set of inner nodes is not empty also the root node is an inner node. Note that all trees are based on φ, therefore all root nodes are assigned this quantifier and variable. We form the evaluation tree of a new root node, that we assign the same quantifier and variable and build recursively a evaluation tree for each successor label of the trees t_H and t_K. The set of successor labels of a tree t' is defined as $S(t') = \{x \mid \exists n \in N_{t'}. (root_{t'}, x, n) \in \delta_{t'}\}$. We let $S = S(t_H) \cup S(t_K)$. For each successor label $s \in S$ we construct the successor tree $t_s = succ_?(t_H, s) \bowtie succ_?(t_K, s)$, where $succ_?(t, s)$ returns $succ(t, s)$ if an s-successor exists and $succ(t, ?)$ otherwise. We get $t_H \bowtie t_K = (N, L, quant, var, eval, \delta, root)$ with

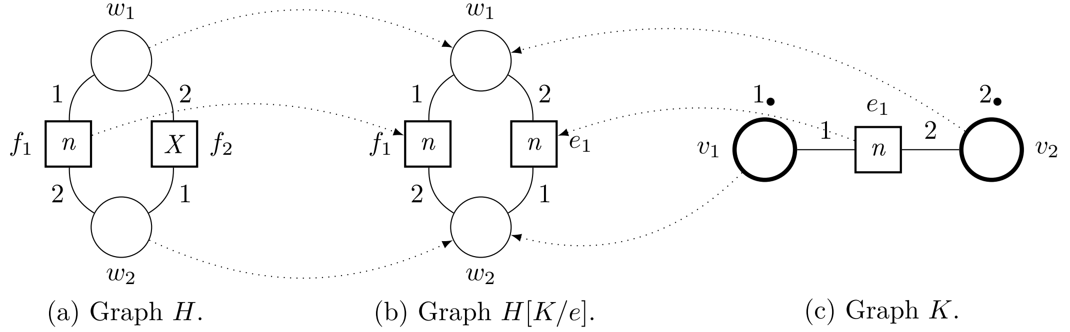
$$\begin{array}{lcl} N & = & \bigcup_{s \in S} N_{t_s} \cup \{n\}, \\ L & = & \bigcup_{s \in S} L_{t_s}, \\ quant & = & \bigcup_{s \in S} quant_{t_s} \cup [n \mapsto quant_t(root_t)], \\ var & = & \bigcup_{s \in S} var_{t_s} \cup [n \mapsto var_t(root_t)], \\ eval & = & \bigcup_{s \in S} eval_{t_s}, \\ \delta & = & \bigcup_{s \in S} (\delta_{t_s} \cup \{(n, s, root_{t_s})\}), \\ root & = & n. \end{array}$$

Example 6.6: Merging of trees

Reconsider the formula φ_{suc} from Example 6.4, describing the property that any vertex has a n-successor:

$$\varphi_{suc} := \forall_V\, x.\ \exists_V\, y.\ \exists_E\, e.\ (\,\mathrm{lab}(e) = n \wedge \mathrm{att}(e) = xy\,).$$

Consider the graphs from Figure 6.8. Replacing edge f_2 in H (a) by K (c) we get $H[K/f_2]$ (b). The dotted lines between the graphs represent the correspondence between vertices and edges of H, K and $H[K/f_2]$. The evaluation trees $t_H = evaltree(H, \emptyset, \varphi_{suc})$ and $t_K = evaltree(K, \emptyset, \varphi_{suc})$ can be found in Figure 6.9. The relabelling for tree t_K^r is given in parentheses in Figure 6.9(b). The result of the composition, the evaluation tree $t = t_H \bowtie t_K^r$ is given in Figure 6.10.



(a) Graph H. (b) Graph $H[K/e]$. (c) Graph K.

Figure 6.8: An exemplary replacement.

Theorem 6.3 (Composition of evaluation trees):
Given $H, K \in HG_\Sigma$, $e \in E_H^N$ and a sentence $\varphi \in FO_\Sigma$ in prenex normal form, for $t_H = evaltree(H, \emptyset, \varphi)$ and $t_K = evaltree(K, \emptyset, \varphi)$ it holds that

$$t_H \bowtie t_K^r = evaltree(H[K/e], \emptyset, \varphi)$$

Proof. The height of an evaluation tree, as well as the variables assigned to the levels of the tree, is uniquely determined by the corresponding FO-formula. Therefore height and variable interpretations are the same for the evaluation trees t_H and t_K. Both trees differ only in the labels of the branching.

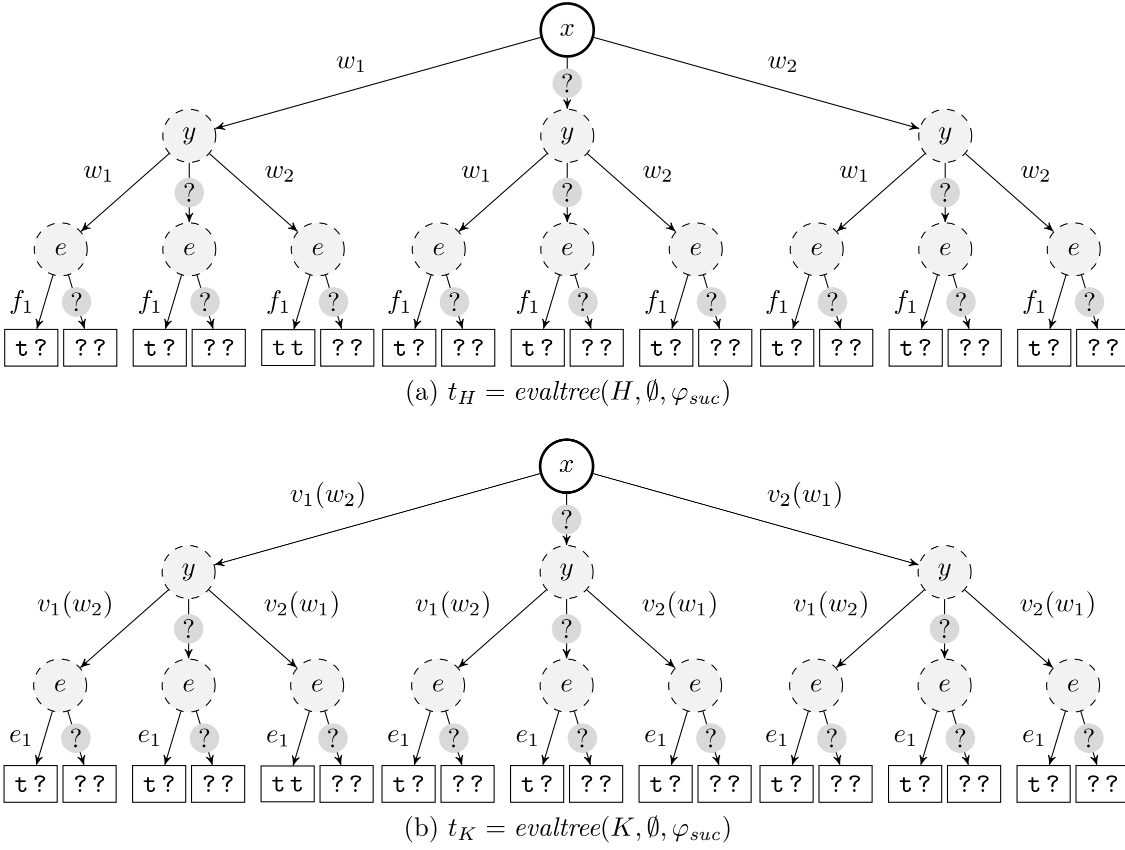
We prove by induction over the number $k \in \mathbb{N}$ of quantifiers in φ, that

$$evaltree(H[K/e], \alpha, \varphi) = evaltree(H, \alpha_H, \varphi) \bowtie evaltree(K, \alpha_K, \varphi)^r,$$

with α_H and α_K defined as on page 141. This directly proves Theorem 6.3, as there we have the special case that $\alpha = \emptyset$, such that $\alpha_H = \alpha_K = \emptyset$.

Induction Base $k = 0$:
If $k = 0$ then φ is quantifier-free and any evaluation tree consists of a single leaf. For $evaltree(H[K/e], \alpha, \varphi) = (\emptyset, \{l\}, \emptyset, \emptyset, [l \mapsto p], \emptyset, l)$ it holds that for $1 \leq i \leq |p|$

$$p[i] = \mathtt{t} \quad \Leftrightarrow \quad (H_N[H_T/e], \alpha) \models \pi_i.$$



(a) $t_H = evaltree(H, \emptyset, \varphi_{suc})$

(b) $t_K = evaltree(K, \emptyset, \varphi_{suc})$

Figure 6.9: Evaluation trees for formula φ_{suc}

Given by Lemma 6.3 if $p[i]$ is `t`, then either $(H, \alpha_H) \models \pi_i$ or $(K, \alpha_K) \models \pi_i$. By the definition of merging for leaves it follows that for the resulting leaf the ith entry is `t`. If $p[i]$ is `?`, then none of the evaluations yields `t` and the merging result is `?`. Thus the merging results in a tree consisting of a single leaf that is equal to the one we get from the direct construction of the evaluation tree, i.e. for a quantifier free formulae φ it holds that

$$t_H \bowtie t_K^r = evaltree(H[K/e], \alpha, \varphi).$$

Induction Hypothesis:

Given $\varphi \in \mathrm{FO}_\Sigma$ with k quantifiers, $H, K \in \mathrm{HG}_\Sigma$, and $\alpha \in \mathcal{A}_{\varphi,H[K/e]}$ it holds for $t_H = evaltree(H, \alpha_H, \varphi)$ and $t_K = evaltree(K, \alpha_K, \varphi)$ that

$$t_H \bowtie t_K^r = evaltree(H[K/e], \alpha, \varphi).$$

Induction Step $k \to k+1$:

Let φ be a formula wit $k+1$ quantifiers with $k \geq 0$, i.e. $\varphi = \mathcal{Q}x.\, \varphi'$ where φ' is an FO-formula with k quantifiers. We distinguish between $\mathcal{Q}$ being an edge quantifier ($\exists_E$ or $\forall_E$) and $\mathcal{Q}$ being a vertex quantifier ($\exists_V$ or $\forall_V$).

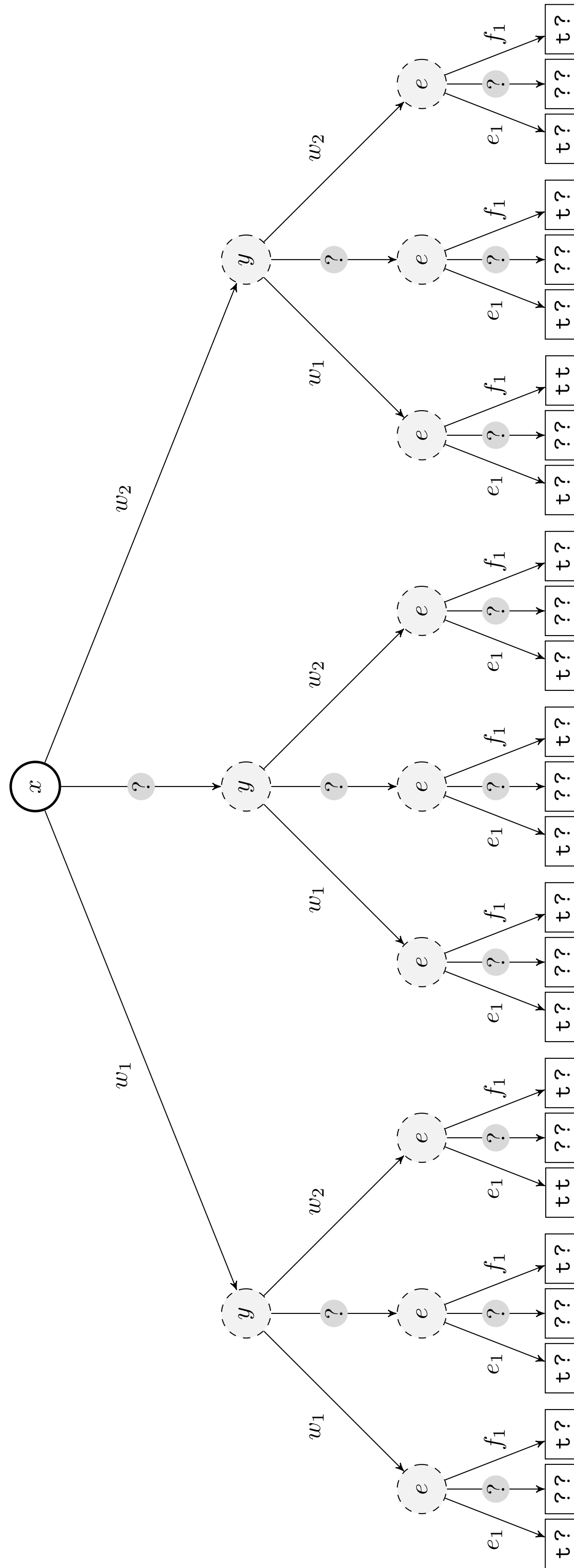


Figure 6.10: Merging result for $t_H \bowtie t^r_K$.

If $\mathcal{Q}$ is an edge quantifier, then the evaluation tree for $H[K/e]$ has one successor for each edge $f \in E_H \cup E_K$, while t_H has a successor for each edge in E_H and t_K a successor for each edge in E_K. Note that the relabelled evaluation tree for t_K^r has the same successors as we only relabel vertex transitions. The merging yields an evaluation tree with one successor for each transition label, i.e. for any $f \in E_H \cup E_K$. It remains to prove that the successors are the same. For $f \in E_H$, the f-successor of the evaluation tree for $H[K/e]$ is the tree $\mathit{evaltree}(H[K/e], \alpha[x \mapsto f], \varphi')$. The f-successor in t_H is the tree $\mathit{evaltree}(H, \alpha_H[x \mapsto f], \varphi')$. In t_K there is no such successor therefore we will use the ?-successor for the merging. The ? successor is $\mathit{evaltree}(K, \alpha_K[x \mapsto ?], \varphi')$ Note that $\alpha[x \mapsto f]_H = \alpha_H[x \mapsto f]$ and that $\alpha[x \mapsto f]_K = \alpha_K[x \mapsto ?]$. Thus the f-successor of the tree $t_H \bowtie t_K$ is

$$\mathit{evaltree}(H, \alpha[x \mapsto f]_H, \varphi') \bowtie \mathit{evaltree}(K, \alpha[x \mapsto f]_K, \varphi')^r,$$

which by induction hypothesis is $\mathit{evaltree}(H[K/e], \alpha[x \mapsto f], \varphi')$. For $f \in E_K$ we get the proof by interchanging the mappings $x \mapsto f$ and $x \mapsto ?$.

If $\mathcal{Q}$ is a vertex quantifier, then the successor labels in the evaluation trees are the vertices from $V_{H[K/e]}$, V_H and V_K. Note that we use the relabelled tree t_K^r for which the vertex labels are from $V_K \setminus [\mathrm{ext}_K] \cup [\mathrm{att}_H(e)]$, where $[\mathrm{att}_H(e)] \subseteq V_H$, thus within the result of the merge the successor labels are from $V_H \cup V_K \setminus [\mathrm{ext}_K] = V_{H[K/e]}$. Therefore the successor labels are the same as the ones in $\mathit{evaltree}(H[K/e], \alpha, \varphi)$. It remains to prove that the successors are the same. In the merging result we get for each $v \in V_{H[K/e]}$ one successor. If $v \in V_H \setminus [\mathrm{att}_H(e)]$ or $v \in V_K \setminus [\mathrm{ext}_K]$, then in analogy to the edge quantifier we merge v-successors with ?-successors yielding $\mathit{evaltree}(H[K/e], \alpha[x \mapsto v], \varphi')$ by induction. If $v \in [\mathrm{ext}_H(e)]$, then we get on the left hand side of the merge operator $\mathit{succ}(\mathit{evaltree}(H, \alpha_H, \varphi'), v) = \mathit{evaltree}(H, \alpha[x \mapsto v]_H, \varphi')$. As in t_K^r transitions for external vertices of K are relabelled with vertices from $\mathrm{att}_H(e)$, the root of t_K^r has also a v-successor. More precisely for $i \in \mathbb{N}$ with $v = \mathrm{att}_H(e)[i]$ we relabel the $\mathrm{ext}_K[i]$-transitions to a v-transition, i.e. we get on the right of the merge operator the successor $\mathit{evaltree}(K, \alpha_K[x \mapsto \mathrm{ext}_K[i]], \varphi')^r$ and as $\alpha_K[x \mapsto \mathrm{ext}_K[i]] = \alpha[x \mapsto \mathrm{att}_H(e)[i]]_K$ we get

$$\mathit{evaltree}(H, \alpha[x \mapsto v]_H, \varphi') \bowtie \mathit{evaltree}(K, \alpha[x \mapsto v]_K, \varphi')^r,$$

which by induction hypothesis is $\mathit{evaltree}(H[K/e], \alpha[x \mapsto v], \varphi')$. That is the successors are the same. □

Note that while we restricted Theorem 6.3 to sentences, we proved the statement for general formulae, thus we could extend the theorem in this sense.

6.2.5 Evaluation

In Section 6.2.2 we used evaluation trees for terminal graphs to evaluate the model relation. We can simply extend the method *eval* to realise evaluations of the model

relation based on extended evaluation trees, assuming that there is no context that has to be considered. We evaluate evaluation tree $t \in \mathit{Tree}$ as follows.

If t is a leaf, i.e. $t = (\emptyset, \{l\}, \emptyset, \emptyset, [l \mapsto p], \emptyset, l)$, we consider any predicate evaluation $p[i]$, that is not $\mathtt{t}$ to be false ($\mathtt{f}$), that is reasonable as there is no further context that could make the predicate become true. Based on the evaluations in p we then can determine a evaluation of the leaf. In β_φ, the matrix of φ, we substitute any predicate $\varphi_i \in \mathit{Pred}\varphi$ by true if $p[i] = \mathtt{t}$ and by false otherwise. The resulting formula is propositional and can be evaluated directly.

$$\mathit{eval}(t) = \beta_\varphi[(p[1] = \mathtt{t})/\pi_1] \ldots [(p[k] = \mathtt{t})/\pi_k], \qquad \mathit{with}\ k = |\mathit{Pred}\varphi|$$

If t is a tree of height at least one, i.e. $N_t \neq \emptyset$, we evaluate the tree according to the semantics of FO, where we omit ? successors, as these are blind references to the context, that does not exist.

$$\mathit{eval}(t) = \begin{cases} \text{true} & , \mathit{if}\ \mathit{root}_t \in N_t^\exists \ \wedge\ \exists t' \in \mathit{succ}(t) \setminus \{\mathit{succ}(t, ?)\}.\ \mathit{eval}(t') = \text{true}, \\ & \ \mathit{or}\ \mathit{root}_t \in N_t^\forall \ \wedge\ \forall t' \in \mathit{succ}(t) \setminus \{\mathit{succ}(t, ?)\}.\ \mathit{eval}(t') = \text{true}, \\ \text{false} & , \mathit{otherwise}. \end{cases}$$

Lemma 6.2 as well as its proof can easily be adapted to extended evaluation trees and *eval* as defined above.

6.2.6 Minimisation

To some sense, evaluation trees are comparable to binary decision diagrams. As the latter they can also be minimised by merging equivalent nodes. Given an evaluation tree $t \in \mathit{Tree}$ we call two nodes equivalent if they evaluate to the same value (true, false) under any possible context. Equivalence is defined recursively as follows.

Definition 6.7 (Equivalence of nodes):
Given $t_1, t_2 \in \mathit{Tree}$, nodes $n_1 \in N_{t_1} \cup L_{t_1}$ and $n_2 \in N_{t_2} \cup L_{t_2}$ are equivalent*, denoted $n_1 \approx n_2$, if one of the following two conditions holds:*

1. *both are leaves $n_1 \in L_{t_1}$ and $n_2 \in L_{t_2}$ and their predicate evaluations are equivalent*

$$e_{t_1}(n_1) = e_{t_2}(n_2),$$

2. *or both are inner nodes $n_1 \in N_{t_1}$ and $n_2 \in N_{t_2}$ and their successors are equivalent*

$$\begin{aligned} & \forall (n_1, x, n_1') \in \delta_{t_1}.\exists (n_2, x, n_2') \in \delta_{t_2}\ \mathit{with}\ n_1' \approx n_2' \\ \wedge\ & \forall (n_2, x, n_2') \in \delta_{t_2}.\exists (n_1, x, n_1') \in \delta_{t_1}\ \mathit{with}\ n_1' \approx n_2'. \end{aligned}$$

We call $t_1, t_2 \in \mathit{Tree}$ equivalent *if their root nodes are equivalent, i.e. $\mathit{root}_{t_1} \approx \mathit{root}_{t_2}$.*

We call an evaluation tree t minimal if all nodes are pairwise non equivalent, i.e. for all $n_1, n_2 \in N_t \cup L_t$ it holds that $n_1 \approx n_2$ if and only if $n_1 = n_2$. Given an evaluation tree t we incrementally construct an equivalent minimal one, using a merging function $\mathit{merge} : \mathit{Tree} \times (N \cup L) \times (N \cup L) \to \mathit{Tree}$.

Starting at the leaves we pairwise merge leaves $l_1, l_2 \in L_t$ with $e(l_1) = e(l_2)$ until all leaves are pairwise non-equivalent. The merging of leaves is defined as follows

$$\begin{array}{rcl} \mathit{merge}(t, l_1, l_2) & = & (N_t, L_t \setminus \{l_2\}, \mathit{quant}_t, \mathit{var}_t, \mathit{eval}_t{\restriction}(L_t \setminus \{l_2\}), \delta', \mathit{root}_t) \\ \mathit{with}\ \delta' & = & \{(n, x, n') \in \delta_t \mid n' \neq l_2\} \cup \{(n, x, l_1) \mid (n, x, l_2) \in \delta_t\} \end{array}$$

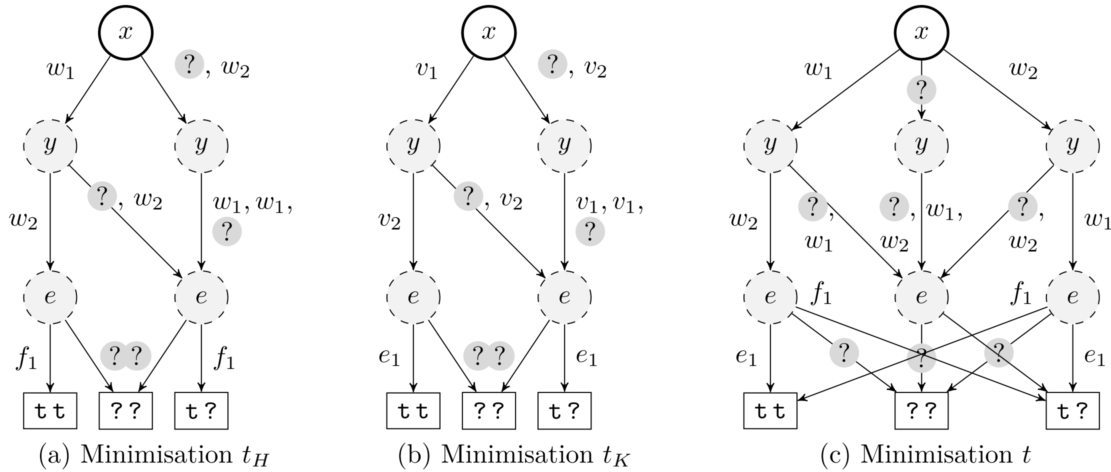
Thus we remove leaf l_2 from the tree and adapt transitions with target l_2 to target the leaf l_1.

Afterwards we continue with the nodes at level $\mathit{level}(l_1) - 1$. Each pair of nodes n_1, n_2 with equal successors, i.e. $\{(x, n) \mid (n_1, x, n) \in \delta_t\} = \{(x, n) \mid (n_2, x, n) \in \delta_t\}$ is merged. Merging of nodes is realised analogue to the merging of leaves by removing n_2 and retargeting the n_2 transitions to n_1.

$$\begin{array}{rcl} \mathit{merge}(t, n_1, n_2) & = & (N_t \setminus \{n_2\}, L_t, \mathit{quant}_t{\restriction}(L_t \setminus \{n_2\}), \mathit{var}_t{\restriction}(L_t \setminus \{n_2\}), \mathit{eval}_t, \delta', \mathit{root}_t) \\ \mathit{with}\ \delta' & = & \{(n, x, n') \in \delta_t \mid n' \neq n_2\} \cup \{(n, x, n_1) \mid (n, x, n_2) \in \delta_t\} \end{array}$$

Example 6.7: Minimisation of evaluation trees

Reconsider the evaluation trees from Example 6.6. The evaluation tree for K is depicted in Figure 6.7 on page 145. Its minimisation is given in Figure 6.11(b). In (a) the minimisation of the evaluation tree for H is given and in (c) the minimisation of the evaluation tree for $H[K/e]$. The original evaluation trees (before minimisation) can be found in Figure 6.9(a) (for H) and Figure 6.10 (for $H[K/e]$).



(a) Minimisation t_H (b) Minimisation t_K (c) Minimisation t

Figure 6.11: Minimised evaluation trees

Note that during minimisation only leaves with equivalent predicate evaluations are merged as well as nodes that (afterwards) share the same successors. Therefore the paths from the root to the leaves are preserved. As construction, evaluation and merging of evaluation trees does not depend on node and leaf identities, but on successors of nodes and evaluations of leaves, it is clear that equivalent trees behave equally under these operations. That is the result of an evaluation of trees is the same as for their minimisation. In the following we do not distinguish between equivalent trees and we always consider and depict minimal trees.

6.2.7 Pre-Evaluation

In Section 6.2.5, we evaluated any predicate not evaluated to `t` to false (`f`). This is reasonable as we considered the corresponding graph not to have any context. In general we can evaluate predicates to `f`, whenever any necessary information is given. That is, we adopt a three-valued interpretation of predicates.

Let $\pi(X_V, X_E) \in \mathrm{FO}_\Sigma$ be a predicate, $H \in \mathrm{HG}_\Sigma$, and $\alpha \in \mathcal{A}_{\pi,H}$. In the following we define the function $eval((H,\alpha) \models \pi)$, that given a three-valued interpretation α evaluates π over H.

$(H,\alpha) \models e = f$, *with* $\alpha(e), \alpha(f) \in E_H \cup \{?\}$

If both edge variables are mapped to edges from E_H, then we simply check for equality of the assigned edges. If one of the edge variables is mapped to an edge of the context, while the other is mapped to one of the graph, they can not be equal. Only if both variables are mapped to edges from the context, we do not know if both are mapped to the same edge (from the context).

$$eval((H,\alpha) \models e = f) = \begin{cases} \mathtt{t} & , \textit{if } \alpha(e) = \alpha(f) \neq ? \\ \mathtt{?} & , \textit{if } \alpha(e) = \alpha(f) = ? \\ \mathtt{f} & , \textit{otherwise} \end{cases}$$

$(H,\alpha) \models x = y$, *with* $\alpha(x), \alpha(y) \in V_H \cup \{?\}$

For vertices the three-valued evaluation of the equality predicates works analogously.

$$eval((H,\alpha) \models x = y) = \begin{cases} \mathtt{t} & , \textit{if } \alpha(x) = \alpha(y) \neq ? \\ \mathtt{?} & , \textit{if } \alpha(x) = \alpha(y) = ? \\ \mathtt{f} & , \textit{otherwise} \end{cases}$$

$(H,\alpha) \models \mathrm{lab}(e) = a$, *with* $\alpha(e) \in E_H \cup \{?\}$, *and* $a \in T_\Sigma$

The label of an edge is always defined in the graph part which contains the edge. Thus if α maps the edge variable to some edge from H we can determine if it is labelled by a.

Otherwise if the edge is within the context ($?$), the relation has to be checked within the context.

$$eval((H,\alpha) \models \text{lab}(e) = a) = \begin{cases} \texttt{t} & , \text{if } \alpha(e) \neq ? \quad \wedge \quad \text{lab}(\alpha(e)) = a \\ \texttt{f} & , \text{if } \alpha(e) \neq ? \quad \wedge \quad \text{lab}(\alpha(e)) \neq a \\ \texttt{?} & , \text{otherwise} \end{cases}$$

$(H,\alpha) \models \text{att}(e) = \bar{s}$, *with* $e \in E_H \cup \{?\}$, *and* $\bar{s} \in (V_H \cup \{?\})^\star$

The attached vertices of an edge are always part of the same graph part, as otherwise we would have dangling pointers. However, if the edge is in the context it could be connected to context vertices, i.e. to vertices that are external or attached to an nonterminal edge.

$$eval((H,\alpha) \models \text{att}(e) = \bar{s}) = \begin{cases} \texttt{t} & , \text{if } \alpha(e) \neq ? \quad \wedge \quad \text{att}(\alpha(e)) = \alpha(\bar{s}) \\ \texttt{?} & , \text{if } \alpha(e) = ? \quad \wedge \quad [\alpha(\bar{s})] \subseteq [\text{ext}_H] \cup \text{att}(E_H^N) \cup \{?\} \\ \texttt{f} & , \text{otherwise} \end{cases}$$

We extend the predicate semi-evaluation for leaves from Definition 6.6 on page 143 to a three-valued evaluation, i.e. $eval : L \to \{\texttt{t}, \texttt{?}, \texttt{f}\}^\star$. We use the three-valued predicate evaluation as defined above to determine the predicates evaluations. That is, for a quantifier-free formula φ we get $evaltree(H, \alpha, \varphi) = (\emptyset, \{l\}, \emptyset, \emptyset, [l \mapsto p], \emptyset, l)$, where $p \in \{\texttt{t}, \texttt{?}, \texttt{f}\}^\star$ with $p[i] = eval((H,\alpha) \models \pi_i)$ for all $\pi_i \in Pred\varphi$. Note that this potentially leads to more nodes. However, the additional information within the leaves allows us to pre-evaluate parts of the tree.

Three-Valued Pre-Evaluation

The partial information in the leaves, given as three-valued evaluations of the predicates in φ can be used to give a three-valued interpretation to the matrix β_φ of φ. Remember that we evaluate leaves by evaluating the matrix of the formula using the predicate evaluations instead of the predicates. Whenever all evaluations are determined, i.e. none of the predicates evaluates to maybe (`?`), we can determine a true or false value. In some cases this is possible even if not all predicate evaluations are known. This is the case when maybe-evaluated predicates have no further influence on the evaluation result.

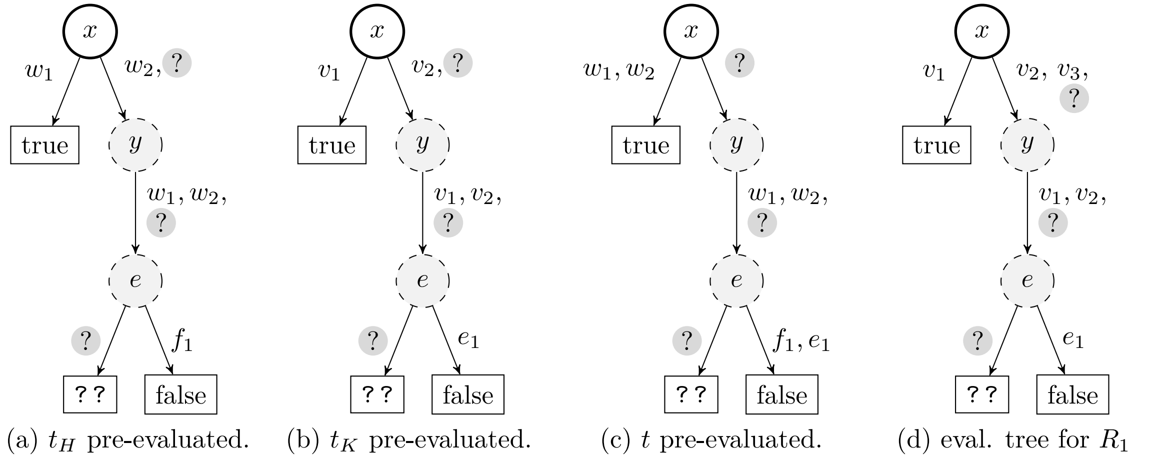
Following this observation we perform a three-valued pre-evaluation of the leaves and whenever the result is true or false we replace the leaf by a special leaf representing this evaluation result. In doing so we get an evaluation tree with leaf evaluation function $e : L \to \{\texttt{t}, \texttt{f}, \texttt{?}\}^\star \cup \{\text{true}, \text{false}\}$.

Given an evaluation tree t we evaluate an existential node to true, if at least one successor evaluates to true, while we evaluate an universal node to false if at least one of its successors evaluates to false. Therefore, if n is an existential (universal) node and the

true-leaf (false-leaf) is one of its successors, then we pre-evaluate the node to true (false), i.e. we replace the node by the true-leaf (false-leaf).

Example 6.8:

Reconsider the minimised evaluation trees from Example 6.7 on page 153. These are trees for the formula φ_{suc} given in Example 6.6 on page 147. The matrix of φ_{suc} is $\beta = \pi_1 \wedge \pi_2$, i.e. β is only fulfilled if both predicates are fulfilled. Note that, if we use three-valued evaluations, the right-most leaves are evaluated to $\mathbf{t}\,\mathbf{f}$, as the edge is known (and labelled by n), but is not attached to the given vertices. Therefore we evaluate this leaf to false, while we evaluate the $\mathbf{t}\,\mathbf{t}$ leaf to true. In Figure 6.12(a)-(c), the pre-evaluated versions of the evaluation trees from Example 6.7 are given.



(a) t_H pre-evaluated. (b) t_K pre-evaluated. (c) t pre-evaluated. (d) eval. tree for R_1

Figure 6.12: Pre-evaluated minimal evaluation trees

The pre-evaluation does not change the semantics of a tree, as redundant information is just condensed. The necessary adaptions to the evaluation of trees are obvious. If we merge a tree with a true-leaf (false-leaf) the result is always again the true-leaf (false-leaf). Due to readability we only depict pre-evaluated trees from now on.

6.2.8 Evaluation for HRG-Languages

Given $G \in \mathrm{HRG}_\Sigma$, $H \in \mathrm{HG}_\Sigma$ and sentence $\varphi \in \mathrm{FO}_\Sigma$ we aim to determine the set $\{\mathit{evaltree}(K, \emptyset, \varphi) \mid K \in L_G(H)\}$ of evaluation trees for the graphs derivable from H. Reconsider Corollary 2.1 on page 27, where we noticed that any graph in $L_G(H)$ can be derived by replacing the nonterminal edges in H by terminal graphs derivable from the corresponding nonterminals.

$$L_G(H) = \{H[e_1/K_1, \ldots e_n/K_n] \mid K_i \in L_G(\mathrm{lab}_H(e_i)^\bullet)\ \forall 1 \leq i \leq n\}$$

In Section 6.2.4 we presented the merging operator $\bowtie$ that allows us to determine the evaluation tree for a replacement based on the evaluation trees of the involved graphs. The latter together with Corollary 2.1 gives us a possibility to determine the set of evaluation trees for the graphs derivable from H based on the sets of evaluation trees for the nonterminals from Σ.

Note that whenever the sets of derivation trees for nonterminals are not finite, neither is the set for H. In general the set of evaluation trees is not finite, as the languages are not finite.

Example 6.9:
Reconsider grammar G for linked lists given in Figure 6.13, introduced in Section 3.3.

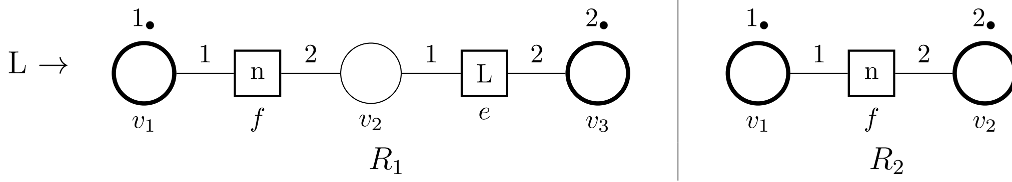


Figure 6.13: Grammar for singly linked lists.

Considering once more the FO-formula φ_{suc}, we are interested in the set of evaluation trees for L, i.e. the set $T_{L^\bullet} = \{evaltree(H, \emptyset, \varphi_{suc}) \mid H \in L(L^\bullet)\}$. As $R_2 \in L(L^\bullet)$ is a terminal graph we build up the evaluation tree as described in Section 6.2.2. The resulting evaluation tree is given in Figure 6.14(a). The evaluation tree for $R_1[R_2/e] \in L(L^\bullet)$ is obtained by merging the evaluation tree for R_1 (Figure 6.12(d)) with the one for R_2 (Figure 6.14(b)). Merging the evaluation tree for R_1 with the one for $R_1[R_2/e]$ yields the evaluation tree for $R_1[R_1[R_2/e]/e']$, and so on. The resulting minimised evaluation trees are given in Figure 6.14. Vertices and edges are renamed if necessary (here $v \mapsto v'$).

Anonymised Evaluation Trees

Note that the resulting trees in Example 6.9 share the same global shape and only differ in the number of transitions between particular nodes. When it comes to the evaluation of an evaluation tree we distinguish between ?-transition and non-?-transitions only, thus in that case the quantity of non-?-transitions between two nodes does not matter. If we determine the evaluation tree for $H[K/e]$ via merging, transitions from one tree that are labelled by vertices not shared by the context are merged with the ?-transition of the other. For K the external vertices are the ones that are shared with the context. That is, we do not distinguish between transitions labelled with inner vertices $V_K \setminus [\text{ext}_K]$, which are all merged with the ?-transition of the merging counterpart. That is, for none of the operations the identity of inner vertices of K matters. Therefore we anonymise them by relabelling transitions labelled by inner vertices with a new label !. The

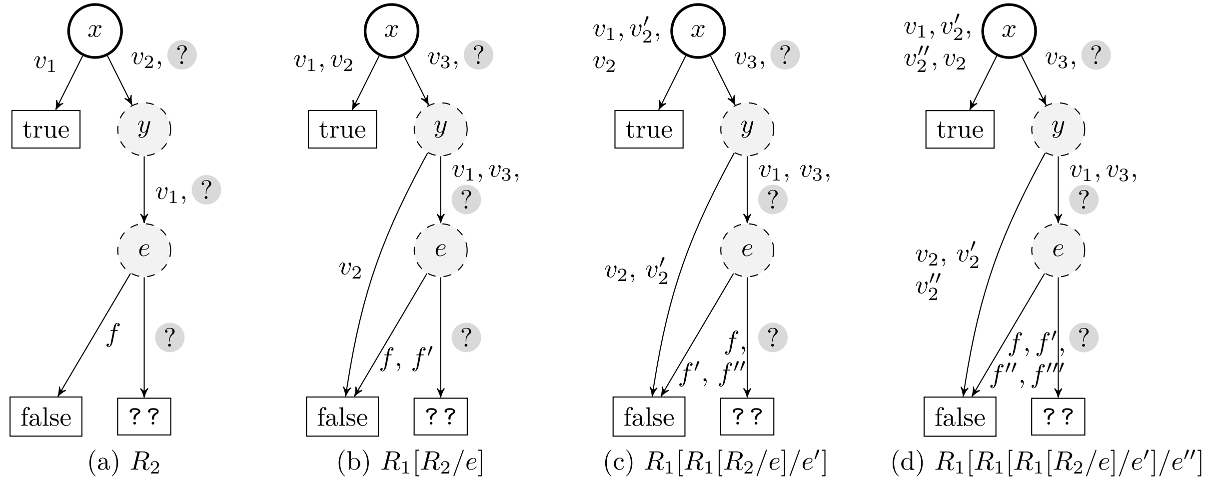


Figure 6.14: Evaluation trees for graphs from $L(L^\bullet)$.

exclamation mark represents an element that is not further specified but that exists and therefore has to be considered. In addition we relabel external vertices $v = \text{ext}_K[i]$ to ext_i to standardise the names between different rules. Anonymising the evaluation trees from Figure 6.14 results in the two anonymised ones given in Figure 6.15(a) and (b).

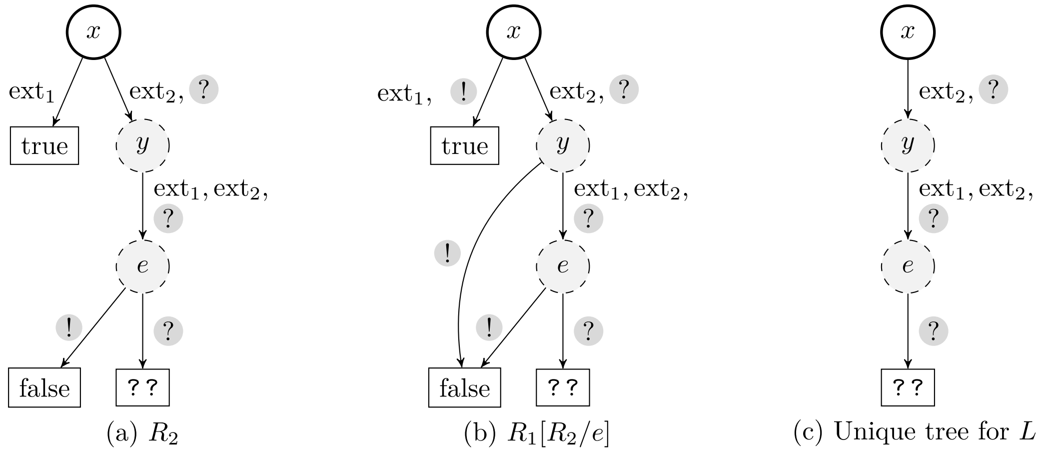


Figure 6.15: Anonymised evaluation trees for $L(L^\bullet)$.

Note that the evaluation trees from Figure 6.15(a) and (b) are logically equivalent. The transition between root node and the true-leaf could be omitted, as the root node is a universal node and true-successors have no influence to the final evaluation. The same holds for transitions from an existential node targeting the false-leaf. If we remove all successors of a universal (existential) node it is pre-evaluated to true (false). We do not discuss this optimisation and the impact to the operations, but we omit these transitions in upcoming evaluation trees. In Figure 6.15(c) the resulting unique evaluation tree of

$T_{L^\bullet}$ is given. The semantics of the tree is straightforward. To fulfil the property we need a context, that for the external vertex $x = \text{ext}_2$ provides a vertex y (which could also be ext_1 or ext_2 itself) such that there is an n-labelled edge in the context connecting this vertex with ext_2. As ext_2 is the only successor at the x-node we know that all other vertices within the represented graph have a corresponding vertex.

Definition 6.8 (Anonymised evaluation trees.):
Given graph $H \in HG_\Sigma$ and an evaluation tree $t \in$ Tree for H, the anonymised *evaluation tree is $anonym(t) = (N_t, L_t, q_t, v_t, e_t, \delta')$, where*

$$\delta' = \{(n, ext_i, n') \mid (n, \text{ext}_H[i], n') \in \delta_t\} \ \cup \ \{(n, \texttt{!}, n') \mid (n, x, n') \in \delta_t \wedge x \notin [\text{ext}_{H_T}]\}$$

Whenever we determine the evaluation tree for some replacement $H[K/e]$, we consider K to be minimised and anonymised. The set of minimised and anonymised evaluation trees for some nonterminal is always finite.

Lemma 6.4 (Finiteness of anonymised evaluation trees):
Given $G \in HRG_\Sigma$ and $\varphi(X_V, X_E) \in FO_\Sigma$, for any $X \in N_\Sigma$ and $\alpha \in \mathcal{A}_{\varphi,H}$ the set $\{anonym(evaltree(H, \alpha, \varphi(X_V, X_E))) \mid H \in L_G(X^\bullet)\}$ is finite.

Proof. We prove by induction over the number $k \in \mathbb{N}$ of quantifiers in $\varphi(X_V, X_E)$, that the number of possible anonymised evaluation trees is bounded by $T(k)$, where $T(n)$ is the recursive function $T(n) = T(n-1)^{\text{rk}(X)} + 2^{T(n-1)}$ with $T(0) = 3^{|Pred_\varphi|}$.

Induction Base $n = 0$**:**

If a formula $\varphi(X_V, X_E)$ is quantifier-free, the corresponding evaluation tree only consists of a leaf. The possible leaves are the true- and false-leaf, as well as one leaf for each element from $\{\texttt{f}, \texttt{t}, \texttt{?}\}^{Pred_\varphi}$. The cardinality of the last set is $3^{|Pred\varphi|}$. Note that any leaf for a sequence without any ?-value has been pre-evaluated to true or false, therefore the number of leaf labels is $3^{|Pred\varphi|-1}$. In most practical cases the number of formulae that cannot be evaluated to true or false is essential smaller. In any case the number of leaves is bounded by $3^{|Pred\varphi|-1} + 2 \leq T(k) = T(0) = 3^{|Pred\varphi|}$.

Induction Hypothesis:

The number of anonymised evaluation trees for $L \in N_\Sigma$ and $\varphi \in \text{FO}_\Sigma$ with $k \in \mathbb{N}$ quantifiers is bounded by $T(k)$.

Induction Step $k \to k+1$**:**

Let $\varphi = \mathcal{Q}x.\, \varphi' \in \text{FO}_\Sigma$ a FO-formula in prenex normal form with $k+1$ quantifiers. Then $\varphi' \in \text{FO}_\Sigma$ is a FO-formula with k quantifiers.

If $\mathcal{Q}$ is an edge quantifier, then the resulting evaluation tree consists of a root node with a set of anonymous outgoing transitions to evaluation trees for φ', i.e. the tree can be uniquely described by a set of evaluation trees for φ'. By induction hypothesis, the number of evaluation trees for φ' is bounded by $T(k)$. Therefore, the number of evaluation trees for φ is bounded by $2^{T(k)} = 2^{T((k+1)-1)} \leq T(k+1)$.

If $\mathcal{Q}$ is a vertex quantifier, then the resulting evaluation tree consists of a root node that besides the anonymous outgoing transitions has $\mathrm{rk}(X)$ additional successors labelled by $\mathrm{ext}_i, \ldots \mathrm{ext}_{\mathrm{rk}(X)}$, as external vertices are not anonymised. Each of these transition targets an evaluation tree for φ'. By induction hypothesis, the number of evaluation tree for φ' is bounded by $T(k)$. Therefore the number of evaluation trees for φ is bounded by $T(k)^{\mathrm{rk}(X)} * 2^{T(k)} \leq T(k+1)$. □

If the rank is fixed, the number of anonymised evaluation trees is approximately $2^{2^{\cdot^{\cdot^{\cdot^{2^{3^{|Pred\varphi|}}}}}}}$ where the height of the tetration is the number of quantifiers in φ. This corresponds to the complexity of the automaton construction by Courcelle [Cou90]. However, during the evaluation we do not consider all possible evaluation trees as we take the properties of the grammar into account. The experimental results (Section 6.4)show that the number of evaluation trees occurring in most practical applications, is quite low.

We denote by $\mathit{Tree}_{\varphi,\Sigma}$ the set of possible evaluation trees for $\varphi \in \mathrm{FO}_\Sigma$ and nonterminals from the alphabet Σ, i.e. with labels $\mathrm{ext}_1, \ldots \mathrm{ext}_m$ with $m = \mathit{max}(\{\mathrm{rk}(X) \mid X \in N_\Sigma\})$.

Evaluation Trees as Compositional Property

As the number of possible evaluation trees is bounded, we can use it as value set for a compositional property (Definition 2.16 on page 32). Given a grammar $G \in \mathrm{HRG}_\Sigma$ we let $p = (\eta, \mathit{Tree}_{\varphi,G})$, where $\eta(H) = \mathit{anonym}(\mathit{evaltree}(H, \emptyset, \varphi))$. The composition is realised via the $\bowtie$-operator. The resulting splitting grammar is denoted by G_φ. The nonterminals from G_φ are tuples of nonterminals of G and evaluation trees from $\mathit{Tree}_{\varphi,G}$, $N_{G_\varphi} \subseteq N_G \times \mathit{Tree}_{\varphi,G}$. Hereby it holds that for any $(X,t) \in N_{G_\varphi}$ the elements from $L((X,t)^\bullet)$ evaluate to the tree t, i.e. for each nonterminal all derivable graphs have the same evaluation tree.

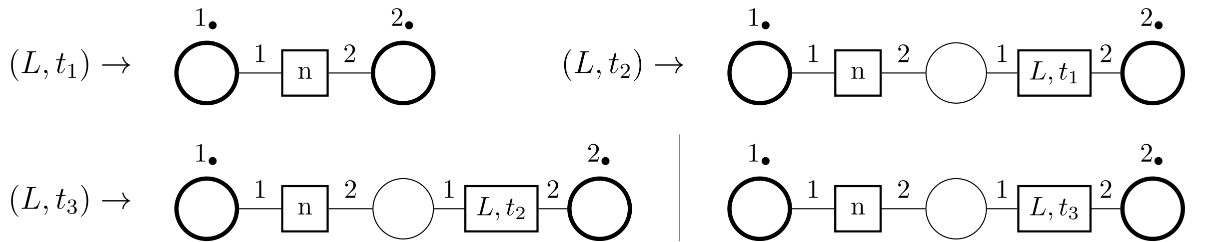


Figure 6.16: Splitting grammar for φ_3.

Example 6.10: FO splitting grammar
Reconsider the grammar G for linked lists given in Figure 6.13. The FO-sentence

$$\varphi_3 = \exists_V x_1, x_2, x_3, x_4.\ x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge x_1 \neq x_4 \wedge x_2 \neq x_3 \wedge x_2 \neq x_4 \wedge x_3 \neq x_4$$

is fulfilled by any graph with at least four pairwise distinct vertices. The splitting grammar G_φ is given in Figure 6.16. The nonterminal (L, t_1) represents an empty list segment, consisting of a start and an end vertex. Correspondingly (L, t_2) represents lists with three vertices, while (L, t_3) represents any list segment with at least four vertices. Any graph containing a (L, t_3)-labelled edge has at least four vertices. Therefore, evaluation tree t_3 is a true-leaf. The evaluation tree $t_2 \in Tree_{\varphi,G}$ is given in Figure 6.17. The ?-successor of every vertex is pre-evaluated to true, as one additional vertex in the context is sufficient to fulfil the formula φ_3. There is no path with repeating labels, as the formula is immediately falsified if two variables are interpreted with the same vertex.

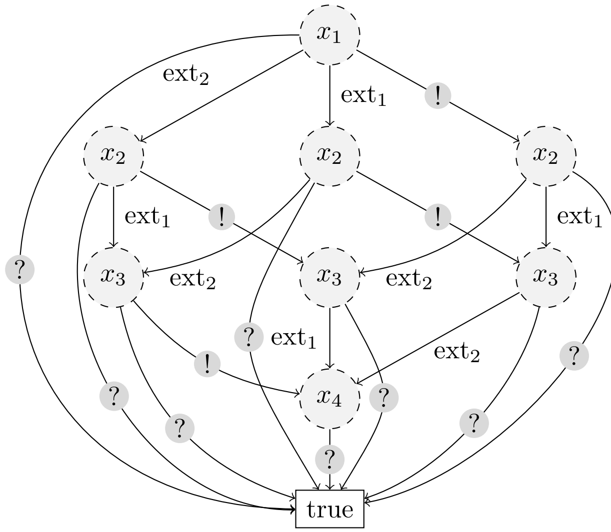


Figure 6.17: Evaluation tree t_2.

We prove Corollary 6.1 stating that given $G \in \mathrm{HRG}_\Sigma$ and a sentence $\varphi \in \mathrm{FO}_\Sigma$ it is decidable if for $H \in \mathrm{HG}_\Sigma$ none, some or all graphs $K \in L_G(H)$ satisfy φ.

Proof. We add a new nonterminal S and production rule $S \to H$ to the grammar G, where we use S as separate start symbol. For the resulting grammar we build the splitting grammar $G_\varphi \in \mathrm{HRG}_{\Sigma'}$. We assume that the splitting alphabet Σ' contains at least one nonterminal (S, t), as otherwise $L_G(H)$ is the empty set. The possible evaluations of the graphs from $L_G(H)$ are $\{eval(t) \mid (S, t) \in N_{\Sigma'}\}$. If this set is $\{\text{true}\}$, then *all* graphs fulfil the FO-property, if it is $\{\text{false}\}$ *none* of the graphs fulfil the property, while if the set is $\{\text{true}, \text{false}\}$ there are graphs fulfilling the property as well as graphs not fulfilling the property, i.e. *some* fulfil the property. □

For $G \in \mathrm{HG}_{\Sigma'}$ over the splitting alphabet Σ' there is always a unique evaluation, as the non-terminals are splitted such that there is a unique evaluation tree.

6.3 Monadic Second Order Logic

Monadic second order logic (MSO) is an extension of FO, where quantification is extended by quantification over sets, i.e. in the graph case sets of vertices and edges.

Definition 6.9 (Monadic second order formula over graphs (MSO)):
Given an alphabet Σ, a set of free vertex variables X_V, a set of free edge variables X_E, a set of free vertex-set variables $X_{\mathcal{P}(V)}$, and a set of free edge-set variables $X_{\mathcal{P}(E)}$. A monadic second order *formula $\varphi(X_V, X_E, X_{\mathcal{P}(V)}, X_{\mathcal{P}(E)})$ over Σ is recursively defined as follows:*

$$
\begin{aligned}
\varphi(X_V, X_E) \quad := \quad & \exists_{\mathcal{P}(V)} V.\ \varphi(X_V, X_E, X_{\mathcal{P}(V)} \uplus \{V\}, X_{\mathcal{P}(E)}) & \mid \\
& \forall_{\mathcal{P}(V)} V.\ \varphi(X_V, X_E, X_{\mathcal{P}(V)} \uplus \{V\}, X_{\mathcal{P}(E)}) & \mid \\
& \exists_{\mathcal{P}(E)} E.\ \varphi(X_V, X_E, X_{\mathcal{P}(V)}, X_{\mathcal{P}(E)} \uplus \{E\}) & \mid \\
& \forall_{\mathcal{P}(E)} E.\ \varphi(X_V, X_E, X_{\mathcal{P}(V)}, X_{\mathcal{P}(E)} \uplus \{E\}) & \mid \\
& \exists_V v.\ \varphi(X_V \uplus \{v\}, X_E, X_{\mathcal{P}(V)}, X_{\mathcal{P}(E)}) & \mid \\
& \forall_V v.\ \varphi(X_V \uplus \{v\}, X_E, X_{\mathcal{P}(V)}, X_{\mathcal{P}(E)}) & \mid \\
& \exists_E e.\ \varphi(X_V, X_E \uplus \{e\}, X_{\mathcal{P}(V)}, X_{\mathcal{P}(E)}) & \mid \\
& \forall_E e.\ \varphi(X_V, X_E \uplus \{e\}, X_{\mathcal{P}(V)}, X_{\mathcal{P}(E)}) & \mid \\
& \varphi(X_V, X_E, X_{\mathcal{P}(V)}, X_{\mathcal{P}(E)}) \wedge \varphi(X_V, X_E, X_{\mathcal{P}(V)}, X_{\mathcal{P}(E)}) & \mid \\
& \varphi(X_V, X_E, X_{\mathcal{P}(V)}, X_{\mathcal{P}(E)}) \vee \varphi(X_V, X_E, X_{\mathcal{P}(V)}, X_{\mathcal{P}(E)}) & \mid \\
& \neg\varphi(X_V, X_E, X_{\mathcal{P}(V)}, X_{\mathcal{P}(E)}) & \mid \\
& \pi(X_V, X_E, X_{\mathcal{P}(V)}, X_{\mathcal{P}(E)})
\end{aligned}
$$

with $\pi(X_V, X_E, X_{\mathcal{P}(V)}, X_{\mathcal{P}(E)})$ a predicate over vertex variables X_V, edge variables X_E, vertex-set variables $X_{\mathcal{P}(V)}$, and edge-set variables $X_{\mathcal{P}(E)}$, with $x, y \in X_V$, $e, f \in X_E$, $X \in X_{\mathcal{P}(V)}$, $F \in X_{\mathcal{P}(E)}$ and $\bar{s} \in X_V^\star$, where all entries of $\bar{s}$ are pairwise distinct, defined as follows:

$$
\pi(X_V, X_E) \quad := \quad x = y \mid e = f \mid \mathrm{att}(e) = \bar{s} \mid \mathrm{lab}(e) = a \mid x \in X \mid f \in F
$$

We denote the set of all monadic second order formulae *over alphabet Σ by MSO_Σ.*

Definitions and notations as quantifier-freeness, sentence, prenex normal form and others, known from FO are used here analogously. Note that we use lower-case letters for element variables (vertices and edges), while we use upper-case letters for set variables (vertex-variable-sets and edge-variable-sets).

Except the four new types of quantifiers, the membership predicate ($x \in X$) and the extension of the free variables by corresponding sets of set-variables the given definition is equal to the Definition 6.1 for FO-formulae.

In the following we give the semantics for the new quantifiers and predicates, while the rest can be carried over one-by-one from the semantics of FO-formulae. Given a terminal hypergraph $H \in \mathrm{HG}_{T_\Sigma}$, a first order formula $\varphi(X_V, X_E, X_{\mathcal{P}(V)}, X_{\mathcal{P}(E)}) \in \mathrm{MSO}_\Sigma$ over the same alphabet Σ and a variable interpretation $\alpha : (X_V \to V_H) \cup (X_E \to V_E) \cup (X_{\mathcal{P}(V)} \to \mathcal{P}(V_H)) \cup (X_{\mathcal{P}(E)} \to \mathcal{P}(V_E))$, we say H under α is a model of φ, denoted by $(H, \alpha) \models \varphi(X_V, X_E, X_{\mathcal{P}(V)}, X_{\mathcal{P}(E)})$, if

- $(H, \alpha) \models x \in X$ iff $\alpha(x) \in \alpha(X)$
- $(H, \alpha) \models f \in F$ iff $\alpha(f) \in \alpha(F)$
- $(H, \alpha) \models \exists_{\mathcal{P}(V)} X.\, \varphi(X_V, X_E, X_{\mathcal{P}(V)} \uplus \{X\}, X_{\mathcal{P}(E)})$ iff there exists a set of vertices $V \subseteq V_H$ such that $(H, \alpha[X \mapsto V]) \models \varphi(X_V, X_E, X_{\mathcal{P}(V)} \uplus \{X\}, X_{\mathcal{P}(E)})$.
- $(H, \alpha) \models \forall_{\mathcal{P}(V)} X.\, \varphi(X_V, X_E, X_{\mathcal{P}(V)} \uplus \{X\}, X_{\mathcal{P}(E)})$ iff for any set of vertices $V \subseteq V_H$ it holds that $(H, \alpha[X \mapsto V]) \models \varphi(X_V, X_E, X_{\mathcal{P}(V)} \uplus \{X\}, X_{\mathcal{P}(E)})$.
- $(H, \alpha) \models \exists_{\mathcal{P}(E)} F.\, \varphi(X_V, X_E, X_{\mathcal{P}(V)}, X_{\mathcal{P}(E)} \uplus \{F\})$ iff there is a set of edge $E \subseteq E_H$ such that $(H, \alpha[F \mapsto E]) \models \varphi(X_V, X_E, X_{\mathcal{P}(V)}, X_{\mathcal{P}(E)} \uplus \{F\})$.
- $(H, \alpha) \models \forall_{\mathcal{P}(E)} F.\, \varphi(X_V, X_E, X_{\mathcal{P}(V)}, X_{\mathcal{P}(E)} \uplus \{F\})$ iff for any set of edges $E \subseteq E_H$ it holds that $(H, \alpha[F \mapsto E]) \models \varphi(X_V, X_E, X_{\mathcal{P}(V)}, X_{\mathcal{P}(E)} \uplus \{F\})$.

Example 6.11: MSO-formulae

Due to the possibility to express set properties, MSO is more expressive than FO. For example, it is possible to express graph connectivity and graph colourability in MSO, while this is not possible in FO. Here we give formulae for these properties.

To express that within a graph some vertex y is connected to some vertex x, we check whether any set X of vertices, that contains x and that is closed under incident vertices also contains y. The property that set X is closed under incident vertices, can be expressed in MSO by

$$\mathit{close}(X) = \forall_V\, v, u.\ (\exists_E\, e.\ (\mathrm{att}(e) = v\,u \vee \mathrm{att}(e) = u\,v)) \to (v \in X \leftrightarrow u \in X).$$

Given $\mathit{close}(X)$ we can express that x and y are connected by the MSO-formula

$$\varphi_{x-y} \;=\; \forall_{\mathcal{P}(V)} X.\ \mathit{closed}(X) \to (x \in X \leftrightarrow y \in X).$$

We have to consider all closed sets, as it could be that x and y are in two different connected components. Thus x and y would not be connected. However, if we select

X such that it contains exactly these two components, then X is closed under incident vertices. Therefore, we have to ensure that we check the containment for the smallest components that are closed under incident vertices. Checking all sets ensures that also the smallest ones are included.

To express that a graph is three-colourable *we use one vertex set X_0, X_1, X_2 for each of the three colours. If a vertex is member of X_i then it is coloured with i. As each vertex should be assigned a colour, each vertex has to be in at least one set. As the colour of a vertex should be unique, vertices cannot be in more than one set. That is, the sets X_0, X_1 and X_2 form a partition of the set of all vertices. This can be expressed in MSO by the formula*

$$\begin{aligned}\varphi_{partition}(X_0, X_1, X_2) \;=\; \forall_V\, x.\; & x \in X_0 \lor x \in X_1 \lor x \in X_2 \\ \land\; & x \in X_0 \leftrightarrow (x \notin X_1 \land x \notin X_2) \\ \land\; & x \in X_1 \leftrightarrow (x \notin X_0 \land x \notin X_2) \\ \land\; & x \in X_2 \leftrightarrow (x \notin X_0 \land x \notin X_1).\end{aligned}$$

Given that X_0, X_1 and X_2 form a partition it remains to ensure that two incident vertices are not equally coloured, i.e. are not elements of the same set. We get the MSO-formula for three-colourability as

$$\begin{aligned}\varphi_{3C} \;=\;& \exists_{\mathcal{P}(V)}\, X_0, X_1, X_2.\; \varphi_{partition}(X_0, X_1, X_2) \\ \land\;& \forall_V\, x, y.\; (\exists_E\, e.\; \mathrm{att}(e) = x\, y) \to \\ & \Big(\quad (x \in X_0 \to y \notin X_0) \\ & \quad \land (x \in X_1 \to y \notin X_1) \\ & \quad \land (x \in X_2 \to y \notin X_2) \Big).\end{aligned}$$

The formulae can easily be adapted to two-colourability, as well as to general n-colourability. Note that in both examples we presume that all terminal edges are of rank two, i.e. consider directed graphs instead of hypergraphs. If this is not the case, then we have to extend the formula for each occurring rank of terminal edges.

Evaluation Trees

Given $\varphi \in \mathrm{MSO}_\Sigma$, $H \in \mathrm{HG}_\Sigma$ and a corresponding extended variable assignment $\alpha \in \mathcal{A}_{\varphi,H}$, we adapt the method *evaltree* for FO from Section 6.2.3 such that we can use it for the evaluation of MSO formulae. If the formula starts with a set quantifier, i.e. $\varphi = \mathcal{Q}X.\, \varphi'$, we add for each possible set of inner vertices $S \subseteq V_H \setminus [\mathrm{ext}_H]$ the successor *evaltree*$(H, \alpha[X \mapsto S], \varphi')$. Note that by considering only inner vertices, we postpone the membership decision for external vertices to the evaluation of the context. That is valid, as these vertices are also part of the context.

Given $H, K \in \mathrm{HG}_\Sigma$ and $e \in E_H$, we construct the evaluation tree $t_I = (I, \alpha, \varphi)$ for the replacement $I = H[K/e]$ - as before - by merging the evaluation trees $t_H =$ *evaltree*(H, α_H, φ) and $t_K =$ *evaltree*(K, α_K, φ). If φ starts with a vertex (edge) set

quantifier, then the outgoing transitions of the root of t_H are labelled with all sets of inner vertices (edges) of $V_H \setminus [\text{ext}_H]$ (E_H^T), i.e. there is one transition for each element from $\mathcal{P}(V_H \setminus [\text{ext}_H])$ ($\mathcal{P}(E_H^T)$), and the outgoing transitions of $root_{t_K}$ are labelled from $\mathcal{P}(V_K \setminus [\text{ext}_K])$ ($\mathcal{P}(E_K)^T$), while in t_I they are labelled from $V_I \setminus [\text{ext}_I] = (V_H \setminus [\text{ext}_H]) \uplus (V_K \setminus \text{ext}_K)$ ($E_I = E_H^T \uplus E_K^T$). Note that given $A \uplus B = C$ it holds that $\mathcal{P}(C) = \{a \cup b \mid (a,b) \in \mathcal{P}(A) \times \mathcal{P}(B)\}$. Therefore we get the successors of the merged node by combining all successors of t_H pairwise with all successors in t_K. That is for each pair of successors $(s_n, s_t) \in S = S(t_H) \times S(t_K)$ we build the evaluation tree $t_{(s_n,s_t)} = \mathit{evaltree}(H, \alpha[X \mapsto (s_n \cup s_t)], \varphi')$ recursively as result of merging $\mathit{evaltree}(H, \alpha_H[X \mapsto s_n], \varphi')$ with $\mathit{evaltree}(K, \alpha_K[X \mapsto s_t], \varphi')$. We compose the evaluation tree $t_I = t_H \bowtie t_K$ by combining the evaluation trees $t_{(s_n,t_n)}$ as follows:

$$
\begin{aligned}
N &= \textstyle\bigcup_{(s_n,s_t)\in S} N_{t_{(s_n,s_t)}} \cup \{n\},\\
L &= \textstyle\bigcup_{(s_n,s_t)\in S} L_{t_{(s_n,s_t)}},\\
\mathit{quant} &= \textstyle\bigcup_{(s_n,s_t)\in S} \mathit{quant}_{t_{(s_n,s_t)}} \cup [n \mapsto \mathit{quant}_{t_H}(\mathit{root}_{t_H})],\\
\mathit{var} &= \textstyle\bigcup_{(s_n,s_t)\in S} \mathit{var}_{t_{(s_n,s_t)}} \cup [n \mapsto \mathit{var}_{t_H}(\mathit{root}_{t_H})],\\
\mathit{eval} &= \textstyle\bigcup_{(s_n,s_t)\in S} \mathit{eval}_{t_{(s_n,s_t)}},\\
\delta &= \textstyle\bigcup_{(s_n,s_t)\in S} \left(\delta_{t_{(s_n,s_t)}} \cup \{(n, s_n \cup s_t, \mathit{root}_{t_{(s_n,s_t)}})\}\right),\\
\mathit{root} &= n.
\end{aligned}
$$

In Figure 6.18(a)-(c) H, K and the result $H = H[K/e]$ are given exemplarily, as well as the resulting evaluation trees t_H, t_K and t_I in Figure 6.18(d)-(f) for formula $\varphi = \forall_{\mathcal{P}(V)} X. \exists y. \varphi'$.

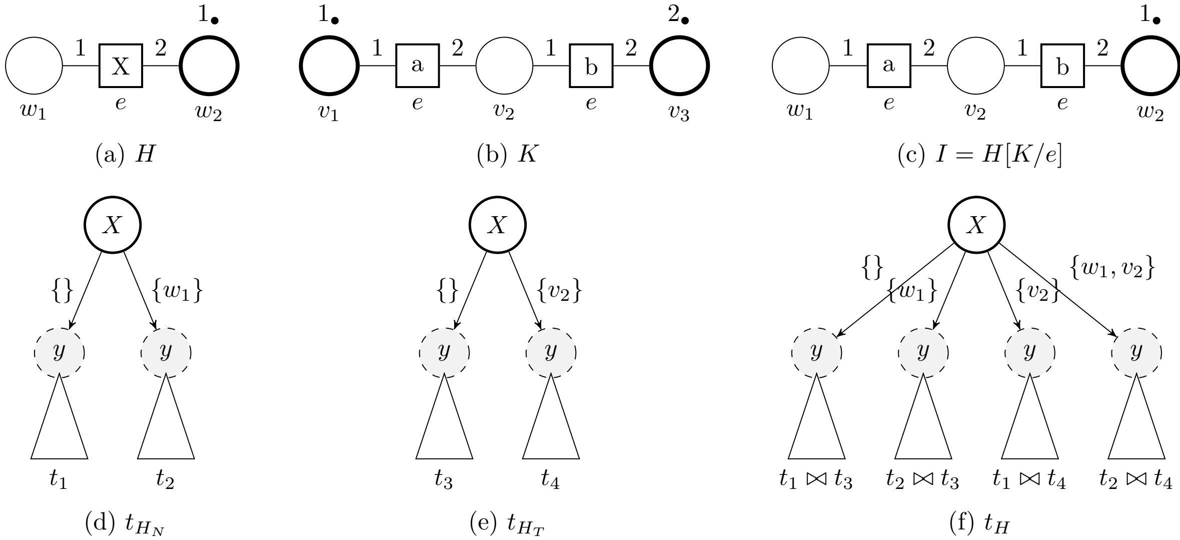


Figure 6.18: Merging of evaluation trees for set quantifiers.

As we combine any successor of a tree with each successor of the other, there is no need for ?-successors, i.e. in both trees we always select one of the set-successors. Further

as we combine any successor of the first tree with any of the second, we can anonymise the successors labels and get only ! -successors. The number of possible anonymised evaluation trees is still bounded, i.e. Lemma 6.4 also holds for MSO formulae. Indeed there are fewer combinations of successors for a set quantifier than for a vertex - and edge quantifier, as for the former all successors are anonymous.

Example 6.12: Splitting grammar for MSO

We have seen how to express n-colourability as MSO-formula in Example 6.11. To express two-colourability one set-variable is sufficient. Consider the following formula φ_{2C} for two-colourability.

$$\varphi_{2C} \;=\; \exists_{\mathcal{P}(V)} X.\ (\forall_V\, x, y.\ \forall_E\, e.\ \mathrm{att}(e) = x\, y \rightarrow (x \in X \leftrightarrow y \notin X)\,).$$

We consider vertices in the set X to be coloured by 1, while vertices not in X are coloured by 0. Note that doing so the partitioning is always given and it suffices to ensure that incident vertices are not equally coloured, i.e. if two vertices are incident exactly one is in X. As $y \notin X$ is the short notation for $\neg(y \in X)$, the predicates of φ_{2C} are $\pi_1 = \mathrm{att}(e) := x\, y$, $\pi_2 := x \in X$ and $\pi_3 := y \in X$.

Reconsider once more the grammar G for linked lists given in Figure 6.13. The rules of the splitting grammar $G_{\varphi_{2C}}$ are given in Figure 6.19. The nonterminal L that represented an arbitrary singly linked list segment between the two external vertices, is split into the three nonterminals (L, t_1), (L, t_2) and (L, t_3). The corresponding evaluation trees are given in Figure 6.20. From nonterminal (L, t_1) only an empty list segment can be derived, i.e. a singly linked list containing only the external vertices. List segments of odd length can be derived from (L, t_2), while list segments of even length can be derived from (L, t_3).

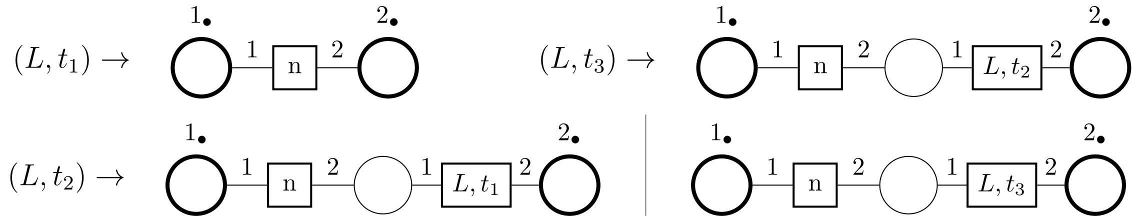
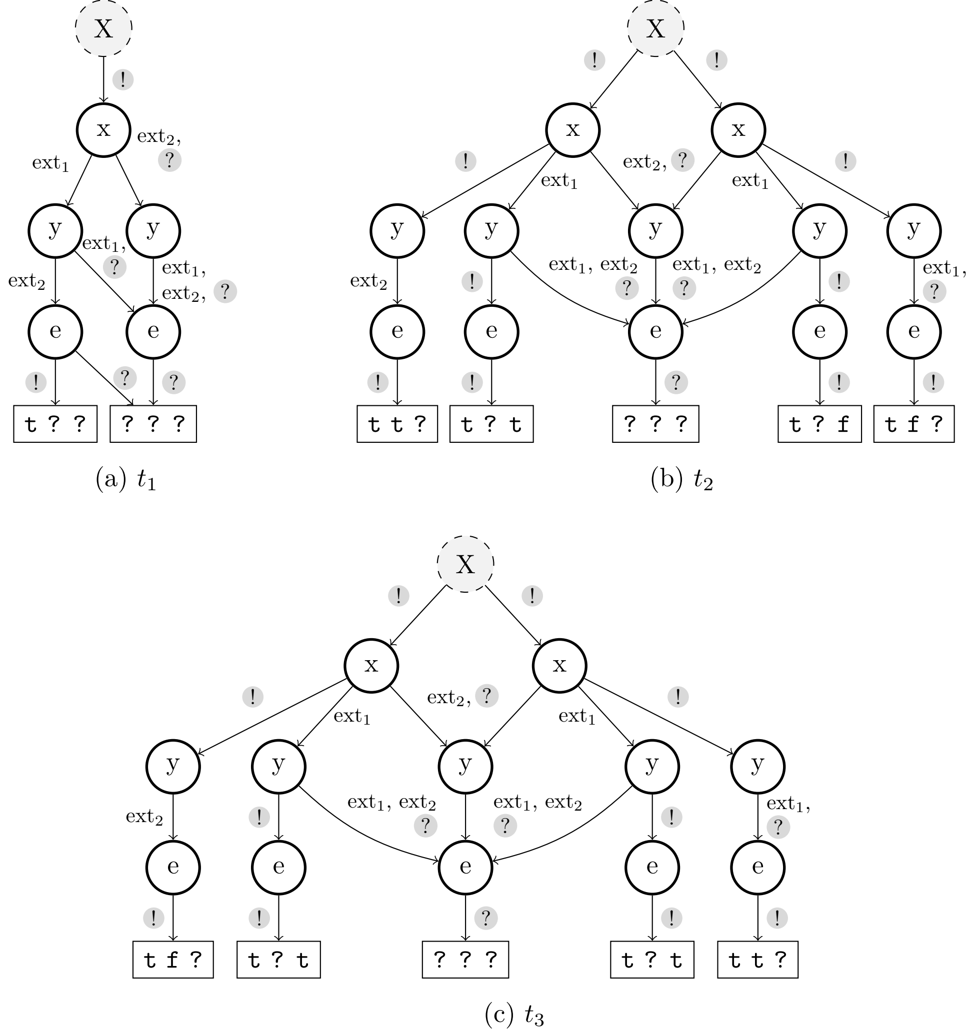


Figure 6.19: Splitting grammar $G_{\varphi_{C2}}$

Consider the tree t_2 that we get for lists of odd length. The X-node has two anonymous successors. Enumerating the elements inside the list segment, starting at the successor of the first external vertex, for the left subtree we decided that X contains all odd vertices, while the right subtree X contains the even ones. For any other set the evaluation tree is pre-evaluated to false, as incident vertices of the list are equally coloured. Note that in the left subtree there exist leaves for which the second and third predicate is true, while these in the right subtree are false. Consider the `t?t`*-leaf in the left subtree. The second predicate that is evaluated to* `?` *is $\pi_2 := x \in X$. To reach this leaf we have to choose the* ext_1 *transition at the x-node, that is we decided x to*



(a) t_1

(b) t_2

(c) t_3

Figure 6.20: Evaluation trees for $G_{\varphi_{2C}}$

be ext_1. *As described before we do not determine the set-membership for external nodes. That is the reason why the predicate* $\pi_2 := x \in X$ *is evaluated to* **?**. *Note that the first predicate* $\pi_1 := \text{att}(e) = x\,y$ *is evaluated to* **t**, *that is the edge* e *that we select at the e-node is the one pointing from* ext_1 *to its successor, which is the anonymous vertex that we decide to be* y. *As seen before within the left subtree the first vertex of the list segment is element of* X *and therefore* $\pi_3 := y \in X$ *is evaluated to* **t**. *As the matrix of* φ_{2C} *is* $\beta = \pi_1 \rightarrow (\pi_2 \leftrightarrow \neg\pi_3)$, *the leaf can only be evaluated to true, if the predicate* π_2 *becomes* **f**, *i.e. if* $\text{ext}_1 \notin X$. *Note that otherwise the whole left subtree is evaluated to false, as all its nodes are universal.*

Equally the **tt?***-leaf can be evaluated to true only if* $\text{ext}_2 \notin X$, *as* $\pi_3 = y \in X$ *must be false to evaluate the leaf to true and* y *is decided to be* ext_2. *Therefore only if neither* ext_1 *nor* ext_2 *are elements of the set* X *the left subtree is fulfilled. In the*

right subtree there are the leaves **`t?f`** *and* **`tf?`**, *such that the right subtree becomes true if and only if both external vertices are elements of the set X. As the X-node is existential the whole tree is evaluated to true if either both external vertices are elements of the set X or if both are not element of the set X.*

The evaluation tree t_3 can be interpreted similarly. Here the left subtree contains the leaves **`tf?`** *and* **`t?t`**, *such that X has to contain* ext_2 *but not* ext_1 *in order to evaluate to true. While to evaluate the right subtree to true, X has to contain* ext_1 *and not* ext_2. *That is, t_3 is evaluated to true if one external vertex is element of X, while the other is not.*

We have seen how to get splitting grammars for MSO-formulae. That brings us the main theorem of the chapter.

Theorem 6.4 (Decidability of MSO properties [Cou90]):
Given $G \in HRG_\Sigma$, sentence $\varphi \in FO_\Sigma$ it is decidable if for $H \in HG_\Sigma$ none, some or all *elements $K \in L_G(H)$ fulfil φ.*

Note that this is Courcelle's theorem [Cou90], that we mentioned earlier in the introduction of Section 6.2. The proof of the theorem can be done analogously to the proof of Corollary 6.1. That is, we have given a constructive proof of Courcelle's theorem. Note that the theoretical worst case time complexity that we provide for our approach is not better than the one by Courcelle. That is because theoretically it could happen that we have to build all possible evaluation trees before one reoccurs. However, the experimental evaluatios we considered so far lead to promising results. The results of the experimental evaluations are given in the next section. As a byproduct of the construction we get the splitting grammar over the splitting alphabet. As seen before for any graph over the splitting alphabet it holds that either all derivable terminal graphs fulfil the property or none. This gives us the following theorem, that again was originally proven by Courcelle [Cou90].

Theorem 6.5 (Intersection of HRG and MSO defined sets [Cou90]):
Given $G \in HRG_\Sigma$, $H \in HG_\Sigma$ and sentence $\varphi \in FO_\Sigma$ and a finite set of start graphs $S \subset HG_\Sigma$ there exists a grammar $G' \in HRG_{\Sigma'}$ and a finite set of start graphs $S' \subset HG_{\Sigma'}$ with

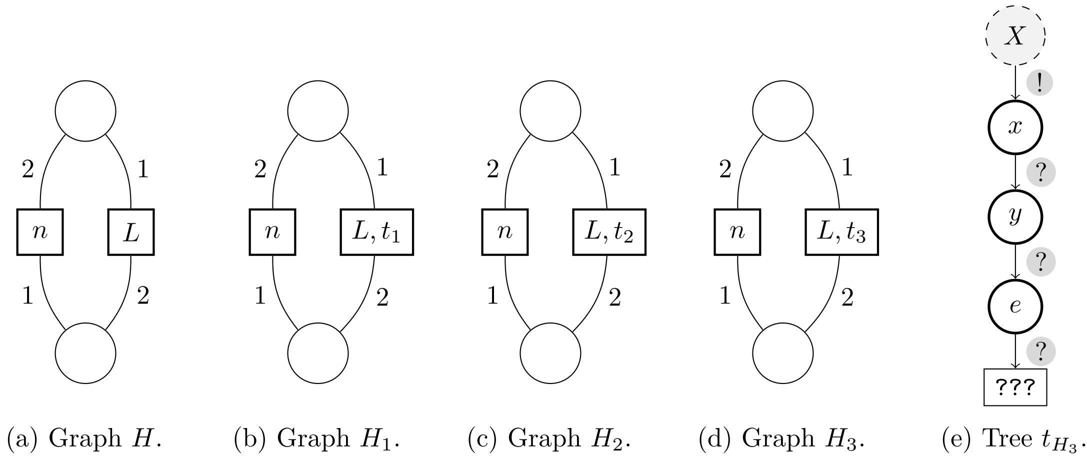
$$L_{G'}(S) \quad = \quad \{K \in L_G(H) \mid K \models \varphi\}$$

can be described by a grammar .

Proof. We prove Theorem 6.5 by giving a construction for the grammar G' and H'. As we have seen before, given G, we can build the quasi-equivalent splitting grammar $G_\varphi \in \mathrm{HRG}_{\Sigma'}$ for the MSO-formula φ. We let $G' = G_\varphi$ be the splitting grammar. Together

with the splitting grammar we get the mapping function $\gamma_{G \to G'}$ for which it holds that $L_G(S) = L_{G'}(\gamma_{G \to G'}(S))$ (see Definition 2.10). Note that as G' is a splitting grammar for φ and as $\gamma_{G \to G'}(S) \subseteq \mathrm{HRG}_{\Sigma'}$ it holds for any graph $H \in \gamma_{G \to G'}(S)$ that φ is either fulfilled for all or none of the graphs in $L_{G'}(H)$. As stated by Theorem 6.4 it is decidable if for the elements of $L_{G'}(H)$ the property is fulfilled or not. Thus we can determine the set $S' = \{H \in \gamma_{G \to G'}(S) \mid \forall\, K \in L'_G(H)\}.\ K \models \varphi$. Note that as $L'_G(\gamma_{G \to G'}(S)) = L_G(S)$ it holds that $L_{G'}(S') = \{K \in L_G(H) \mid K \models \varphi\}$, i.e. is the set of graphs that fulfil the property φ, while $L_{G'}(L'_G(\gamma_{G \to G'}(S) \setminus S') = \{K \in L_G(H) \mid K \not\models \varphi\}$, is the set of graphs from $L_G(S)$ that do not fulfil the property φ. □

Example 6.13: Intersection of graph languages defined by HRG and MSO
Reconsider the grammar G for singly linked lists and the MSO-formula φ_{2C} from Example 6.12, expressing two-colourability. The start graph H is depicted in Figure 6.21(a).



(a) Graph H. (b) Graph H_1. (c) Graph H_2. (d) Graph H_3. (e) Tree t_{H_3}.

Figure 6.21: Quasi-equivalent start graphs.

The splitting grammar G' is given in Figure 6.19 on page 166, the quasi-equivalent start graphs are $\gamma_{G \to G'}(H) = \{H_1, H_2, H_3\}$. The graphs are given in Figure 6.21(b)-(d). From H_1 the single derivable graph is a cyclic list with two vertices, thus H_1 is two-colourable. From H_2 we can derive arbitrary cyclic list of odd length. As the number of vertices is odd and as the list is cyclic, there is no colouring such that all incident vertices are coloured differently. The evaluation trees for H_2 consists of a single false-leaf, i.e. $H_2 \not\models \varphi_{2C}$. The evaluation tree t_{H_3} for H_3 is given in Figure 6.21(e). Note that evaluating t_{H_3} yields true, as if we remove all ?-transitions the universal x-node has no successors and therefore is evaluated to true- resulting in a true successor for the existential X-node. The evaluation tree for H_1 is the same as the one for H_3. Therefore $S' = \{H_1, H_3\}$. Note that $L_{G'}(S')$ yields all cyclic lists of even length.

6.4 Experimental Results

We implemented a prototype tool for the evaluation of MSO-formulae over languages of hypergraphs defined via HRGs. The tool is written in Java and is an implementation of the technique presented in this chapter. Beside the ones presented here, we realised some optimisations like memoization of merging results. We plan to use the tool as basis for further optimisations.

The evaluation results presented here were obtained on a MacBook Pro with a 2 GHz Intel Core i7 processor and 8 GB RAM. We used the Java(TM) SE Runtime Environment (build 1.7.0_17-b02) with a heap size of 20 MB.

For the experiments we used the different FO- and MSO-formulae introduced in this chapter. By *Successor* we refer to the property that each vertex of a graph has an n-successor. The corresponding formula φ_{suc} was given in Example 6.4 on page 136 (three quantifiers). The formula φ_{dll} from Example 6.1 on page 129 is represented here by *DLL* (four quantifiers). A graph fulfils the *DLL* property if it represents a doubly linked list, i.e. for any n-edge there is a p-edge with reversed orientation. The previous formulae are FO-formulae. The MSO-formulae we consider are connectivity,

$$xCy = \exists_V\, x, y.\, (\, \exists_E\, e_x, e_y.(\, \mathrm{lab}(e_x) \,\wedge\, \mathrm{att}(e_x) = x \wedge \mathrm{lab}(e_y) \,\wedge\, \mathrm{att}(e_y) = y\,) \wedge \varphi_{x-y}(x, y)\,),$$

stating that there is a vertex with an attached x-edge and one with an attached y-edge, such that these vertices are within one component of the graph (eight quantifiers). Formula φ_{x-y} was introduced in Example 6.11 on page 163 (four quantifiers). By *2C(short)* we refer to the formula φ_{2C} for two-colourability, introduced in Example 6.12 on page 166 (four quantifiers). Formula *2C(standard)* is the adaption of the formula for three-colourability for two-colourability, defined below (six quantifiers).

$$\begin{array}{rcl} \mathit{2C(standard)} & = & \exists_{\mathcal{P}(V)}\, X, Y.\, (\\ & & \forall_V\, a.\, (\, a \in X \leftrightarrow a \notin Y\,) \\ & \wedge & \forall_V\, x, y.\, (\, \exists_E\, e.\, (\, \mathrm{att}(e) = x\, y\,) \to (\, x \in X \leftrightarrow y \notin X\,))) \end{array}$$

Note that the formulae 2C(short) and 2C(standard) are logically equivalent, while they differ in the number of quantifiers.

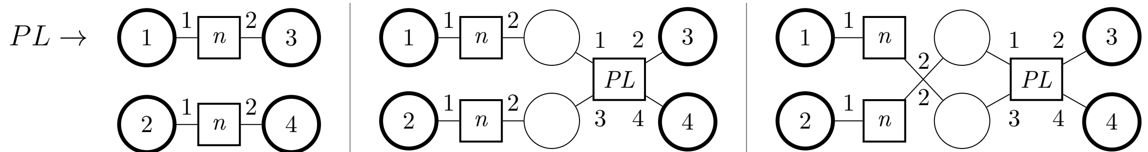
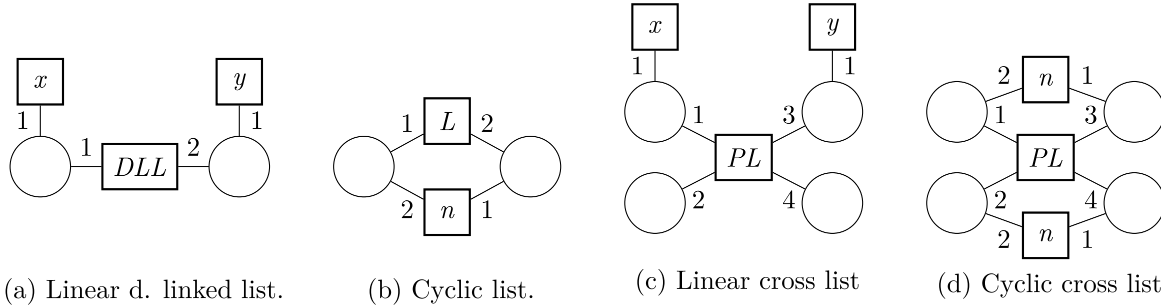


Figure 6.22: Cross-list grammar.

We use three different grammars. The first one is the *List* grammar from Figure 6.13 on page 157 for singly-linked lists. The second is called *Cross-List* and contains the rules from Figure 6.22. It generates two parallel lists one starting at tentacle $(C, 1)$, the other



(a) Linear d. linked list. (b) Cyclic list. (c) Linear cross list (d) Cyclic cross list

Figure 6.23: Start graphs.

at $(C, 2)$. The endpoints of the lists are the tentacles $(C, 3)$ and $(C, 4)$. Where which list ends depends on the rules used for the derivation, as the third rule flips the endpoints. The third grammar is the one from Figure 3.6 on page 46 for doubly linked lists. The different start graphs are given in Figure 6.23.

For different combinations of graphs, grammars and formulae the results, as well as runtimes, number of generated nodes and number of merge operations are given in Figure 6.24.

formula	grammar	start graph	result	#nt	#nodes	#merges	time
Successor	List	Linear List	none	1	11	17	80 ms
Successor	List	Cyclic List	all	1	13	17	80 ms
DLL	List	Linear List	all	2	40	92	110 ms
DLL	List	Cyclic List	none	2	33	64	90 ms
2C (short)	List	Cyclic List	some	3	80	253	110 ms
2C (short)	Cross-List	Cyclic C. List	some	5	169	2,640	170 ms
2C (standard)	List	Cycle	some	6	391	1,959	269 ms
2C (standard)	Cross-List	Cyclic C. List	some	9	764	8,824	740 ms
Reachability	List	Linear List	all	29	11,673	425k	5,936 ms
Reachability	Cross-List	Linear C. List	some	11	34,343	551k	8,319 ms

Figure 6.24: Experimental results.

6.5 MSO-Logic and JVM-States

We proposed FO and MSO as a possibility to describe properties of a program state. In the last sections we introduced FO and MSO for general hypergraphs and hyperedge replacement grammars. In this section we focus on JVM states as introduced in Chapter 5 and we adapt the definition of MSO to these.

6.5.1 Pointers Instead of Edges

The edges within concrete JVM states are considered to be pointers, i.e. all edges are of rank two and we consider them to be directed, pointing from the vertex at the first tentacle to the vertex at the second. Further for each label there is at most one outgoing pointer at each vertex. Therefore any edge can be uniquely described as combination of vertex and pointer label. Consider the edge e in Figure 6.25(a), with $\text{att}(e) = v_1\, v_2$ and $\text{lab}(e) = a$. The edge e is a terminal edge and considered to be a pointer from v_1 to v_2, as depicted in Figure 6.25(b). The properties of heap configurations ensure that the edge can be uniquely determined by the tuple (v_1, a).

(a) Representation as hyperedge (b) Representation as pointer

Figure 6.25: Representation of edges

Instead of the two MSO-predicates $\text{att}(e) = x\, y$ and $\text{lab}(e) = a$ we provide a combined property $x.a = y$, that is fulfilled if there is an a-labelled edge from x to y, i.e. the formula $x.a = y$ is equivalent to the formula $\exists_E\, e.\ (\text{att}(e) = x\, y \wedge \text{lab}(e) = a)$. In addition, we add the predicate $x.? = y$ that is equivalent to $\exists_E\, e.\ \text{att}(e) = x\, y$, i.e. is fulfilled if there is an arbitrarily labelled pointer from vertex x to vertex y. These notations are also more convenient for the description of pointers as it corresponds to the notation of object-oriented languages such as Java, C++ or Objective-C.

By replacing the predicates $\text{att}(e) = x\, y$ and $\text{lab}(e) = a$ we eliminate any predicate that makes use of edge variables. Thus it is no longer reasonable to quantify over edges. Correspondingly we do not consider edge quantifiers any more.

Note that a logic without quantification over edge sets is less expressive. For example, the existence of a Hamiltonian cycle cannot be expressed without quantification over sets of edges [EF95]. However, properties over heap structures usually focus on object (vertices) rather then on pointers (edges).

6.5.2 Typed Vertices

JVM states are heap configurations and thus each vertex is assigned a type. The types determine the semantics as well as the outgoing pointers of the represented object. Therefore there is an interest in describing properties that depend on the type of objects. We provide the predicate $\text{type}(x) = T$ that is fulfilled if the vertex referenced by x is of type $T \in \mathbb{T}$ ($\text{type}(\alpha(x)) \preceq T$). Whenever we want to express a property $\varphi(x)$ for vertices of type $T \in \mathbb{T}$, we will use a formula of the form $\exists_V\, x.\ \text{type}(x) = T \rightarrow \varphi(x)$. As

the type also determines the outgoing pointers, edge predicates $x.a = y$ are commonly preceded by such a formula segment. We adapt the quantifiers such that they quantify over vertices of a specific type only. We write $\mathcal{Q}\, x \in T.\ \varphi$ to quantify over vertices of type $T \in \mathbb{T}$, or $\mathcal{Q}\, X \subseteq T.\ \varphi$ to quantify over sets of vertices of type $T \in \mathbb{T}$, respectively. That is, x is either of the type T, or of a subtype ($\text{type}(x) \preceq T$). Note that we omit the V and $\mathcal{P}(V)$ index, as we do not consider edge quantification anymore. The distinction between element and set quantifiers is given by $\in$, respectively $\subseteq$.

Beside shorter and more readable formulae the use of extended quantifiers also reduces the branching within the evaluation trees, as for any quantifier we only need to branch over the vertices of the given type. Quantifications over all heap objects are still possible by quantifying over the type Object, which is super-type for all objects.

6.5.3 Variables

Considering states of a program, variables play an important role. We add the predicate $var(v) = x$, which is fulfilled if the static or local variable v of the active method refers to the object that is assigned to the MSO-variable x. Note that in a JVM the targets of static variables as well as the variables of the active method are always concrete. Therefore a variable predicate can be evaluated to `t` only within the start graph, while within the grammar it is evaluated to `f` for inner vertices and to `?` for external ones.

6.5.4 MSOJ

Considering the above ideas and concepts we define a version of MSO that is adapted for consideration of JVM states, and that we call *Monadic Second Order Logic for JVM states* (MSOJ).

Definition 6.10 (MSO for JVM States):
Given a typed alphabet Σ, monadic second order formulae for JVM states φ *over the alphabet Σ are recursively defined as:*

$$\begin{array}{rcl} \varphi & := & \exists\, x \in T.\ \varphi \mid \forall\, x \in T.\ \varphi \mid \exists\, X \subseteq T.\ \varphi \mid \forall\, X \subseteq T.\ \varphi \mid \\ & & \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \pi \mid \neg\, \varphi \end{array}$$

where $T \in \mathbb{T}$ is a type, and π is a predicate with logic variables x and y, program variable v, type $T \in \mathbb{T}$, and field $a \in \mathbb{F}$, defined as follows:

$$\pi \ := \ var(v) = x \mid x = y \mid \text{type}(x) = T \mid x.a = y \mid x.? = y \mid x \in X$$

We denote the set of all monadic second order formulae over JVM states *by $MSOJ_\Sigma$.*

Example 6.14: MSOJ formula for reachability

A MSOJ-definable property that is commonly used for verification purpose is reachability, i.e. can we reach a heap object y from some heap object x via pointers. The type of pointers can be arbitrary or restricted by a set of fields $F \in \mathbb{F}$. We write $x \overset{?}{\rightsquigarrow} y$ for the former, $x \overset{F}{\rightsquigarrow} y$ for the latter. As the connected property, introduced in Example 6.11, the definition of reachability is based on sets closed under successors. A set is closed if for any contained object also its successors are contained. In case that a set of fields $F \subseteq \mathbb{F}$ is given restricted to successors labelled from F. The corresponding formulae are as follows:

$$closed(X) \quad := \quad \forall x, y \in \mathsf{Object}.\ (x \in X \wedge x.? = y) \rightarrow y \in X$$

$$closed(X, F) \quad := \quad \forall x, y \in \mathsf{Object}.\ (x \in X \wedge \bigvee_{f \in F} x.f = y) \rightarrow y \in X.$$

Consider a singly linked list and the unrestricted closed-predicate. Any non-empty set X not containing the last element of the list is not closed, as sets are only closed under successors, if for each element of the set each of its successors are element of the set. To check reachability it is not sufficient to consider an arbitrarily closed set, as also two not connected objects could be part of one closed set, e.g. a set containing two separated lists is closed. Instead we have to check for $x \overset{?}{\rightsquigarrow} y$, that every closed set containing x also contains y. If this is the case then also the smallest set containing x, i.e. the set that contains only x and its successors, contains y and thus y is successor of x.

$$x \overset{?}{\rightsquigarrow} y \quad := \quad \forall X \subseteq \mathsf{Object}.\ closed(X) \rightarrow (x \in X \rightarrow y \in X)$$

$$x \overset{F}{\rightsquigarrow} y \quad := \quad \forall X \subseteq \mathsf{Object}.\ closed(X, F) \rightarrow (x \in X \rightarrow y \in X)$$

The two MSOJ-formulae $x \overset{?}{\rightsquigarrow} y$ and $x \overset{F}{\rightsquigarrow} y$ will be used in the remainder of this chapter as well as in Chapter 7. In the following example we use the reachability predicates to describe doubly linked lists.

Example 6.15: MSOJ formula for doubly linked lists

Reconsider the implementation of a doubly linked list as introduced in Example 3.5. A doubly linked list consists of a DLList*-object, that contains a head-pointer to the first element of a list of doubly linked elements of the type* ListElement *and tail-pointers to the last element of the same list. Any* ListElement *has a list-pointer to the enclosing* DLList*-object, in addition to the next and previous pointer. The corresponding Java class definitions are given in Figure 6.26.*

Given a JVM state containing doubly linked lists, we want to check for each DLList*-object l, if it is a valid doubly linked list, i.e.*

1. *There is a head and tail element, such that the tail is reachable from the head via next-pointers as well as the head from the tail via previous-pointers.*

   $connected(l) \ := \ \exists h, t \in \mathsf{ListElement}.\ l.head = h \ \wedge \ l.tail = t \ \wedge \ h \overset{next}{\rightsquigarrow} t \ \wedge \ t \overset{prev.}{\rightsquigarrow} h$

```
public class DLList{
  ListElement head, tail;
}
```

(a) Class definition for DLList

```
public class ListElement{
  ListElement next, previous;
  DLList list;
}
```

(b) Class definition for ListElement

Figure 6.26: Java class definitions for doubly linked lists.

2. *For each list element in the list, i.e. all elements reachable from l, it holds that it is the successor of its predecessor and vice versa,*

$$doublylinked(l) := \forall x \in \mathsf{ListElement}.\ l \overset{?}{\rightsquigarrow} x \ \rightarrow\ (x.next = y \ \leftrightarrow\ y.previous = x)$$

3. *and that the list-pointer of any* ListElement *in the list points to l.*

$$backpointer(l) := \forall x, y \in \mathsf{ListElement}.\ l \overset{?}{\rightsquigarrow} x \ \rightarrow\ x.list = l$$

That is all DLList*-objects represent a correct doubly linked list, if the following formula is fulfilled*

$$\varphi_{dll} := \forall l \in \mathsf{DLList}.\ connected(l) \wedge doublylinked(l) \wedge backpointer(l).$$

6.5.5 Experiments: Lindstrom's algorithm

Reconsider Lindstrom's algorithm as introduced in Section 1.1. We already presented a suitable abstraction grammar in Chapter 5, where we also generated a corresponding abstract state space for traversing arbitrary binary trees (Section 5.5). Our prototype tool determined 4521 reachable configurations and among them four terminal configurations. In Section 5.5, we already observed that the terminal configurations represent trees. While this was a rather vague observation we now aim at checking treeness automatically. We express the properties cycle- and sharing-freeness by the following MSOJ-formulae. Formula φ_{cycle} expresses cycle-freeness by ensuring for each tree node that there is no path from one of its successors back to the node:

$$\varphi_{cycle} := \forall x, y \in \mathsf{Tree}.\ x.? = y \rightarrow \neg(y \overset{?}{\rightsquigarrow} x)$$

The formula $\varphi_{sharing}$ expresses sharing-freeness. For any pair of successors *left* and *right* there is no common reachable state:

$$\varphi_{sharing} := \forall x, l, r \in \mathsf{Tree}.\ (x.left = l \wedge x.right = r) \rightarrow \neg(\exists y \in \mathsf{Tree}.\ (l \overset{?}{\rightsquigarrow} y \wedge r \overset{?}{\rightsquigarrow} y))$$

We use an extension of the tool Juggrnaut [HNR10] used in Section 6.4 in order to check treeness for all 4521 reachable configurations of Lindstrom's algorithm. Note

that the tool implements some additional optimisations which we have not presented here. The experimental results for checking φ_{cycle}, $\varphi_{sharing}$ and $\varphi_{cycle} \wedge \varphi_{sharing}$ are given in Figure 6.27. The row *#nonterminal* gives the amount of nonterminals in the splitting grammar, *#model* is the number of states that fulfil the corresponding formula, while *#not model* gives the amount of states that do not model the formula. The time spend to generate the splitting grammar can be found in the row *splitting*, while the row *checking* contains the time needed to check the formula on each of the 4521 states.

formula	#nonterminal	#model	#not model	splitting	checking
φ_{cycle}	5	4521	0	< 1 s	9 s
$\varphi_{sharing}$	17	3894	627	1 s	140 s
$\varphi_{cycle} \wedge \varphi_{sharing}$	40	3894	627	3 s	390 s

Figure 6.27: Experimental results.

Note that even if treeness appears to be a rather simple property it involves twelve vertices and three set variables (most of them hidden within the auxiliary formula $x \overset{?}{\rightsquigarrow} y$). While all states are cycle-free there are 627 states with sharing. One of these states is given in Figure 6.28. States with sharing occur as intermediate states during rotation of the pointers (see also Section 1.1).

There is no need of refining the abstraction as we get a unique true/false-evaluation for all states. Otherwise we could use the determined splitting grammar to generate an abstract state space with unique evaluations (see page 162). Note that the generated splitting grammars contain more nonterminals than necessary. In fact splitting is not necessary at all, as with respect to the formula the derivable graphs for T as well as for P are not distinguishable, i.e. there is no context in which a T- or P-edge would not be uniquely evaluated. The overhead added by the additional nonterminals is best noticeable comparing the runtimes of φ_{cycle} and $\varphi_{sharing}$ with the one for $\varphi_{cycle} \wedge \varphi_{sharing}$. As the latter is a simple *and*-combination of the formers its runtime should not exceed the sum of the runtimes of the former two.

6.6 Conclusions

This chapter presented an alternative implementation of the theorem by Courcelle [Cou90]. The approach is based on evaluation trees that we first introduced for a single graph and an FO-formula (Definition 6.5). Instead of considering isolated graphs we can also generate evaluation trees for a partial graph taking possible context into account (Definition 6.6). Given a hyperedge replacement we can compose the evaluation tree of the replacement result from the partial evaluation trees of the involved graphs (Theorem 6.3). This composition is the first step towards the analysis of HRGs. As the number of graphs represented by HRGs is potentially infinite so is the number

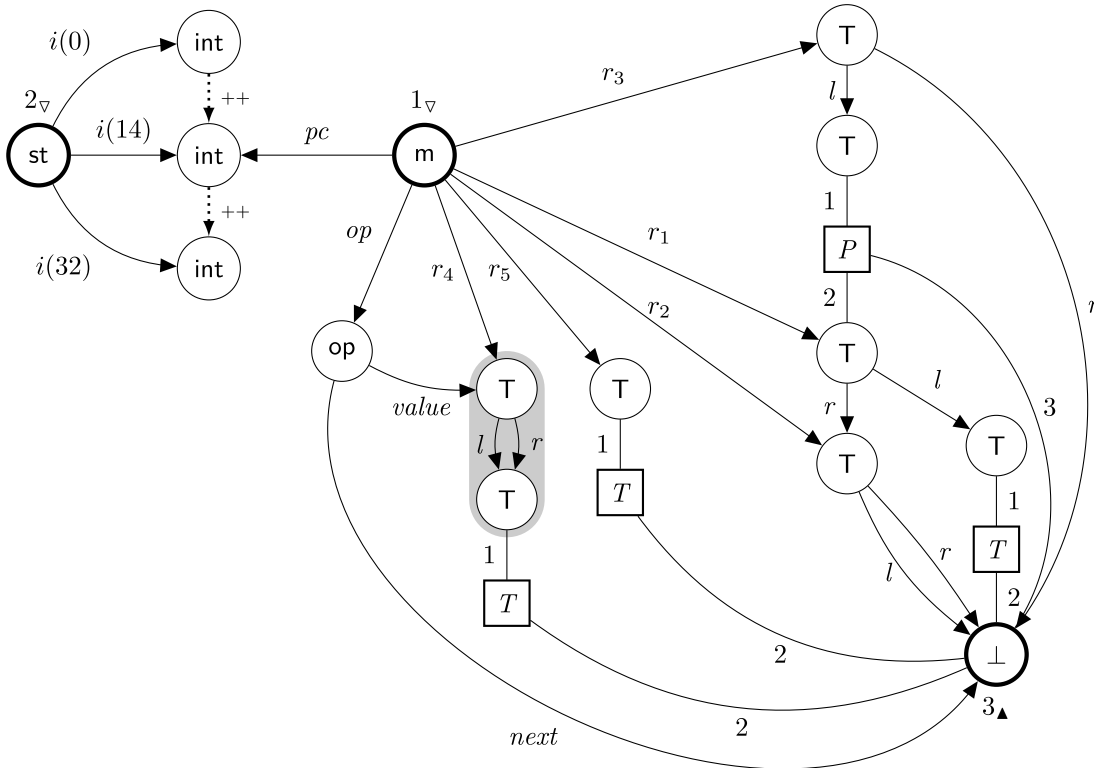


Figure 6.28: Intermediate state with sharing.

of evaluation trees. We introduced an anonymisation function abstracting from node identities in Section 6.2.8 in order to reduce the amount of evaluation trees that have to be considered. Indeed we can prove that for any combination of grammar and formula the amount of occurring anonymised evaluation trees is finite (Lemma 6.4 on page 159). This finiteness is essential and as we can compose evaluation trees for hyperedge replacements any FO-formula can be considered as a compositional property. That is we can determine a splitting grammar, where for each nonterminal its induced language has a unique evaluation.

The technique introduced for FO easily extends to MSO. That is we can decide MSO properties over graph languages defined by HRGs. We implemented the approach in a prototype tool for MSO over arbitrary HRGs as well as for the analysis of JVM states in Section 6.5. As mentioned in the introduction the classical approach to check MSO properties on HRGs is based on transforming MSO-formulae into finite tree automata which can be intersected with a grammar. This approach exploits the fact that HRGs describe graphs of bounded tree-width and for those automata can be constructed. Unfortunately the size of the resulting automata is non-elementary in the underlying tree-width. While the automata tend to get huge even for tree-width one they get unmanageable for higher tree-width. Therefore attempts to use the approach failed in practice [Mar06; CD12; CE12; Kep05]. In his thesis [Sog08], Soguet presents some

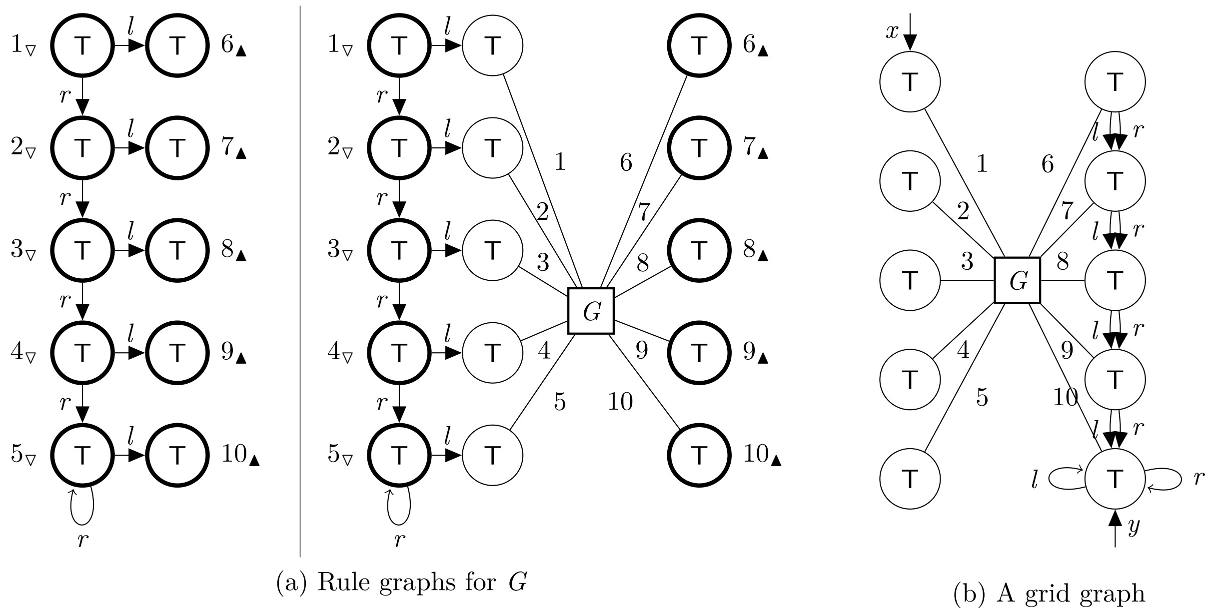
Problem	cw = 2	cw = 3
3-maxDeg	91	out
4-maxDeg	231	out
$x \overset{?}{\leadsto} y$	26	out
2-Color	11	57
3-Color	21	out
3-Cover	63	414
4-Cover	111	1037

Figure 6.29: Results by Soguet (MONA) [Sog08].

Problem	result	time
3-maxDeg	maybe	8 s
4-maxDeg	yes	43 s
$x \overset{?}{\leadsto} y$	yes	3770 s
2-Color	yes	316 s
3-Color	out	
3-Cover	maybe	31 s
4-Cover	maybe	180 s

Figure 6.30: Results for $5 \times n$ Grid.

experimental results generating automata for graphs with bounded clique-width, another measurement for decomposable graphs. In Figure 6.29 the size of some of the automata constructed by Soguet for clique-width two (cw=2) and three (cw=3) are given in number of states. He used MONA [Hen+95], a highly optimised tool for the translation of MSOs-formulae into automata. The presented problems are: none of the vertices exceeds a degree of i (i-maxDegree), there is a path from x to y ($x \overset{?}{\leadsto} y$), two and three colorability (2/3-Color), and three and four vertex cover set, i.e. there is a set of three (respectively four) vertices such that each vertex is incident with one of the sets.



(a) Rule graphs for G

(b) A grid graph

Figure 6.31: $5 \times n$ grids.

Note that the obtained automata are surprisingly small – e.g. two colorability with eleven states for clique-width two and 57 states for clique-width three. This is owed to the quite good minimisation techniques implemented in MONA. However, also MONA suffers from state space explosions – at least in intermediate states – which is the

reason for the numerous *out* entries. While the results are for bounded clique-width we have to expect similar problems considering bounded tree-width. In general, automata for bounded clique-width are easier to construct than the ones for equal tree-width. Unfortunately Soguet did neither provide the runtimes for the construction nor for any further experiments based on the resulting automata.

In Figure 6.30 we give our experimental results analysing an abstract $5 \times n$ grid. The analysed abstract graph is given in Figure 6.31(b) the corresponding grammar in Figure 6.31(a). The variables x, y used by the formulas for paths are depicted as labelled, directed edges without source vertices. Note that $5 \times n$ grids have a bounded tree-width of five and a bounded clique-width of seven. It seems to be hopeless to construct automata for this grammar, however, given the rather simple and regular graph language our approach is capable to obtain results for the majority of the examples.

Chapter 7

7 Quantified Model Checking

In Chapter 5 we generated state spaces of Java programs. Our aim is to analyse these state spaces in order to verify a variety of program properties. In Chapter 6 we did a first step by providing the logic MSOJ as a possibility to describe and check properties of single states. The state space of a program, however, does not just provide the reachable states of a program, but also its temporal behaviour, that is possible sequences of states, so-called paths (Definition 3.13 on page 64). Properties of single paths can be expressed in e.g. linear temporal logic (*LTL*) [BK08]. *LTL*-formulae are based on a set of *atomic propositions*, characterising single states. In Section 7.1 we introduce *LTL* for Java programs, where we use MSOJ-formulae as atomic propositions. There exists a wide variety of algorithms for *LTL* model checking. We present an on-the-fly approach by Grumberg *et. al.* [BCG95] in Section 7.1.1.

With *LTL* we can describe sequences of shape properties, however, it is not possible to describe the temporal behaviour of objects, e.g. we can describe that the heap of a program at any point in time consists of two separated lists, but we cannot express that objects remain in the same list along the execution. In Section 7.2 we introduce *quantified linear time logic* (*qLTL*), that extends LTL by existential and universal quantification over objects on the heap. These quantifications allow us to track the identity of objects over time. We use parameterised MSOJ-formulae to describe the behaviour of theses tracked objects, i.e. their position within the heap. In Section 7.2.1 we extend the model checking algorithm from Section 7.1.1 towards *qLTL*.

The algorithms and logics mentioned above are introduced for finite concrete state spaces. In Section 7.3 we then consider model checking for abstract state spaces. First abstract LTL model checking is introduced in Section 7.3.1. Abstract model checking for *qLTL*, considered in Section 7.3.2, is more challenging as the number of objects that have to be tracked is (potentially) unbounded.

Finally in Section 7.4 we discuss how to extend the approach for quantified LTL towards quantified $CTL^\star$ ($qCTL^\star$) as well as the benefits of the on-the-fly character of the presented model checking approach.

7.1 Linear Temporal Logic

One possibility to describe the temporal behaviour of a system is *linear temporal logic* (*LTL*) first proposed by Pnueli [Pnu77]. In *LTL* we describe properties of paths (Definition 3.13 on page 64). Besides the boolean operators *negation* ($\neg$), and *and* ($\wedge$) *LTL* contains the temporal operators *next* ($\bigcirc$) and *until* (U) and a set of atomic propositions. In our case atomic propositions are MSOJ-formulae φ which we put into square brackets $[\varphi]$ to visual separate them from the rest of the LTL-formula.

Definition 7.1 (*LTL* [BK08]):
Given a set $A \subseteq MSOJ_\Sigma$ of atomic propositions *with $\varphi \in A$: LTL or path formulae are recursively defined as*

$$\psi \quad := \quad true \;\big|\; [\varphi] \;\big|\; \neg\psi \;\big|\; \psi \wedge \psi \;\big|\; \bigcirc\psi \;\big|\; \psi\,\mathsf{U}\,\psi$$

By LTL_A we denote the set of all LTL formulae over the atomic propositions A.

In the base cases the LTL-formula either consists of a single true or of an MSOJ-formula $[\varphi]$. The LTL-formula ψ = true is fulfilled by any path, while path π fulfils formula $\psi = [\varphi]$ if and only if the first state of the path fulfils $\varphi \in MSOJ\Sigma$, i.e. $\pi[1] \models \varphi$ (Figure 7.1(a)). A formula $\bigcirc\psi$ is fulfilled by π if the path $\pi[2\ldots]$, starting at the second position of the path π fulfils ψ, i.e. $\pi[2\ldots] \models \psi$ (Figure 7.1(b)). A formula $\psi_1 \,\mathsf{U}\, \psi_2$ is fulfilled if there exists $i \in \mathbb{N}$ such that the path $\pi[i\ldots] \models \psi_2$ and for any position $j < i$ it holds that $\pi[j\ldots] \models \psi_1$ (Figure 7.1(c)).

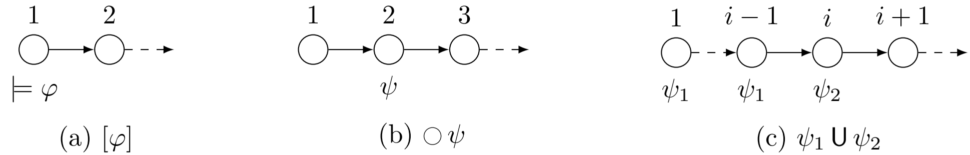


Figure 7.1: Temporal operators.

We define the semantics of a *LTL* formula by a model relation. Given $\psi \in LTL_{\mathrm{MSOJ}_\Sigma}$, a concrete transition system $\mathcal{T} \in TS_{T_\Sigma}$ (Definition 3.12) and a path $\pi \in Path_\mathcal{T}$, the model relation $\models$ for path formulae is defined by:

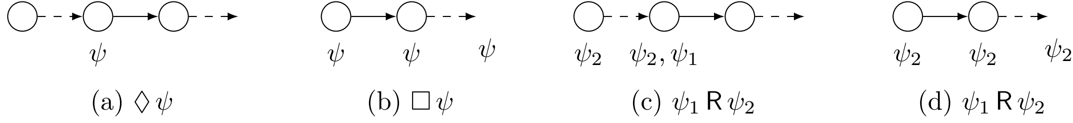
$$\begin{array}{lll}
\pi \models \text{true} & & \\
\pi \models [\varphi] & \textit{iff} & \pi[1] \models \varphi \\
\pi \models \neg\psi & \textit{iff} & \textit{not}\ \pi \models \psi \\
\pi \models \psi_1 \wedge \psi_2 & \textit{iff} & (\pi \models \psi_1)\ \textit{and}\ (\pi \models \psi_2) \\
\pi \models \bigcirc\psi & \textit{iff} & \pi[2\ldots] \models \psi \\
\pi \models \psi_1 \,\mathsf{U}\, \psi_2 & \textit{iff} & \exists i \geq 1.\, (\pi[i\ldots] \models \psi_2 \wedge (\forall 1 \leq k < i.\, \pi[k\ldots] \models \psi_1))
\end{array}$$

Given a state $s \in S_{\mathcal{T}}$, we write $s \models \psi$ if for all $\pi \in Path_s$ it holds that $\pi \models \psi$ and $\mathcal{T} \models \psi$ if for all $\pi \in Path_{\mathcal{T}}$ it holds that $\pi \models \psi$.

There are additional commonly used operators that can be expressed by the ones given in the definition:

$$
\begin{array}{lll l}
\text{false} & := & \neg\text{true} & \textit{logical false} \\
\psi_1 \vee \psi_2 & := & \neg(\neg\psi_1 \wedge \neg\psi_2) & \textit{logical or} \\
\psi_1 \rightarrow \psi_2 & := & \neg\psi_1 \vee \psi_2 & \textit{logical implication} \\
\psi_1 \leftrightarrow \psi_2 & := & (\psi_1 \rightarrow \psi_2) \wedge (\psi_2 \rightarrow \psi_1) & \textit{logical equivalence} \\
\Diamond\, \psi & := & \text{true}\, \mathsf{U}\, \psi & \textit{temporal finally} \\
\Box\, \psi & := & \neg\Diamond\, \neg\psi & \textit{temporal globally} \\
\psi_1\, \mathsf{R}\, \psi_2 & := & \neg(\neg\psi_1\, \mathsf{U}\, \neg\psi_2) & \textit{temporal release}
\end{array}
$$

The semantics of the temporal operators finally, global, and release is depicted in Figure 7.2.



(a) $\Diamond\, \psi$ (b) $\Box\, \psi$ (c) $\psi_1\, \mathsf{R}\, \psi_2$ (d) $\psi_1\, \mathsf{R}\, \psi_2$

Figure 7.2: Additional temporal operators.

Definition 7.2 (Positive normal form [BK08]):
Given a set of atomic propositions $A \subseteq MSOJ_{\Sigma}$, LTL formulae in positive normal form *are formed regarding to the following grammar, with arbitrary $\varphi \in A$:*

$$\psi \;:=\; \mathit{true} \;\big|\; \mathit{false} \;\big|\; [\varphi] \;\big|\; \neg[\varphi] \;\big|\; \psi_1 \wedge \psi_2 \;\big|\; \psi_1 \vee \psi_2 \;\big|\; \bigcirc\, \psi \;\big|\; \psi_1\, \mathit{U}\, \psi_2 \;\big|\; \psi_1\, \mathit{R}\, \psi_2$$

Note that in positive normal form negations are only allowed in front of atomic propositions. However, for any *LTL* formula there exists an equivalent one in positive normal form [BK08]. That is because for each operator there exists an operator that represents the negation, allowing to transform any *LTL* formula into positive normal form:

$$
\begin{array}{lll}
\neg\neg\psi & = & \psi \\
\neg\text{false} & = & \text{true} \\
\neg(\psi_1 \wedge \psi_2) & = & \neg\psi_1 \vee \neg\psi_2 \\
\neg\bigcirc\, \psi & = & \bigcirc\, \neg\psi \\
\neg(\psi_1\, \mathsf{U}\, \psi_2) & = & \neg\psi_1\, \mathsf{R}\, \psi_2
\end{array}
$$

Example 7.1: LTL formulae
In Figure 7.3(a) an algorithm is given that reverses doubly linked lists. The class definitions for doubly linked lists are given in Figure 7.3(b). They are the same as in Example 6.15 on page 174. The algorithm reverses doubly-linked lists by interchanging the next and previous pointer of each list element as well as the head and the tail pointer of the list.

```
void reverse(DLList list){
  ListElement cur = list.head;
  while(cur != list.tail){
    ListElement tmp = cur.next;
    cur.next = cur.previous;
    cur.previous = tmp;
    cur = cur.previous;
  }
  cur.next = cur.previous;
  cur.previous = null;
  list.tail = head;
  list.head = cur;
}
```

(a) Reverse Algorithm

```
public class DLList{
  ListElement head, tail;

}


public class ListElement{
  ListElement next, previous;
  DLList list;
}
```

(b) Class definition

Figure 7.3: Reverse doubly linked list algorithm.

The start configuration for calling the reverse-method with parameter list pointing to a doubly linked list with two elements is given in Figure 7.4. The state space induced by this start configuration is depicted in Figure 7.5. Note that the JVM states are partially given, namely the part of the heap that we want to focus on in this example, omitting the representation of the method stack and static values.

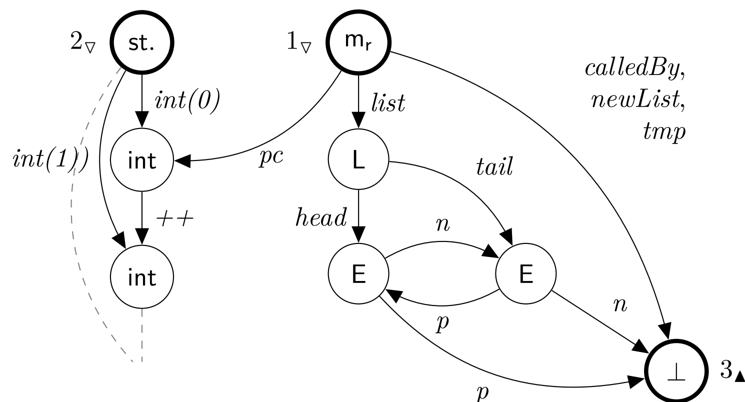


Figure 7.4: Concrete start configuration - list with two elements

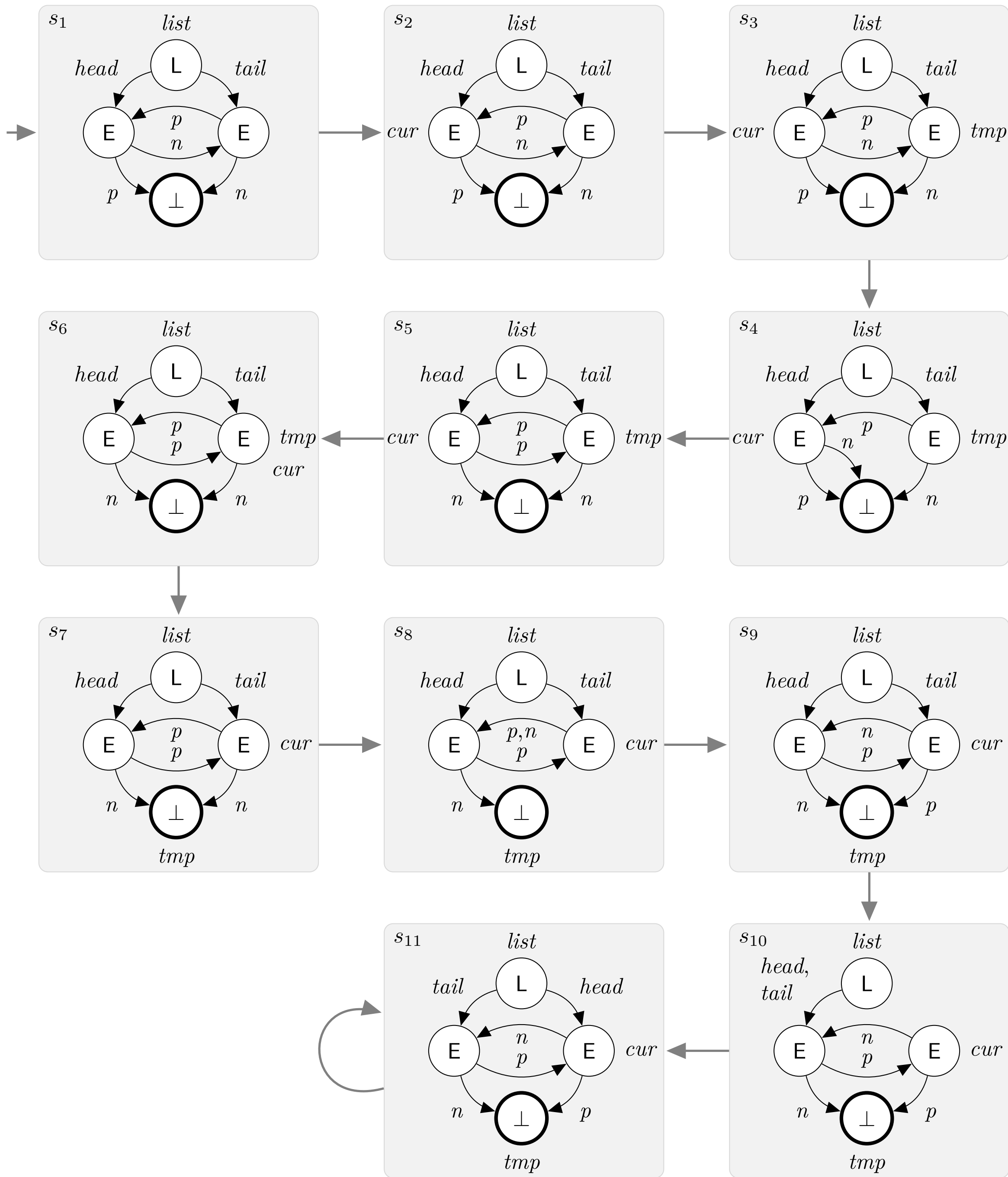


Figure 7.5: State space for doubly-linked list reversal.

We express two properties by LTL-formulae.

The first property is that the head of the list is always reachable from the tail.

$$\Box\,[\exists l, h, t \in \mathsf{ListElement}.\ (\,\mathit{var}(\mathit{list}) = l \wedge l.\mathit{head} = h \wedge l.\mathit{tail} = t \wedge t \overset{?}{\rightsquigarrow} h\,)\,]$$

The second property states that from some point on the list-elements form valid doubly linked lists. The MSOJ-formula φ_{dll} can be found in Example 6.15 on page 174.

$$\Diamond\,\Box\,[\varphi_{dll}]$$

The state space given in Figure 7.5 fulfils both formulae.

7.1.1 *LTL* Model Checking

To decide the model relation for a given (concrete) transition system and an *LTL* formula in positive normal form, Grumberg *et. al.* [BCG95] introduced an on-the-fly model checking algorithm. In this section we present their algorithm that we extend in Section 7.2.1. The definitions in this section are taken from Grumberg *et. al.* [BCG95] and adapted to our notation.

The algorithm is based on the tableaux technique [Dam92]. Based on a set of tableaux rules a *proof structure* is constructed. A proof structure is a directed graph, where the vertices are *assertions* of the form $s \vdash \Phi$, with s a state and Φ a set of *LTL* formulae. Semantically an assertion $s \vdash \Phi$ holds if $s \models \bigvee_{\psi\in\Phi} \psi$. We use $\lambda, \lambda_1, \lambda_2, \ldots$ to denote assertions and Λ to denote the set of all assertions. The following set of tableaux rules defines the edge relation, i.e. there is an edge from λ_1 to λ_2 if there exists a tableaux rule $\dfrac{\lambda_1}{\lambda_2}$.

$$(R^{\models})\ \frac{s \vdash \Phi \cup \{[a]\}}{\text{true}}\ \text{if } s \models a, \text{ and }\ (R^{\not\models})\ \frac{s \vdash \Phi \cup \{[a]\}}{s \vdash \Phi}\ \text{if } s \not\models a$$

$$(R^{\vee})\ \frac{s \vdash \Phi \cup \{\psi_1 \vee \psi_2\}}{s \vdash \Phi \cup \{\psi_1\} \cup \{\psi_2\}} \qquad (R^{\wedge})\ \frac{s \vdash \Phi \cup \{\psi_1 \wedge \psi_2\}}{s \vdash \Phi \cup \{\psi_1\} \qquad s \vdash \Phi \cup \{\psi_2\}}$$

$$(R^{\mathsf{U}})\ \frac{s \vdash \Phi \cup \{\psi_1 \,\mathsf{U}\, \psi_2\}}{s \vdash \Phi \cup \{\psi_2, \psi_1\} \qquad s \vdash \Phi \cup \{\psi_2, \bigcirc(\psi_1 \,\mathsf{U}\, \psi_2)\}}$$

$$(R^{\mathsf{R}})\ \frac{s \vdash \Phi \cup \{\psi_1 \,\mathsf{R}\, \psi_2\}}{s \vdash \Phi \cup \{\psi_2\} \qquad s \vdash \Phi \cup \{\psi_1, \bigcirc(\psi_1 \,\mathsf{R}\, \psi_2)\}}$$

$$(R^{\bigcirc})\ \frac{s \vdash \{\bigcirc\psi_1, \ldots, \bigcirc\psi_n\}}{s_1 \vdash \{\psi_1, \ldots, \psi_n\} \qquad \ldots \qquad s_m \vdash \{\psi_1, \ldots, \psi_n\}}$$

Where $\{s_1, \dots s_m\} = \{s' \mid s \rhd s'\}$ are the successors of s. Note that we alter the state of an assertion only if all of its formulae start with a next-operator. That is, a state switch is always performed simultaneously for all formulae of a set Φ. The given rules model the semantics of *LTL*. The rules for the binary operators U and R are based on their recursive characterisations [BK08]:

$$\psi_1 \mathsf{U} \psi_2 = \psi_2 \vee (\psi_1 \wedge \bigcirc (\psi_1 \mathsf{U} \psi_2)) \qquad \text{and} \qquad \psi_1 \mathsf{R} \psi_2 = \psi_2 \wedge (\psi_1 \vee \bigcirc (\psi_1 \mathsf{R} \psi_2)).$$

Definition 7.3 (Proof structure [BCG95]):
A proof structure *for $\lambda \in \Lambda$ is a tuple (V, E) with $V \subseteq (\Lambda \cup true)$ and $E \subseteq V \times V$, such that for any $\lambda' \in V$ it holds that λ' is reachable from λ and that the successors of λ' are the ones that result from applying some of the rules, i.e.*

$$(\lambda_1, \lambda_2) \in E \quad \textit{iff} \quad \frac{\lambda_1}{\lambda_2 \quad s \dots}$$

An assertion can be seen as a goal of the verification. Given the assertion $s \vdash \Phi$ we aim in proving that $s \models \bigvee_{\psi \in \Phi} \psi$. To prove the goal we split it into a set of subgoals. The subgoals for each assertion are defined via the tableaux rules. In order to verify an assertion we have to verify each of its subgoals. For an assertion $\lambda \in \Lambda$ the corresponding proof structure contains all subgoals of λ. If the proof structure contains an empty assertion $(s \vdash \emptyset)$, then this subgoal is not fulfilled and thus neither is λ.

Note that cyclic dependencies could only occur due to the R^{U} and R^{R} rule, as all other rules reduce the size of one of the formulae. Any cycle in the proof structure represents an *infinite path* of the underling state space for which the left side of some U (respectively R) operand is fulfilled for each state, while none of the states fulfil the right hand side. Thus this infinite path fulfils the definition of R but violates the definition of U. That is a cycle in the proof structure, that origins from successively applying rule R^{U} falsifies the start assertion. On the other hand if the cycle origins from applying R^{R} rules, the partial subgoal is fulfilled and we call the induced infinite path *successful*.

These observations are reflected by the following notations and definitions. Given a proof structure (V, E): [BCG95]:

1. $\lambda \in V$ is a leaf of the proof structure if there is no $\lambda' \in V$ with $(\lambda, \lambda') \in E$. We call a leaf λ successful if $\lambda =$ true.
2. We call an infinite path $\lambda_1 \lambda_2 \dots$ in (V, E) successful iff there exists a position $i \in \mathbb{N}$ with $\psi_1 \mathsf{R} \psi_2 \in \lambda_i$ and for all $j \geq i$ it holds that $\psi_2 \notin \lambda_j$.
3. We call (V, E) successful if every leaf as well as every of its infinite paths is successful.

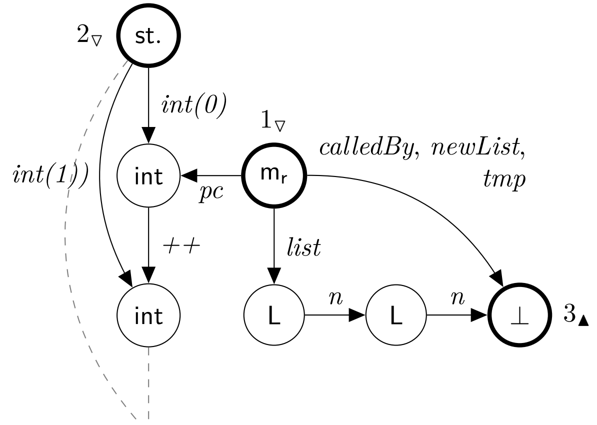
Theorem 7.1 (Correctness of the tableaux construction [BCG95]):
Given a concrete transition system $\mathcal{T}$ with $s \in S_{\mathcal{T}}$ and an LTL formula ψ. Let (V, E) be the proof structure for $s \vdash \{\psi\}$. Then it holds that $s \models \psi$ iff (V, E) is successful.

Example 7.2: LTL properties of state spaces
In Figure 7.6(a) an algorithm for reversing singly linked lists is given and in Figure 7.6(b) a corresponding start state s_1 for reversing a list with two elements. Figure 7.7 depicts the resulting concrete state space.

```
class List{
  List next;

  List reverse(List list){
    List newList = null;
    while(list != null){
      List tmp = list;
      list = list.next;
      tmp.next = newList;
      newList = tmp;
    }
    return newList;
  }
}
```

(a) Reverse algorithm



(b) Start configuration s_1

Figure 7.6: Algorithm for reverse singly-linked lists.

The property "eventually the tmp variable points to the same object as the variable list and up to this point in time is set to null " can be expressed by the LTL-formula:

$$\psi = [\,\exists x \in \bot.\ var(tmp) = x\,]\ \mathsf{U}\,[\,\exists x \in \mathsf{List}.\ var(tmp) = x \wedge var(list) = x\,]$$

We use $\psi = [t = \bot]\,\mathsf{U}\,[t = l]$ as a shorthand for this formula. In Figure 7.8 the proof structure for $s_1 \vdash \{[t = \bot]\,\mathsf{U}\,[t = l]\}$ is given. Note that there are no infinite paths and the only leaf is labelled true. We can thus conclude that $\mathcal{T} \models \psi$.

LTL-formulae are based on atomic propositions, i.e. MSOJ-sentences describing the relation between vertices of a single heap state. As a result statements about objects (represented by vertices) are restricted to single heap states, i.e. to one point in time. E.g. in *LTL* we can express that for any state of a path the objects form two separated lists, but not that along the path each object remains in the same list. If an object occurs in two succeeding states, then in both states there exists a vertex representing this object. That is there is an object-identity relation between vertices of succeeding states. Tracking object identities gives us the ability to express temporal properties of objects.

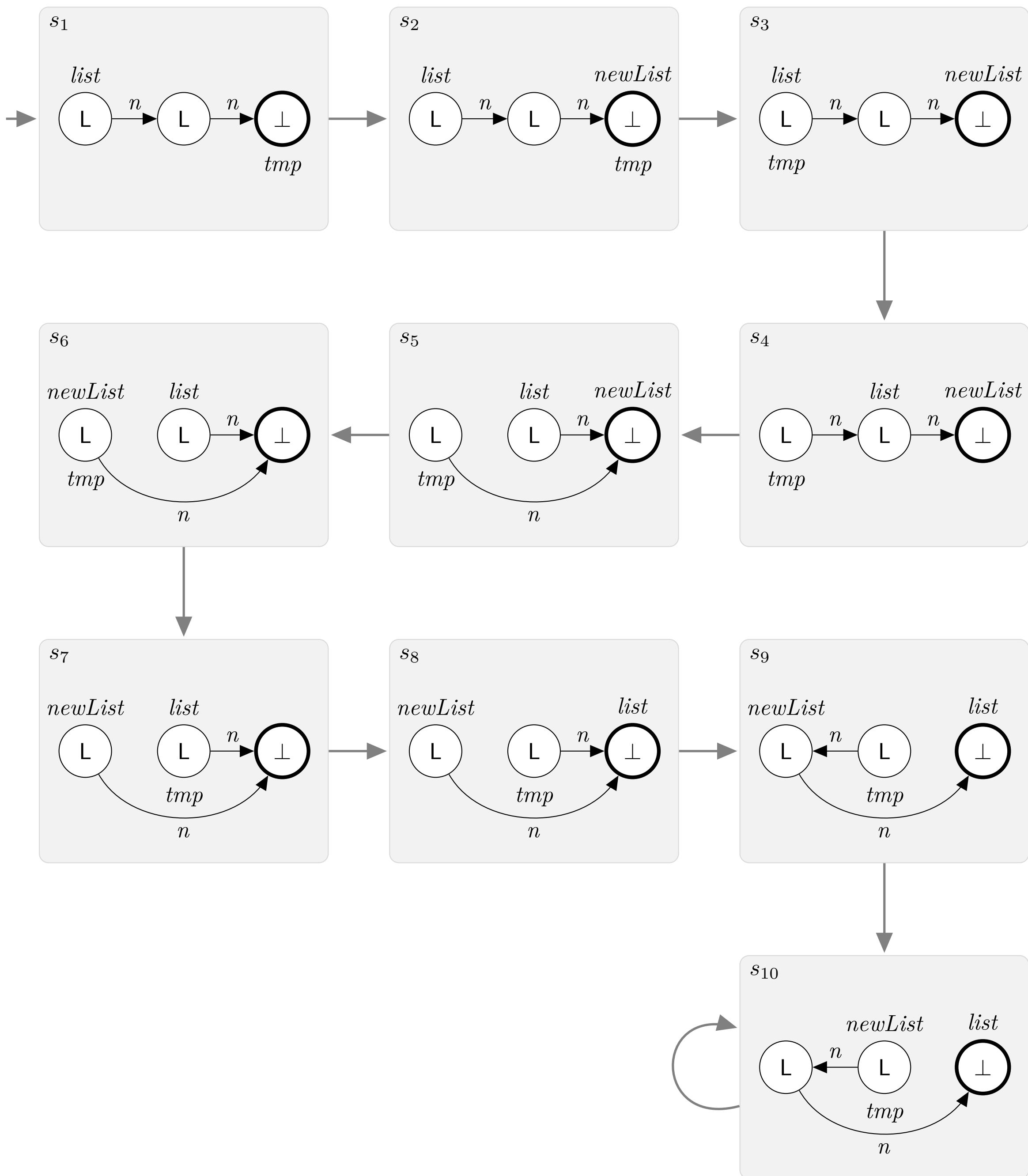


Figure 7.7: State space for the singly-linked list reversal.

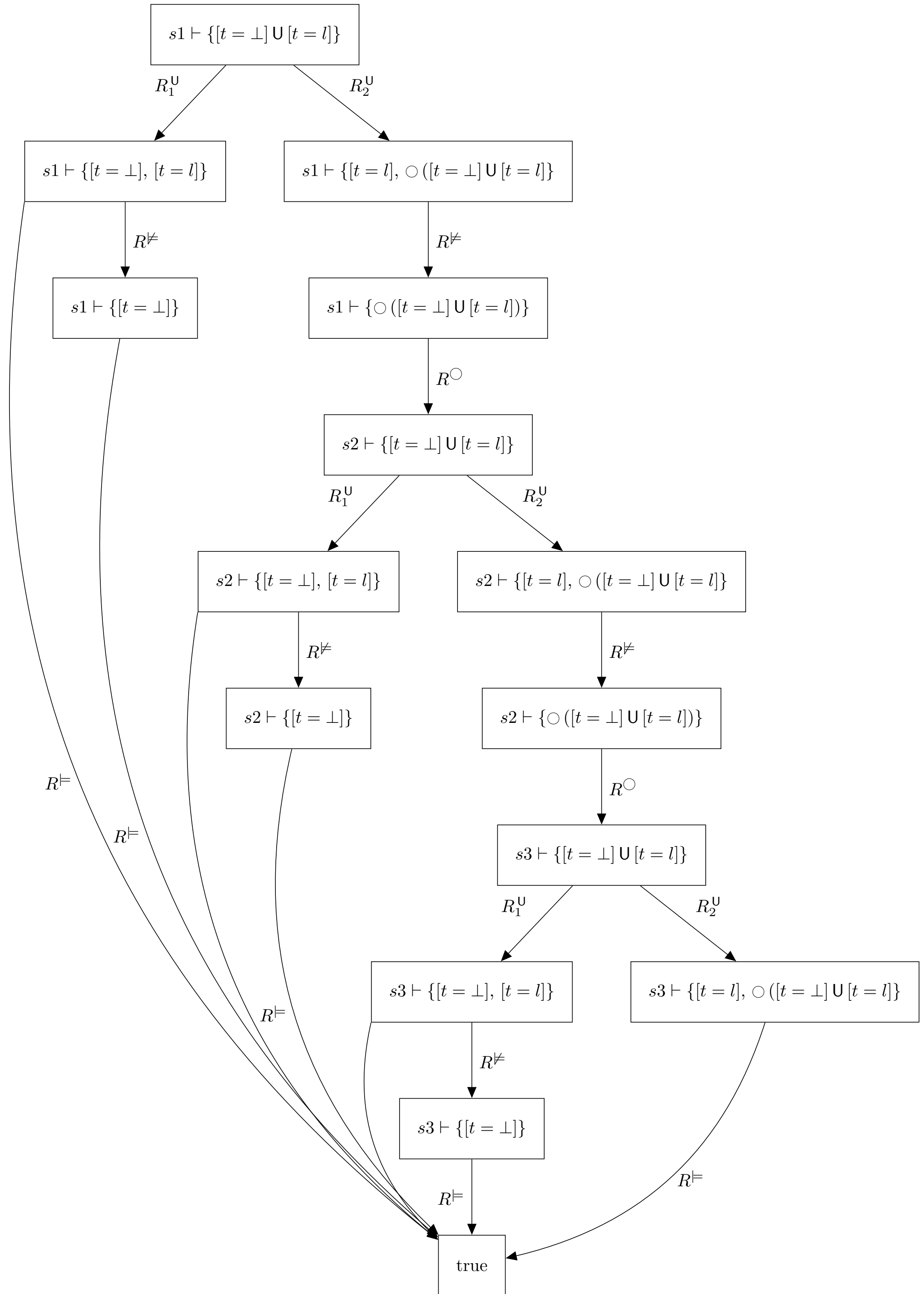


Figure 7.8: Proof structure for $s1 \vdash \{[t = \bot] \mathsf{U} [t = l]\}$

Example 7.3: Object properties

For the list reversal algorithm from Example 7.2, we could express that any object that is an element of the input-list should also be element of the resulting list. To formalize this property, we need to track object identities. Given the identity of an object on the heap that we name x we use the following parametric LTL-formula:

$$\psi(x) \;=\; [\exists s \in \mathsf{List}.\ var(list) = s \wedge s \overset{n}{\rightsquigarrow} x] \;\rightarrow\; \Diamond\Box\,[\exists s \in \mathsf{List}.\ var(list) = s \wedge s \overset{n}{\rightsquigarrow} x]$$

To check that the algorithm indeed reverses the order of objects we check for any pair of objects x, y that

$$\psi(x, y) \;=\; [x \overset{n}{\rightsquigarrow} y] \;\rightarrow\; \Diamond\Box\,[y \overset{n}{\rightsquigarrow} x]$$

We need to check these formulae for any object x, pair of objects x, y respectively, that exists in the start configuration.

In order to track the identity of objects we introduce a marking of vertices [HNR10]. This marking names objects that then can be identified by their name. Let M be a finite set of names that we use as markers. Considering the above example we get $M = \{x, y\}$. We achieve the marking by introducing an additional vertex ***marking*** (a fourth external vertex) with one outgoing edge for each marker from M. We name an object by directing the corresponding marker edge to the vertex representing it. Figure 7.9 depicts a marked version of the configuration given in Figure 7.6(b). Here the first element of the list is marked as x and the second as y.

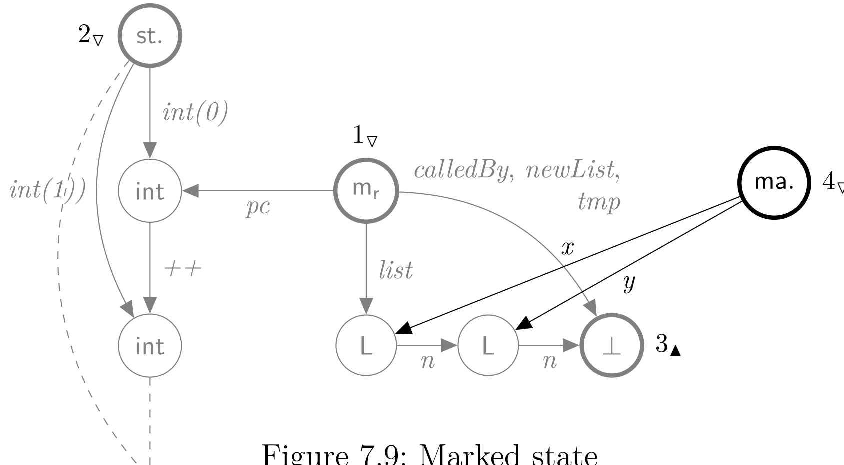


Figure 7.9: Marked state

The marking is not modified by the transition relation and thus allows us to identify object identities between succeeding states. E.g. if we execute list .next = tmp on the marked configuration from Figure 7.9 we get the marked configuration depicted in Figure 7.10. The object represented by the vertex attached to the x-edge is the same for both configurations. The same holds for the vertices attached to the y-edge. While $x \overset{n}{\rightsquigarrow} y$ holds for the first configuration, it does not hold for the second.

For a given (marked) JVM state H and a set of markers M, we denote the set of all (re-)marked versions of H by $mark_M(H)$. Given a marked state H we set mark

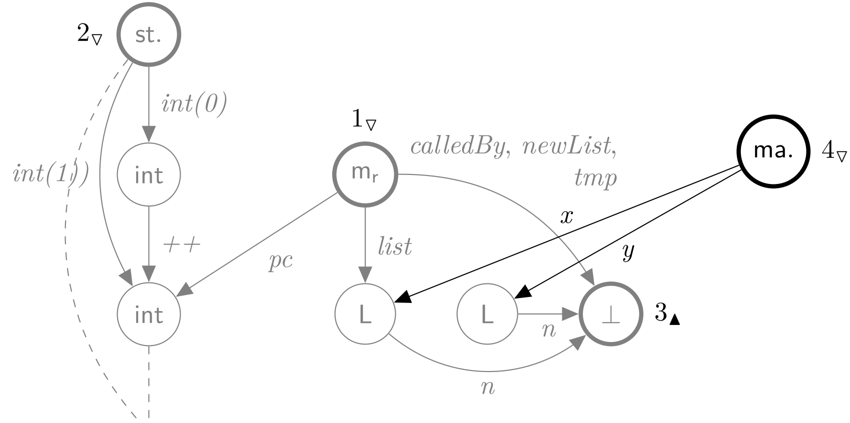


Figure 7.10: A successor of the state given in Figure 7.9

$x \in M$ to vertex v by $setMark(H, x, v)$. In addition we use the notation $mark(H, x) = \{setMark(H, x, v) \mid v \in V_H\}$.

Replacing the start configuration $s \in \mathrm{HC}_\Sigma$ of a transition system by the set $mark(s)$ we get a state space of marked configurations.

Example 7.4: State space with markings
In Figure 7.11, a version of the state space from Figure 7.7 with markings is given. Marking the start configuration s_1 by markers from $\{x, y\}$ results in the set $mark_{\{x,y\}}(s_1) = \{s_1^a, s_1^b, s_1^c, s_1^d\}$ used as start configurations in Figure 7.11. Again we focus only on the heap-part of the representation and we depict the markings by small, labelled circles attached to the vertices. For each marked start state we get one connected component of states. We give only state s_1, s_5, and s_{10} of each set. Omitted states are represented by dashed transitions. Note that each of the four paths fulfils the formulae $\psi(x)$ and $\psi(x, y)$ given in Example 7.3.

By marking objects of the initial configuration we can track objects of the start configuration. However, objects can be generated and destroyed during runtime, thus it is necessary to track also objects that do not exists within the start configuration but are created during execution. In the following section we extend LTL towards *quantified LTL* that allows us to quantify arbitrarily over object identities.

7.2 Quantified LTL

Quantified LTL (*qLTL*) is an extension of LTL by quantification over heap-objects. That is, we extend LTL by operators for existential ($\exists x$) and universal ($\forall x$) quantification over objects on the current heap.

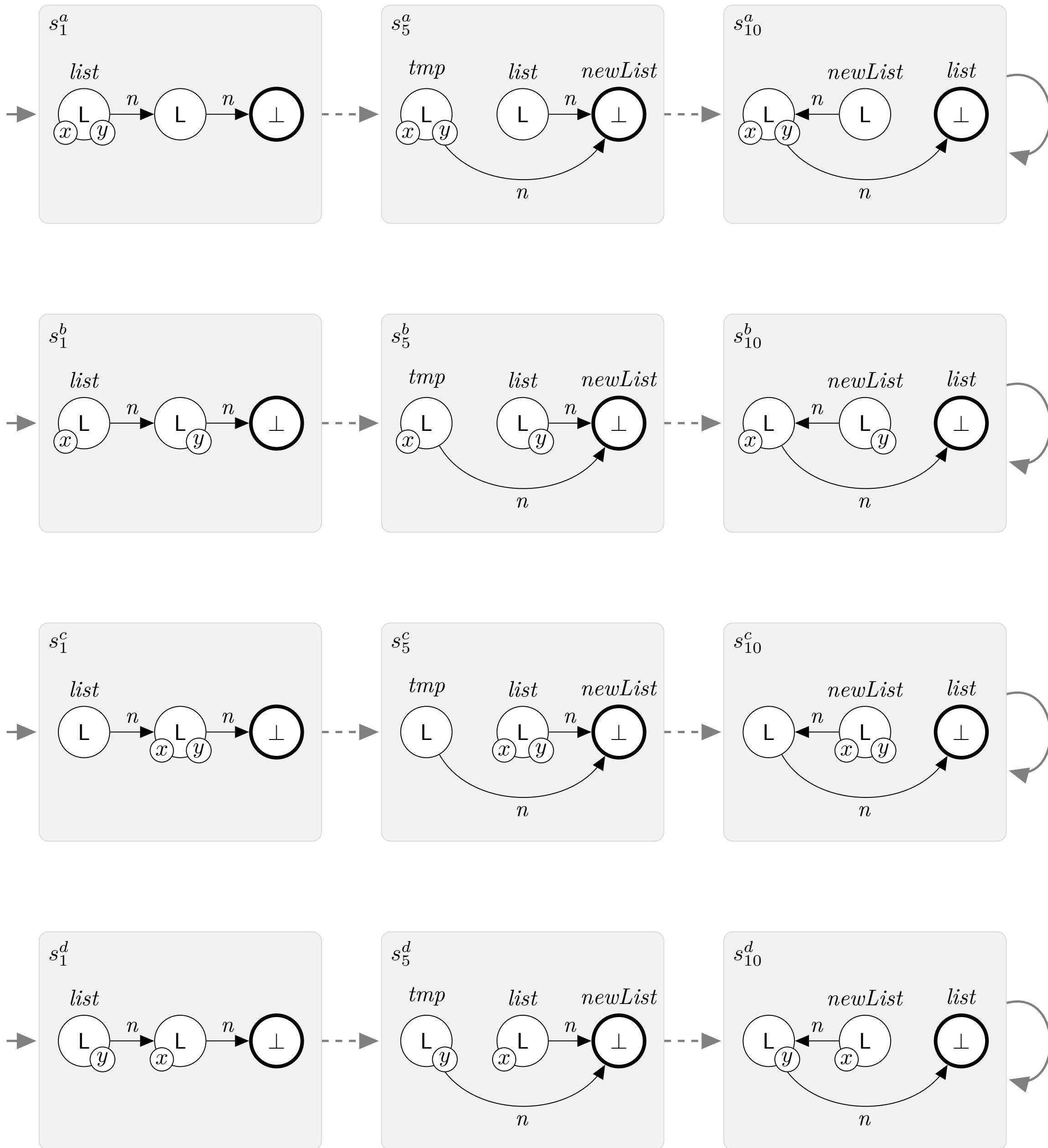


Figure 7.11: Marked version of the state space for the singly-list reversal.

Definition 7.4 ($qLTL$):
Given a set of atomic propositions $A \subseteq MSOJ_\Sigma$, and a set of markers M, quantified LTL (qLTL) formulae are formed according to the following grammar, where $\varphi \in A$ and $x \in M$ are arbitrary:

$$\psi \;:=\; true \;\big|\; [\varphi] \;\big|\; \neg\psi \;\big|\; \psi \wedge \psi \;\big|\; \bigcirc\psi \;\big|\; \psi\, \mathsf{U}\, \psi \;\big|\; \exists x.\, \psi \;\big|\; \forall x.\, \psi$$

We denotes the set of all qLTL formulae over the atomic propositions A and markers M by $qLTL_{A,M}$.

Using the *mark*-function (see page 191) we introduce *marked paths*, that we use to define the semantics of $qLTL$.

Definition 7.5 (Marked path):
Given a (marked) path $\pi = s_1\ s_2 \ldots \in \mathcal{T}_\triangleright$ and a set of markings M, a (re-)marked path for π is $\pi' = s'_1\ s'_2 \ldots$ with

$$(1)\ s'_i \in mark_M(s_i) \quad \text{and} \quad (2)\ s'_i \triangleright s'_{i+1} \qquad \text{for all } i \in \mathbb{N}.$$

For path π we denote the set of all (re-)marked paths by $mark(\pi)$ and we use $mark(\pi, x) = \{\pi' \in mark(\pi) \mid \pi'[1] \in mark(\pi[1], x)\}$.

The semantics of an $qLTL$-formula on a marked path is defined via the model relation $\models$ that is equivalent to the one for LTL-formulae up to the two new operators defined as follows:

$$\pi \models \exists x.\ \psi \quad \text{iff} \quad \text{there exists } \pi' \in mark(\pi, x) \text{ with } \pi' \models \psi$$
$$\pi \models \forall x.\ \psi \quad \text{iff} \quad \text{for all } \pi' \in mark(\pi, x)\ \pi' \models \psi \text{ holds}$$

As before for LTL we define a *positive normal form* for $qLTL$, where we use the equations $\neg\exists\, x.\, \psi = \forall\, x.\, \neg\psi$ and $\neg\forall\, x.\, \psi = \exists\, x.\, \neg\psi$ to transform arbitrary $qLTL$ formulae into positive normal form.

7.2.1 $qLTL$ Model Checking

We extend the tableaux based algorithm from Grumberg *et. al.* towards $qLTL$ formulae in positive normal form. We use marked state spaces, that are essentially an extension of state spaces by their marked paths.

Definition 7.6 (Marked state space):
Given a (concrete) start configuration $s \in HC_{T_\Sigma}$, a transition function $\rhd$ and a set M of markers, the marked state space *is a triple $(S, \rhd, \mu)$, where*

$$S = \{s' \in mark(s) \mid s \in S_{T_{\rhd,s}}\}$$
$$\mu \subseteq S \times M \times S \quad \textit{with } (s, m, s') \in \mu \textit{ iff } s' \in mark(s, m)$$

Example 7.5: Marked state space
Figure 7.12(b) depicts the marked version of the state space given in Figure 7.12(a) for the list reversal algorithm and the marks $M = \{x, y\}$. Note that the states are the same as the ones in Example 7.4 on page 192. That is because no objects are created during runtime. Between the separated paths from Example 7.4 there are transitions labelled by the markers x and y. These additional μ-transitions allow us to jump between the different marked versions while constructing the proof structure. Note that for each state s in the unmarked state space there is a set of marked states $mark(s)$ that form a component connected via μ-transitions.

To check a *qLTL* formula we build a proof structure that is similar to the one from Grumberg *et. al.* [BCG95]. The vertices of the structure are assertions of the form $s \vdash \Phi$, where s is an unmarked state and Φ is a set of tuples (s', ψ) with $s' \in mark(s)$ and $\psi \in qLTL$. Thus each of the formulae ψ, that can be seen as subgoals that have to be proven, is associated by a marking mapping logical variables to vertices. The tableaux rules that define the edges of the proof structure are as follows:

$$(R^{\models})\ \frac{s \vdash \Phi \cup \{(s', [a])\}}{\text{true}} \text{ if } s' \models a, \text{ and } \quad (R^{\not\models})\ \frac{s \vdash \Phi \cup \{(s', [a])\}}{s \vdash \Phi} \text{ if } s' \not\models a$$

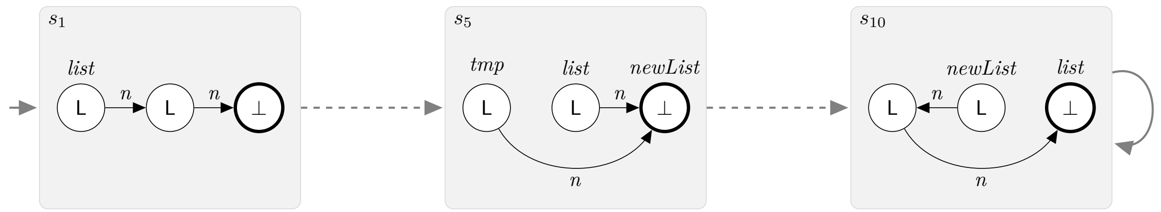
$$(R^{\vee})\ \frac{s \vdash \Phi \cup \{(s', \psi_1 \vee \psi_2)\}}{s \vdash \Phi \cup \{(s', \psi_1), (s', \psi_2)\}} \qquad (R^{\wedge})\ \frac{s \vdash \Phi \cup \{(s', \psi_1 \wedge \psi_2)\}}{s \vdash \Phi \cup \{(s', \psi_1)\} \qquad s \vdash \Phi \cup \{(s', \psi_2)\}}$$

$$(R^{\mathsf{U}})\ \frac{s \vdash \Phi \cup \{(s', \psi_1 \,\mathsf{U}\, \psi_2)\}}{s \vdash \Phi \cup \{(s', \psi_2), (s', \psi_1)\} \qquad s \vdash \Phi \cup \{(s', \psi_2), (s', \bigcirc (\psi_1 \,\mathsf{U}\, \psi_2))\}}$$

$$(R^{\mathsf{R}})\ \frac{s \vdash \Phi \cup \{(s', \psi_1 \,\mathsf{R}\, \psi_2)\}}{s \vdash \Phi \cup \{(s', \psi_2)\} \qquad s \vdash \Phi \cup \{(s', \psi_1), (s', \bigcirc (\psi_1 \,\mathsf{R}\, \psi_2))\}}$$

$$(R^{\bigcirc})\ \frac{s \vdash \{(s^1, \bigcirc \psi_1), \ldots, (s^n, \bigcirc \psi_n)\}}{s_1 \vdash \{(s_1^1, \psi_1), \ldots, (s_1^n, \psi_n)\} \qquad \ldots \qquad s_m \vdash \{(s_m^1, \psi_1), \ldots, (s_m^n, \psi_n)\}}$$

Where $\{s_1, \ldots s_m\} = \{s' \mid s \rhd s'\}$ are the successors of the unmarked state s and s_j^i is the successor of s^i ($s^i \rhd s_j^i$), that is a marked version of s_j ($s_j^i \in mark(s_j)$).



(a) Unmarked state space.

(b) Marked state space.

Figure 7.12: A state spaces and its marked version.

$$(R^{\forall x})\ \frac{s \vdash \Phi \cup \{(s', \forall x.\psi)\}}{s \vdash \Phi \cup \{(s^1, \psi)\} \qquad \ldots \qquad s \vdash \Phi \cup \{(s^n, \psi)\}}$$

$$(R^{\exists x})\ \frac{s \vdash \Phi \cup \{(s', \exists x.\psi)\}}{s \vdash \Phi \cup \{(s^1, \psi),\ \ldots\ (s^n, \psi)\}}$$

Where $\{s^1, \ldots s^n\} = \{s^\# \mid (s', x, s^\#) \in \mu\}$ are re-markings of s'. Note that for any re-markings $s^\#$ it holds that $s^\# \in mark(s)$, as $s' \in mark(s)$.

We extend the definition and notations for proof structures as introduced in Section 7.1.1 towards *qLTL* and marked state spaces. By adapting Theorem 7.1 we get the following theorem.

Theorem 7.2 (Correctness of the tableaux construction for *qLTL*):
Given a concrete transition system $\mathcal{T}$ with $s \in S_\mathcal{T}$ and an $qLTL_{MSOJ_\Sigma,M}$-formula ψ. Let (V, E) be the proof structure for $s \vdash \{\psi\}$. Then it holds that:

$$s \models \psi \text{ iff } (V, E) \text{ is successful.}$$

Theorem 7.2 can be proven analogous to Theorem 7.1, proved by Grumberg *et. al.* [BCG95]. The only necessary extension is to show that the subgoals defined by the rules $R^{\forall x}$ and $R^{\exists x}$ correspond to the semantics of the corresponding operators in *qLTL*. Which is obviously as by definition of μ it holds that $mark(s') = \{s^\# \mid (s', x, s^\#) \in \mu\}$, i.e. $\{s^\# \mid (s', x, s^\#)\}$ are the start states of the possible remarked paths.

7.3 Abstract Model Checking

In the previous sections we introduced *LTL* and *qLTL* model checking for concrete state spaces. As discussed before in many cases the considered state spaces are either infinite (due to an infinite number of start states or unbounded creation of objects during runtime) or unmanageable in size. In these cases we want to abstract the state space - resulting in a substantial reduction of states. How to obtain abstract state spaces was subject of Section 3.5 and we considered abstraction of JVM state spaces in Section 5.4.3.

In this section we now consider model checking for abstract state spaces, i.e. given an abstract state space we want to know if any represented concrete state space fulfils some temporal property. In Section 7.3.1 we start with LTL model checking. The results for general LTL model checking for over approximations can easily be adapted to our approach. In Section 7.3.2 we then consider *qLTL* model checking which is more complicated as one abstract state represents multiple concrete ones and marking abstract states corresponds to marking (potentially) infinitely many concrete states.

7.3.1 LTL Model Checking

To check $\psi \in LTL_A$ on an abstract state space $\mathcal{T}$ we have to be able to evaluate the formulae from the set $A \subset \mathrm{MSOJ}_\Sigma$ over the (abstract) states $s \in S_\mathcal{T}$. If we evaluate $\varphi \in A$ over $L(s)$ the result is a set in $\mathcal{P}(\{\mathrm{true}, \mathrm{false}\})$, reflecting if none, all or some of the represented states fulfil φ. Whenever the result is $\{true, false\}$ it is not clear how to handle this result within the semantics of LTL. Indeed we can neither handle it as true nor as false. However, if we get a $\{true\}$ or $\{false\}$ result for each combination of states and formulae, we can evaluate ψ on the abstract state space using the algorithm from Section 7.1.1.

Theorem 7.3 (LTL inclusion):
Given $A \subset MSOJ_\Sigma$ and $\mathcal{T}_1, \mathcal{T}_2 \in TS_\Sigma$ with $\mathcal{T}_2 \succeq \mathcal{T}_1$, such that for any $\varphi \in A$ and $s \in S_{\mathcal{T}_1} \cup S_{\mathcal{T}_2}$ it holds that either $\forall s' \in L(s).\ s' \models \varphi$ or $\forall s' \in L(s).\ s' \not\models \varphi$ then for any $\psi \in LTL_A$ it holds that

$$\mathcal{T}_2 \models \psi \quad \textit{implies} \quad \mathcal{T}_1 \models \psi$$

Theorem 7.3 can be proven as follows. From Definition 3.14 on page 64 and the unique evaluation for formulae from A we directly get the inclusion of *traces* from $\mathcal{T}_1$ in $\mathcal{T}_2$ which in turn implies LTL inclusion [BK08].

Remember that for any $G \in \mathrm{DSG}_\Sigma$ and $A \subset \mathrm{MSO}_\Sigma$ there is a grammar $G' \approx G$ such that any $H \in \mathrm{HG}_\Sigma$ and $\varphi \in A$ evaluates to a unique value. The construction of this grammar was subject of Section 6.3 (Theorem 6.5 on page 168). That is, for any grammar $G \in \mathrm{DSG}_\Sigma$ and $A \subset \mathrm{MSOJ}_\Sigma$, there is a grammar $G_A \approx G$ that we can use to generate an abstract state space on which we can apply the LTL model checking algorithm from Section 7.1.1.

7.3.2 *qLTL* Model Checking

To mark an abstract state it is not sufficient to consider the vertices of the hypergraph, as there are more objects abstracted within nonterminal edges. To include also these abstracted objects into the marking we need to concretise them first. Grammars in LGNF (Definition 3.8 on page 52) allow us to concretise the successors of external vertices, but not of arbitrary ones. Therefore we define a special form of grammars, called *marking sets*, that are used to mark abstracted vertices of configurations.

Definition 7.7 (Marking sets):
A marking set is a set of triples (X, H, v) where $X \in N_\Sigma$, $H \in HG_\Sigma$ with $|\mathrm{ext}_H| = \mathrm{rk}(X)$ and $v \in (V_H \setminus [\mathrm{ext}_H])$.

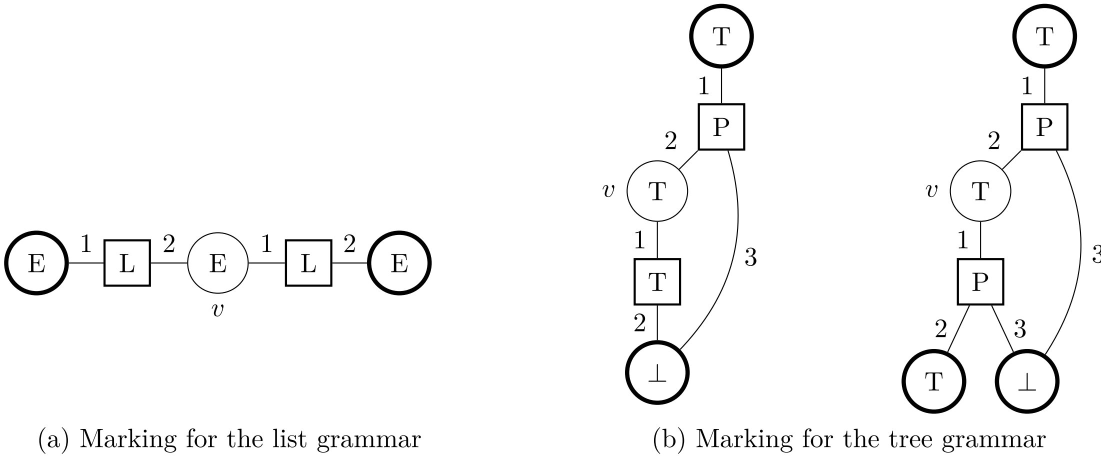
Marking sets are an extension of HRGs as defined in Definition 2.7 on page 20, where for each rule graph one vertex is qualified. Marking set M is called a marking for grammar $G \in \mathrm{HRG}_\Sigma$ if for any $X \in N_\Sigma$ it holds that

$$\{(H, v) \mid (X, H', v) \in M,\ H \in L_G(H')\} \quad = \quad \{(H, v) \mid H \in L_G(X),\ v \in V_H\}$$

Example 7.6: Marking sets

The marking set for the list grammar consists of a unique triple (L, H, v). Graph H and its vertex v are given in Figure 7.13(a). The vertex v here is neither the first nor the last vertex of the list but some vertex in between.

For the tree grammar as defined in Section 5.5 we get one triple for B and one for P. Both together with their specific vertex are given in Figure 7.13(b).



(a) Marking for the list grammar

(b) Marking for the tree grammar

Figure 7.13: Marking for grammars.

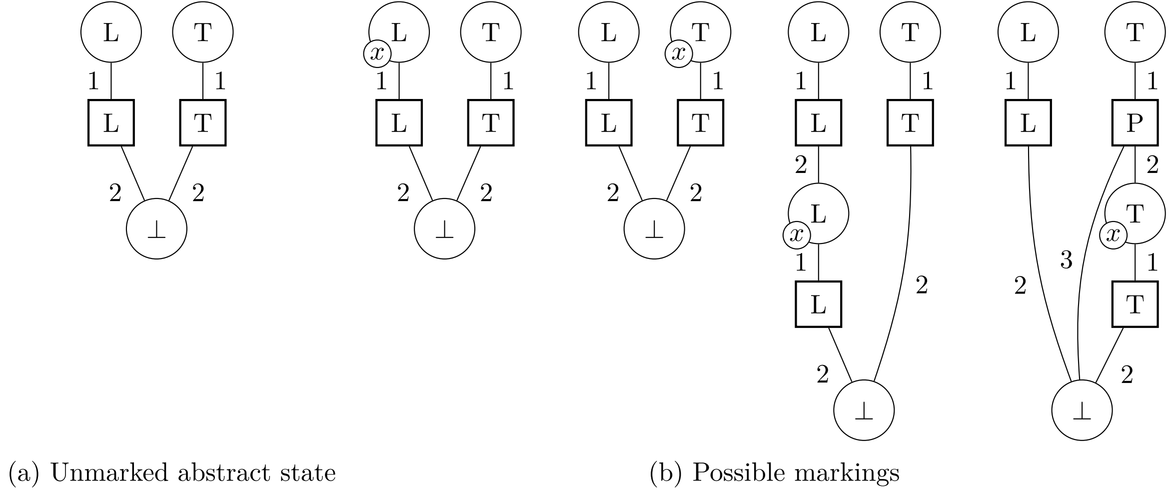
Given M, a marking for $G \in \mathrm{HRG}_\Sigma$ we can easily determine the set of marked abstract states that represents all possible marked concretisations. We either set the mark to one of the concrete vertices of the abstract configuration or to one of the abstracted vertices, represented by some nonterminal, which we concretise via the marking set.

$$\begin{array}{rcl} \mathit{mark}_\forall(H, x) & = & \{\mathit{setMark}(H, x, v) \mid v \in V_H\} \\ & \cup & \{\mathit{setMark}([H[R/e], x, v) \mid (X, R, v) \in M,\ \mathrm{lab}(e) = X\} \end{array}$$

Example 7.7:

In Figure 7.14(a) an abstract state is depicted. The possible abstract markings for this state are given in Figure 7.14(b).

That is given a single marker x we get four marked states. If we add a second marker y to these $\{x\}$-marked states we get for the first and second one four $\{x, y\}$-marked states and six $\{x, y\}$-marked states for the third and fourth. That is there are 20



(a) Unmarked abstract state (b) Possible markings

Figure 7.14: Marking of abstract states.

$\{x, y\}$-marked states for the unmarked abstract state from Figure 7.14(a). In general the number of marked states is exponentially in the number of marks.

For $H \in \mathrm{HRG}_\Sigma$ the $mark_\forall$ function returns a set of configurations representing all possible markings of all graphs in $L(H)$, i.e.

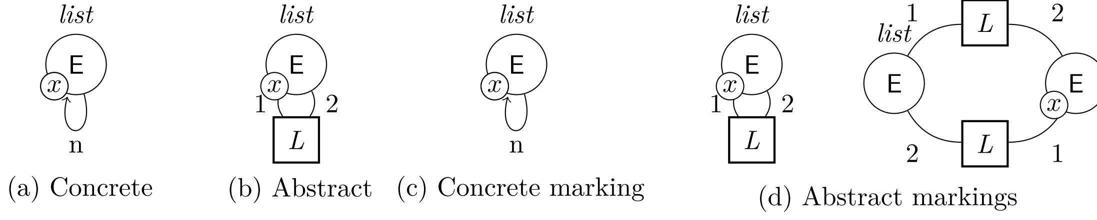
$$mark(L(H), x) \quad = \quad L(mark_\forall(H, x)).$$

That is, if we use in Definition 7.6 the function $mark_\forall$ instead of *mark*, to build the abstract state space, we get trace inclusion (see Theorem 7.3). Given an $\exists\, x$-free formula (in positive normal form) we can use the model checking algorithm presented in Section 7.2.1 for concrete state spaces. Unfortunately we can not use the algorithm for formulae containing $\exists\, x$-operators. As the abstract state space (potentially) contains more traces than the concrete one, a specific trace from the abstract state space not necessarily exists in the concrete one. We demonstrate such a case in the following example.

Example 7.8:

Figure 7.15(a) depicts a concrete configuration s and Figure 7.15(b) its abstraction s'. The single marking for s is depicted in Figure 7.15(c), while Figure 7.15(d) depicts the two markings for s'. Consider the qLTL-formula $\psi := \exists\, x.\, [\neg(var(head) = x)]$. Any state space with s as start state will falsify ψ, as there is no marking for x such that $s \models \neg(var(head) = x)$. But any state space starting with s' will verify ψ as there is a marking such that $s' \models \neg(var(head) = x)$.

The inclusion result (Theorem 7.3) that we got for *LTL*-formulae does not hold for general *qLTL*-formulae. The problem here is that considering a single abstract marking (potentially) reduces the amount of the correspondingly considered unmarked states. A



(a) Concrete (b) Abstract (c) Concrete marking (d) Abstract markings

Figure 7.15: Empty cyclic list and its markings.

good solution to this problem would be to not just consider single successors, but all subset of abstract markings that represents the complete set of unmarked configurations. However, to do so we need to decide language inclusion, which is not decidable for HRGs (Theorem 2.1 on page 22). Instead we decided to use a weaker solution here, where we restrict the choice for existential steps to markings of objects concretely represented within a state. That is the markings for existential quantification are determined by

$$mark_{\exists}(H, x) \quad = \quad \{setMark(H, x, v) \mid v \in V_H\}.$$

This marking function is less exact (results in additional false negatives) but sound. We define an extension to the marked state space considering both *mark*-functions.

Definition 7.8 (Extended marked state space):
Given an (abstract) start configuration $s \in HRG_\Sigma$, a transition function $\blacktriangleright$ and a set of markers M, the extended marked state space *is a tuple $(S, \blacktriangleright, \mu_\exists, \mu_\forall)$, where*

S	$\{s' \in mark_\forall(s) \mid s \in S_{\mathcal{T}_{\blacktriangleright,s}}\}$
$\mu_\forall \subseteq S \times M \times S$	*with* $(s, m, s') \in \mu_\forall$ *iff* $s' \in mark_\forall(s, m)$
$\mu_\exists \subseteq S \times M \times \mathcal{P}(S)$	*with* $(s, m, S') \in \mu_\exists$ *iff* $S' \in mark_\exists(s, m)$

Given an extended marked state space and an *qLTL*-formula we build a proof structure using the tableaux rules from Section 7.2.1 where we replace $R^{\forall x}$ and $R^{\exists x}$ by

$$(R^{\forall x})\ \frac{s \vdash \Phi \cup \{(s', \forall x.\psi)\}}{s \vdash \Phi \cup \{(s^1, \psi)\} \qquad \dots \qquad s \vdash \Phi \cup \{(s^n, \psi)\}}\ ,$$

where $\{s^1, \dots s^n\} = \{s^\# \mid (s', x, s^\#) \in \mu_\forall\}$, and

$$(R^{\exists x})\ \frac{s \vdash \Phi \cup \{(s', \exists x.\psi)\}}{s \vdash \Phi \cup \{(s^1, \psi),\ \dots\ (s^n, \psi)\}}\ ,$$

where $\{s^1, \dots s^n\} = \{s^\# \mid (s', x, s^*) \in \mu_\exists\}$.

Note that in case that s is a concrete state and $\blacktriangleright$ is a concrete transition function, it holds that $\mu_\exists = \mu_\forall$, i.e. the model checking for the extended marked state space behaves equal to the one for the marked state space.

Theorem 7.4 (Correctness of tableaux construction for abstract qLTL):
Given $\psi \in qLTL$, $s \in HC_\Sigma$ and transition functions $\blacktriangleright$ and $\triangleright$ with $\blacktriangleright \succeq \triangleright$. Let (V, E) be the proof structure over the extended marked state space (for s and $\blacktriangleright$) and the start assertion $s \vdash \{\psi\}$ then

$$(V, E) \text{ is successful} \quad \text{implies} \quad \mathcal{T}_{s,\triangleright} \models \psi$$

7.4 On The Fly Model Checking $qCTL^\star$

The temporal logic $CTL^\star$ is a combination of computational tree logic (CTL)[BK08] and LTL. $CTL^\star$ combines the path formulae from LTL with state formulae from CTL [BK08]. Analogous we can extend $qLTL$ to $qCTL^\star$ defined as follows.

Definition 7.9 ($qCTL^\star$):
Given a set of atomic proposition $A \subset MSOJ_\Sigma$, $qCTL^\star_A$ or state formulae are formed regarding to the following grammar:

$$\Phi \;:=\; true \;\big|\; [\varphi] \;\big|\; \Phi_1 \wedge \Phi_2 \;\big|\; \neg\Phi \;\big|\; \mathsf{E}\psi$$

where $\varphi \in A$ and ψ is a path formula *constructed by the following grammar*

$$\psi \;:=\; \Phi \;\big|\; \psi_1 \wedge \psi_2 \;\big|\; \neg\psi \;\big|\; \bigcirc\psi \;\big|\; \psi_1 \,\mathsf{U}\, \psi_2 \;\big|\; \forall x.\psi \;\big|\; \exists x.\psi$$

We define the semantics of a $qCTL^\star$ formula as a model relation. Given a $CTL^\star$ formula Φ, a transition system $\mathcal{T} \in TS_\Sigma$ (Definition 3.12) and a state $s \in S_\mathcal{T}$, the model relation $\models$ for state formulae is defined by:

$$\begin{array}{lll} s \models \neg\Phi & \textit{iff} & \textit{not } s \models \Phi \\ s \models \Phi_1 \wedge \Phi_2 & \textit{iff} & (s \models \Phi_1) \textit{ and } (s \models \Phi_2) \\ s \models \mathsf{E}\,\Phi & \textit{iff} & \pi \models \Phi \textit{ for some } \pi \in Path_s \end{array}$$

Grumberg *et. al.* extended their model checking algorithm for LTL towards $CTL^\star$ [BCG95]. Their extensions to the algorithm can be used to extend also $qLTL$ to $qCTL^\star$.

Adaptions are not necessary as the quantifications occur only within path formulae. Note that for over-approximated state spaces we need to restrict formulae to $qACTL^\star$, the analogue to $ACTL^\star$, as for full $CTL^\star$ the inclusion property (Theorem 7.3 on page 198) does not hold [BK08].

Further Grumberg *et. al.* extend the algorithm to work on the fly, i.e. the transition system that should be verified can be constructed during the run of the algorithm. That brings advantages as not all states of a transition system needs to be determined. Reconsider Example 7.2 on page 188, where we verified the given *LTL*-formula based on the first three states of the transition system (containing ten states), without the need of determining the remaining states. The impact of the on-the-fly ability becomes even more important when abstraction and markings are involved, as the number of marked abstract states is exponential in the number of markers.

7.5 Model Checking Lindstrom's Algorithm

In the previous chapters we verified some of the properties postulated in Section 1.1.2. In Chapter 5, we generated the abstract state space and verified the absence of *null*-pointer dereferences. In Chapter 6, we defined trees via MSO-formulae and verified that the terminal states have a tree shape. But there are still properties that we have to consider: *Completeness*, i.e. each node of the tree is been visited at least once, *termination* and *structure preservation*. While we already proved the treeness for terminal states we want to check in addition that these trees in the terminal states are exactly the same as in the start states. Each property can be expressed by a qLTL formula:

We say a node is visited whenever the variable *cur* referenced it. Thus *completeness* refers to the fact that for any node in the tree *cur* points at least once to that node:

$$\psi_{completness} := \forall x.\ \Diamond\ [var(cur) = x]$$

The algorithm *terminates* when *cur* points to the same object as *sen*. We therefore check that this happens after finite time by checking that *cur* neither points to none of the other objects nor to *null* infinitely often.

$$\psi_{termination} := \forall x.\ \Diamond\Box\ \neg[var(cur) = x] \wedge \Diamond\Box\ \neg[x = null]$$

Note that this formula approves termination only if no objects are generated during runtime. For Lindstrom's algorithm this approach is correct as the *sen*-object is the only object generated during execution and as the algorithm terminates as soon as *cur* points to it.

For *shape retainment* we can check in a first step, that the terminal states have a tree like structure.

$$\psi_{treeness} := \Diamond\Box\ ([\varphi_{sharing} \wedge \varphi_{cycle}])$$

To verify that the trees represented in the final states are the same as the ones in the start configuration we use an inductive argument. First we verify that the root node is the same at the beginning as at the end and secondly we verify for each vertex that the successors after termination are the same as in the beginning of the traversal.

$$\begin{array}{rcll}\psi_{treeness} & := & \forall x, l, r.\ (& [var(root) = x] \rightarrow \Diamond\Box\ [var(root) = x]) \\ & & \quad\wedge & ((x.l = l \wedge x.r = r) \rightarrow \Diamond\Box\ ((x.l = l \wedge x.r = r))\ )\end{array}$$

Unfortunately we cannot present any experimental result here as we have not implemented the model checking algorithm so far. However, in [HNR10; HBJ12] we used start configurations with manually added markings to verify the *qLTL*-formulae given above. The generation of the marked state space took less then ten seconds [HBJ12] and model checking was left to an external LTL model checker.

7.6 Conclusions

In the analysis of programs various properties of interest are temporal, i.e. depend on the development of the heap structure during execution. We can use LTL and standard model checking techniques to check temporal properties of the heap. However, if we want to analyse the dynamic behaviour of individual objects we need to track their identities between states. For this purpose we introduce markings and marked state spaces (Definition 7.6 on page 195). In order to describe the behaviour of objects over time we introduced *qLTL* (Definition 7.4 on page 194), a logic that extends *LTL* by quantifications over heap objects. With *qLTL* we are able to formalise the properties of termination, completeness and shape inheritance for Lindstrom's algorithm (Section 7.5). To model check *qLTL* we introduced an algorithm based on work by Grumberg *et. al.* (Section 7.1.1). First for concrete state spaces in Section 7.2 and then for abstract ones in Section 7.3. To realise markings for the latter we defined marking sets as an extension to HRGs that allows us to mark also abstracted objects.

The presented algorithm for qLTL model checking can easily be extended to $qCTL^\star$ and to work on-the-fly along the lines of [BCG95].

Chapter 8

8 Conclusion

We aim at a framework for the verification of object-oriented programs that is intuitive, efficient, fully automatised and powerful enough to verify relevant properties of modern, object-oriented programs. Even if we did not accomplished this ambitious goal in its entirety we attained some major achievements.

We develop a heap abstraction technique that is based on *hyperedge replacement grammars*. The hyperedge replacements as involved in HRG derivation steps are used for concretisation while the backward application yields abstraction. Combining concretisation and abstraction with a concrete semantics for object oriented language provides us with an abstract semantics that we use to generate abstract state spaces. We extended the base framework to be able to analyse Java bytecode programs. To specify properties of heap states we considered MSO over graphs and provide an algorithm to check them for abstract states. For temporal properties we introduce *qCTL* an extension of *CTL* that allows to describe the dynamic behaviour of objects during the execution. The combination of MSO over graphs and *qCTL* provides us with an expressiveness logic able to describe complex properties of object-oriented programs.

The Lindstrom's algorithm (on trees) is a convenient use case as it impressively illustrates how difficult it is to manually comprehend and verify significant properties of algorithms. The automatic verification with quite low user intervention (manual provided grammar for trees) demonstrates the abilities of our approach while the runtimes compare quite favorably to those in the literature. Further experimental results can be found in [Hei+14]. Important characteristics of our approach are the high degree of automation and the natural modelling of abstract heaps using graph grammars. Differently than competitive approaches, our framework is not restricted to handle lists and trees but supports user-defined data structures, which can be described in an intuitive and natural manner reflecting their recursive definitions.

The restriction to data structures of *bounded tree-width* as imposed by the use of hyperedge replacement grammars is a major disadvantage of our approach. E.g. neither the set of all directed acyclic graphs nor the set of all arbitrary graphs can be described and

thus we are not able to analyse programs that use such data structures. For example, Lindstrom's algorithm works on arbitrary directed acyclic graphs as input, but as we are not able to represent a corresponding abstract start configuration we cannot construct a state space. However, most commonly used data structures have a bounded tree-width and thus can be treated. Among those are the most common ones such as lists and trees as well as varieties and combinations of those. In this thesis we considered e.g. trees, lists, doubly-linked lists, rooted list and trees with linked leaves and parent pointers as well as grid structures of fixed height.

Another limitation imposed by the use of hyperedge replacement grammars is the missing support for primitive data types apart from boolean variables. Many algorithms of interest depend on their use such as algorithms on binary search trees, balanced data structures or sorting algorithms.

8.1 Achievements

This thesis presents our entire approach. We introduce the preliminary foundations, develop a framework to determine the (abstract) state space of Java bytecode programs, present a model checking algorithm for *quantified CTL* an extension of CTL that allows us to express properties of the dynamic behaviour of objects, as well as an algorithm to determine if an abstract shape fulfils some MSO-defined properties. These together amount to a complete framework for the analysis of object-oriented programs. The main results of this thesis are shortly summarised in the following.

User Definable Data Structures

The support for user defined data structures is one of the main advantages of our approach. Desired data structures can be defined via hyperedge replacement grammar. As the abstraction is entirely based on HRGs no additional steps such as providing abstract semantics are needed. However, we recognise that to be suitable for our approach the grammar needs to fulfil some properties. It is not always evident whether a grammar fulfils these properties and it is tedious to establish them. We presented techniques to verify that a given grammar provides the necessary properties, as well as algorithms to obtain those for a given HRG in an automated manner without the need of user interaction. Within these algorithms the construction of the *Local Greibach Normal Form* takes a prominent position, as it is needed for *local concretisations*, which in turn is essential to realise abstract executions within our framework.

MSO Defined Graph Properties

In order to describe structural properties of heaps we introduce the logics FO and MSO over graphs. We present an alternative proof of the famous theorem by Courcelle [Cou90] which states that it is decidable if the graphs of an HRG-defined set fulfil a property defined by an MSO-formula. Our proof involves the construction of the intersection of an HRG and an MSO-formula, which leads to a practical relevant algorithm that allows us to check reasonable formulae and graph grammars.

We demonstrate the practical usability by checking the abstract states of Lindstrom's algorithm against the MSO-formula for tree structures. Also the comparison with some experiment results obtained by a MONA-based implementation of the classical automata approach throws a positive light on our approach. Our results for MSO could be adapted to other areas where big graphs and systems are (abstractly) described by hyperedge replacement grammars or similar approaches and where a needed for checking those against formal specifications given as MSO-formulae exists.

qCTL Model Checking

We extend the classical LTL logic by quantifications over heap objects. This allows to describe the dynamic behaviour of objects. A marking is used in order to track the identity of objects between states. We call the logic *qLTL* and we present a model checking algorithm for concrete state spaces that is based on the on-the-fly model checking approach by Grumberg *et. al.* . Markings for abstract state spaces need to include also the abstracted objects. Therefore we define an extension of HRGs called marking sets, that allows us to concretise arbitrary objects that then can be marked. Using marking sets we extend the model checking algorithm for *qLTL* towards abstract state spaces. The approach easily extends to $qCTL^\star$. Adding quantifiers to classical $CTL^\star$ model checking together with MSO defined state properties allows us to specify and verify complex temporal properties that involve tracking of objects over time. Among others this includes evolution of shape properties as well as reachability and connectivity properties.

Verification Tool for Java Bytecode

Most of the concepts and techniques presented in this thesis are implemented into the prototype tool Juggrnaut. The tool takes a Java program compiled to Java bytecode, a data structure grammar and an abstract start configuration as input and generates the corresponding abstract state space. In addition the tool is able to check state properties provided as MSO-formula for each of the generated states. We discuss necessary adaptions to our framework imposed by Java's object-oriented approach and language specific concepts.

The tool is implemented in Java and involves more than 20.000 lines of code. All experimental results presented in this thesis are obtained using this implementation on a MacBook Pro with 2 GHz and 8 GB of memory. The runtimes for Lindstrom's algorithm as well as for the various MSO experiments are encouraging and compare well with those in the literature.

8.2 Future Work

The performance of our abstraction technique highly depends on the provided DSG. For common data structures, a library of grammars can be readily made available. Whether or not a grammar is suitable, however, not only depends on the used data structures but also on the analysed program and how it alters the heap. The Lindstrom algorithm for example traverses a tree and therefore we needed to add path rules to the grammar in order to abstract intermediate states of the algorithm. In Example 5.3 on page 101, the X-nonterminal abstracts also parts of the method stack to handle unbounded recursion. Thus, depending on the algorithm it might be necessary to encode additional knowledge into the abstraction grammar. To maximise the automation of our approach, we plan to infer suitable grammars at runtime. We were able to achieve some promising results using experimental implementations of learning algorithms.

As the major disadvantages of our approach are imposed by the use of hyperedge replacement grammars, the use of other graph rewriting techniques such as the DPO-approach [Roz97] could be considered. The challenge in extending the framework to other graph grammars is to adapt the four necessary properties specified in Chapter 3. In order to consider also primitive data values the use of *parametrised grammars* should be considered, where vertices and edges are annotated by variables and conditions representing the primitive data values. Parametrised grammars could also be used to express size conditions of data structures such as the balance of trees.

In Section 7.5 we use a *qLTL* formula to describe termination. While the presented approach works fine for the termination analysis of Lindstrom's algorithm it might not work in other cases, especially when objects are allocated during runtime. Performing automated termination analysis on the basis of the generated abstract state space is an interesting approach. Note that a loop-free state space ensures termination of the underlying program. If there are loops in the state space we need to check if they can be iterated infinitely often. There are two techniques that could be considered to prove that the number of iterations is finite. The first based on markings, where we mark specific nodes within one state of the cycle and verify that this specific marking is never revisited. From this we can conclude that after visiting a finite set of possible markings the loop has to be left eventually. The second is based on *size change* [LJB01]. For some state in the loop we consider the number of objects abstracted within the nonterminal edges. If during one iteration the number decreases for one of the edges, we can conclude that the objects represented by this edge are consumed after finitely many iterations and thus the

loop has to be left. If all loops of a state space can be iterated only finitely often the underlying program terminates. Besides termination analysis, the results for single loops can also be used to infer LTL-fairness constraints.

So far we have not implemented the model checking algorithm presented in Chapter 7. It is the part that is missing to complete the implementation of the framework. In model checking of abstract state spaces the marking sets play an important role. It is still an open question whether for each HRG there exists a marking set and if it can be constructed. Thus it is still necessary to provide marking sets manually. There is a good chance that the construction could be inferred from the one for LGNF. Marking sets could be of more general interest in graph research, as they provide the possibility to consider arbitrary vertices of the derivable graphs.

The implementation of our MSO approach used for the experimental results in Section 6.5.5 and Section 6.6 includes some optimisations that are not presented within this thesis nor published elsewhere. The formal definition of these optimisations will be content of future research as well as the development of further optimisations.

Index

Symbols

E_H^N 18
E_H^T 18
G^X 20
$G^{\overline{X}}$ 20
$L_G(H)$ 20
N_Σ 18
$N_t^\forall$ 135
T_Σ 18
$Emb(I, H)$ 24
$\Diamond$ 183
$\Box$ 183
HAG_Σ 43
$\Sigma_\mathbb{F}$ 79
MSOJ_Σ 173
$\mathbb{N}_t^\exists$ 135
$Paths_s$ 64
R 183
$Tent_\Sigma$ 17
U 183
$\bigcirc$ 183
$\Leftarrow_G$ 26
$\Leftarrow_G^*$ 26
$\succeq$ 64
Σ 18
ClassFile$_O$ 96
ClassFile$_S$ 96
▲-tentacle 75
fields 95
implements 95
isInterface 95
methods 95
modifiers 95
name 95
super 95
$\Rightarrow^*$ 20
eval 138
$evaltree(H, \alpha, \varphi)$ 136
maxAbstr 89
$\simeq$ 21
$intValue_H$ 106
$\rightsquigarrow$ 121
$\mapsto$ 15
cEnv 95
$suc_H(v, a)$ 40
$\bowtie$ 147
null$_H$ 101
$\mathrm{out}(v)$ 38
$\mathrm{outLabel}_G(X, i)$ 41, 44
$\mathcal{P}(A)$ 15
$\preceq$ 75
$\approx$ 23
$mark_M(H)$ 191
$f{\restriction}S$ 15
$\rightarrow_p$ 53
$setMark(H, x, v)$ 192
static$_H$ 101
$succ(t)$ 138
method$_H$ 101
∇-tentacle 75
$\mathcal{T}$ 63
$\uplus$ 15
$\rhd$ 63
$f : A \rightarrow B$ 15
Class/Field 95
MSig 95
incPc 106
null 97
new 104
peek 106
pop 105
push 105

$qCTL^{\star}$ 202
$qLTL$ 192
setSuc 104
suc 104

A

abstract heap graph 40
abstract instructions 94, 103
abstract vertex 41
abstraction 88, 119
abstraction function 118
active method 101
approximation 65
associativity 27

B

backward application 24, 26
backward confluence 29
backward replacement 89
basic modifier 104
boolean 99
bounded grammar 51

C

class definition 74
compositional properties 31
compositional property 32
concrete heap graph 39
concrete semantics 107
concrete vertex 40
Concretisation 118
concretisation 88
concretisation function 60
concretisation set 88
concretiser 68
confluence 26
connected tentacles 18
critical pair 30

D

decomposition 27
derivation 20

E

embedding 89
evaluation tree 135, 140, 143
  anonymisation 157
  as compositional property 160
  composition 146
  evaluation 151
  minimisation 152
  pre-evaluation 154
  three-valued evaluation 155
exception handling 102
extended evaluation tree 143

F

first order logic 128
FO 128
FO properties 131

G

garbage collector 121
grammar equivalence 21
Greibach Normal Form 51

H

handle 18
heap abstraction grammar 43
heap alphabet 75
heap configuration 78, 101
heap graph
  abstract heap graph 40
hyperedge 16
hyperedge replacement 19, 79
hyperedge replacement grammar 20
hypergraph 16
  definition 17
hypergraph embedding 24
hypergraph replacement 25

I

increasing 60
increasing grammar 28
initial path 64

J

Java class file 95
JVM 93
JVM state 97

JVM$_{\mathcal{O}}$ 94

K

k-safe 68

L

language 82
language decomposition 27
level of a node 137
Lindstrom's Algorithm 2
literals 99
local concretisation 49
Local Greibach Normal Form 51, 52
logic equivalence 129
LTL 182
  model checking 186

M

matrix 130
method stack 100
monadic second order logic 162
MSO 162
MSO for JVM 173

N

non-increasing grammar 28
nonterminal edges 18
null node 97

O

object fields 95
opcode 94
  Checkcast 115
  Cond 110
  Dupx 108
  GetField 114
  GetStatic 111
  Goto 110
  InstanceOf 114
  InvokeSpecial 115
  InvokeStatic 111
  InvokeVirtual 116
  Load 109
  New 113
  Pop 109
  Prim(iand) 108
  Prim(iconst_0) 108
  Prim(iconst_1) 108
  Prim(if_acmpeq) 107
  Prim(if_acmpneq) 107
  Prim(if_icmpeq) 107
  Prim(if_icmpneq) 107
  Prim(ior) 108
  Prim(PrimOp) 107
  PutField 114
  PutStatic 111
  Return 112
  Store 109
outgoing edge 38
over approximation 64

P

path 64
Positive normal form 183
prefix 130
prenex normal form 130
productive grammar 28
program counter 100

Q

quasi-equivalence 23

R

rank 16
ranked alphabet 18
reduction tentacle 41
restricted abstraction 117
rule graph 20

S

safe approximation 65
safe transition relations 66
sequence 15
splitting grammar 32
state space 6, 63
  abstraction 119
  extended marking 201
  marking 195
  over approximation 64
static environment 95

static fields 95
static variables 99
string
    grammar 52
    representation 52
subtype relation 75
successor 40

T

tentacle 17
tentacle concretisation 61
tentacle label set 41
terminal edges 18
terminal state 102
termination 102
transition function 64
transition relation 64
    concretisation 67
transition system 63
typing 44

V

Val 96
vertex concretisation 61
vertex node 136

Bibliography

[AB02] A. Asteroth and C. Baier. *Theoretische Informatik: eine Einführung in Berechenbarkeit, Komplexität und formale Sprachen mit 101 Beispielen.* Pearson Studium, 2002.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking.* The MIT Press, 2008.

[Bau+08] Jörg Bauer, Iovka Boneva, Marcos E. Kurbán, and Arend Rensink. "A Modal-Logic Based Graph Abstraction". In *ICGT*, Volume 5214 of *Lecture Notes in Computer Science*, pages 321–335. Springer, 2008.

[BM77] John L. Bell and Moshé Machover. *A Course in Mathematical Logic.* North-Holland, 1977, pages I–XVIII, 1–599.

[BCO06a] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. "A Decidable Fragment of Separation Logic". In *FMCO*, Volume 4111 of *Lecture Notes in Computer Science*, pages 97–109. Springer, 2006.

[BCO06b] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. "Smallfoot: Modular Automatic Assertion Checking with Separation Logic". In *FMCO*, Volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2006.

[BCI11] Josh Berdine, Byron Cook, and Samin Ishtiaq. "SLAyer: Memory Safety for Systems-Level Code". In *CAV*, Volume 6806 of *Lecture Notes in Computer Science*, pages 178–183. Springer, 2011.

[BCG95] Girish Bhat, Rance Cleaveland, and Orna Grumberg. "Efficient On-the-Fly Model Checking for CTL". In, pages 388–397. IEEE Computer Society Press, 1995.

[Bog+07] Igor Bogudlov, Tal Lev-Ami, Thomas W. Reps, and Mooly Sagiv. "Revamping TVLA: Making Parametric Shape Analysis Competitive". In *CAV*, Volume 4590 of *Lecture Notes in Computer Science*, pages 221–225. Springer, 2007.

[Bou+06] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomás Vojnar. "Abstract Regular Tree Model Checking". In volume 149 of *Electr. Notes Theor. Comput. Sci.* Pages 37–48, 2006.

[Bro+13] James Brotherston, Carsten Fuhs, Nikos Gorogiannis, and Juan Navarro Pérez. *A Decision Procedure for Satisfiability in Separation Logic with Inductive Predicates.* Tech. rep. RN/13/15. University College London, 2013.

[Cou90] Bruno Courcelle. “The Monadic Second-Order Logic of Graphs. I. Recognizable Sets of Finite Graphs”. In volume 85 of *Inf. Comput.* Pages 12–75, 1990.

[CD12] Bruno Courcelle and Irène Durand. “Automata for the verification of monadic second-order graph properties”. In volume 10 of *J. Applied Logic*, pages 368–409, 2012.

[CE12] Bruno Courcelle and Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic - A Language-Theoretic Approach.* Vol. 138. Encyclopedia of mathematics and its applications. Cambridge University Press, 2012, pages I–XIV, 1–728.

[CC77] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, *POPL '77*, pages 238–252. ACM, Los Angeles, California, 1977.

[Dam92] Mads Dam. “CTL and ECTL* as Fragments of the Modal /*mu*-Calculus”. In *CAAP*. Ed. by Jean-Claude Raoult. Vol. 581. Lecture Notes in Computer Science. Springer, 1992, pages 145–164.

[DKR04] Dino Distefano, Joost-Pieter Katoen, and Arend Rensink. “Who is Pointing When to Whom?” In *FSTTCS*, Volume 3328 of *Lecture Notes in Computer Science*, pages 250–262. Springer, 2004.

[DKR06] Dino Distefano, Joost-Pieter Katoen, and Arend Rensink. “Safety and Liveness in Concurrent Pointer Programs”. In *FMCO*, Volume 4111 of *Lecture Notes in Computer Science*, pages 280–312. Springer, 2006.

[DP08a] Dino Distefano and Matthew J. Parkinson. “jStar: Towards Practical Verification for Java”. In *OOPSLA*, pages 213–226. ACM, 2008.

[DRK02] Dino Distefano, Arend Rensink, and Joost-Pieter Katoen. “Model Checking Birth and Death”. In *IFIP TCS*, Volume 223 of *IFIP Conference Proceedings*, pages 435–447. Kluwer, 2002.

[DP08b] Mike Dodds and Detlef Plump. “From Hyperedge Replacement to Separation Logic and Back”. In volume 16 of *ECEASST*, 2008.

[DPV11] Kamil Dudka, Petr Peringer, and Tomás Vojnar. “Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic”. In *CAV*, Volume 6806 of *Lecture Notes in Computer Science*, pages 372–378. Springer, 2011.

[EF95] Heinz-Dieter Ebbinghaus and Jörg Flum. *Finite Model Theory.* Perspectives in Mathematical Logic. Springer, 1995.

[Eng92] Joost Engelfriet. “A Greibach Normal Form for Context-free Graph Grammars”. In *Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, Volume 623 of *Lecture Notes in Computer Science*, pages 138–149. Springer, 1992.

[Hab92] Annegret Habel. *Hyperedge Replacement: Grammars and Languages*. Vol. 643. Lecture Notes in Computer Science. Springer, 1992.

[HBJ12] Jonathan Heinen, Henrik Barthels, and Christina Jansen. “Juggrnaut - An Abstract JVM”. In *FoVeOOS*, Volume 7421 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2012.

[Hei+14] Jonathan Heinen, Christina Jansen, Joost-Pieter Katoen, and Thomas Noll. “Verifying Pointer Programs using Graph Grammars”. In *Science of Computer Programming*, pages, 2014.

[HNR10] Jonathan Heinen, Thomas Noll, and Stefan Rieger. “Juggrnaut: Graph Grammar Abstraction for Unbounded Heap Structures”. In volume 266 of *Electr. Notes Theor. Comput. Sci.* Pages 93–107, 2010.

[Hen+95] Jesper G. Henriksen, Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. “Mona: Monadic Second-Order Logic in Practice”. In *TACAS*, Volume 1019 of *Lecture Notes in Computer Science*, pages 89–110. Springer, 1995.

[IRS13] Radu Iosif, Adam Rogalewicz, and Jirí Simácek. “The Tree Width of Separation Logic with Recursive Definitions”. In *CADE*, Volume 7898 of *Lecture Notes in Computer Science*, pages 21–38. Springer, 2013.

[JGN14] Christina Jansen, Florian Göbe, and Thomas Noll. “Generating Inductive Predicates for Symbolic Execution of Pointer-Manipulating Programs”. English. In *Graph Transformation*. Ed. by Holger Giese and Barbara König. Vol. 8571. Lecture Notes in Computer Science. Springer, 2014, pages 65–80.

[Jan+11] Christina Jansen, Jonathan Heinen, Joost-Pieter Katoen, and Thomas Noll. “A Local Greibach Normal Form for Hyperedge Replacement Grammars”. In *LATA*, Volume 6638 of *Lecture Notes in Computer Science*, pages 323–335. Springer, 2011.

[KR06] Harmen Kastenberg and Arend Rensink. “Model Checking Dynamic States in GROOVE”. In *SPIN*, Volume 3925 of *Lecture Notes in Computer Science*, pages 299–305. Springer, 2006.

[Kep05] Stephan Kepser. “HLT/EMNLP 2005, Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference, 6-8 October 2005, Vancouver, British Columbia, Canada”. In *HLT/EMNLP*, The Association for Computational Linguistics, 2005.

[KL09] Joachim Kneis and Alexander Langer. "A Practical Approach to Courcelle's Theorem". In volume 251 of *Electr. Notes Theor. Comput. Sci.* Pages 65–81, 2009.

[KLR11] Joachim Kneis, Alexander Langer, and Peter Rossmanith. "Courcelle's Theorem - A Game-Theoretic Approach". In volume abs/1104.3905 of *CoRR*, 2011.

[Knu08] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 0: Introduction to Combinatorial Algorithms and Boolean Functions (Art of Computer Programming)*. 1st. Addison-Wesley, 2008.

[Lan+12] Alexander Langer, Felix Reidl, Peter Rossmanith, and Somnath Sikdar. "Evaluation of an MSO-Solver". In *ALENEX*, pages 55–63. SIAM / Omnipress, 2012.

[LJB01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. "The Size-Change Principle for Program Termination". In *POPL*, pages 81–92. ACM, 2001.

[LMS04] Tal Lev-Ami, Roman Manevich, and Shmuel Sagiv. "TVLA: A System For Generating Abstract Interpreters". In *IFIP Congress Topical Sessions*, pages 367–376. Kluwer, 2004.

[LS00] Tal Lev-Ami and Shmuel Sagiv. "TVLA: A System for Implementing Static Analyses". In *SAS*, Volume 1824 of *Lecture Notes in Computer Science*. Springer, 2000.

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. 2nd. Addison-Wesley, Boston, MA, USA, 1999.

[Lin73] Gary Lindstrom. "Scanning List Structures Without Stacks or Tag Bits". In volume 2 of *Inf. Process. Lett.* Pages 47–51, 1973.

[LRS06] Alexey Loginov, Thomas W. Reps, and Mooly Sagiv. "Automated Verification of the Deutsch-Schorr-Waite Tree-Traversal Algorithm". In *SAS*, Volume 4134 of *Lecture Notes in Computer Science*, pages 261–279. Springer, 2006.

[Mar06] Hendrik Maryns. "On the Implementation of Tree Automata: Limitations of the Naive Approach". In *TLT*, pages 235–246, 2006.

[NR08] Thomas Noll and Stefan Rieger. "Verifying Dynamic Pointer-Manipulating Threads". In *FM 2008: Formal Methods*. Ed. by Jorge Cuellar, Tom Maibaum, and Kaisa Sere. Vol. 5014. Lecture Notes in Computer Science. Springer, 2008, pages 84–99.

[Plu05] Detlef Plump. "Confluence of Graph Transformation Revisited". In *Processes, Terms and Cycles*, Volume 3838 of *Lecture Notes in Computer Science*, pages 280–308. Springer, 2005.

[Plu10] Detlef Plump. "Checking Graph-Transformation Systems for Confluence". In volume 26 of *ECEASST*, 2010.

[Pnu77] Amir Pnueli. "The Temporal Logic of Programs". In *FOCS*, pages 46–57. IEEE Computer Society, 1977.

[Ren03] Arend Rensink. "Towards Model Checking Graph Grammars". In *Workshop on Automated Verification of Critical Systems (AVoCS)*, *Technical Report*, pages 150–160. University of Southampton, 2003.

[Ren04] Arend Rensink. "The GROOVE Simulator: A Tool for State Space Generation". In *AGTIVE*, Volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer, 2004.

[Ren06] Arend Rensink. "Model Checking Quantified Computation Tree Logic". In *CONCUR*, Volume 4137 of *Lecture Notes in Computer Science*, pages 110–125. Springer, 2006.

[RZ09] Arend Rensink and Eduardo Zambon. "A Type Graph Model for Java Programs". In *FMOODS/FORTE*, Volume 5522 of *Lecture Notes in Computer Science*, pages 237–242. Springer, 2009.

[RZ10] Arend Rensink and Eduardo Zambon. "Neighbourhood Abstraction in GROOVE". In volume 32 of *ECEASST*, 2010.

[RZ12] Arend Rensink and Eduardo Zambon. "Pattern-Based Graph Abstraction". In *ICGT*, Volume 7562 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2012.

[Rey02] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[Roz97] Grzegorz Rozenberg, ed. *Handbook of Graph Grammars and Computing by Graph Transformation: volume I. Foundations*. World Scientific Publishing, 1997, p. 553.

[SRW96] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. "Solving Shape-Analysis Problems in Languages with Destructive Updating". In *POPL*, pages 16–31. ACM Press, 1996.

[SRW02] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. "Parametric Shape Analysis via 3-Valued Logic". In volume 24 of *ACM Trans. Program. Lang. Syst.* Pages 217–298, 2002.

[SW67] H. Schorr and W. M. Waite. "An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures". In volume 10 of *Commun. ACM*, pages 501–506. ACM, 1967.

[ST93] Paul D. Seymour and Robin Thomas. "Graph Searching and a Min-Max Theorem for Tree-Width". In volume 58 of *J. Comb. Theory, Ser. B*, pages 22–33, 1993.

[Sko97] Konstantin Skodinis. "The Bounded Tree-Width Problem of Context-Free Graph Languages". In *WG*, Volume 1335 of *Lecture Notes in Computer Science*, pages 303–317. Springer, 1997.

[Sog08] D. Soguet. “Génération automatique d’algorithmes linéaires: Décomposition de graphes, logique, stratégies de capture”. PhD thesis. University of Paris XI, 2008.

[SSB01] Robert F. Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer, 2001.

[Ull76] Jeffrey R. Ullmann. “An Algorithm for Subgraph Isomorphism”. In volume 23 of *J. ACM*, pages 31–42. ACM, New York, NY, USA, 1976.

[Yan+08] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O’Hearn. “Scalable Shape Analysis for Systems Code”. In *CAV*, Volume 5123 of *Lecture Notes in Computer Science*, pages 385–398. Springer, 2008.

[ZR11] Eduardo Zambon and Arend Rensink. “Using Graph Transformations and Graph Abstractions for Software Verification”. In volume 38 of *ECEASST*, 2011.

[ZR12] Eduardo Zambon and Arend Rensink. “Graph Subsumption in Abstract State Space Exploration”. In *GRAPHITE*, Volume 99 of *EPTCS*, pages 35–49, 2012.