Analysis and Abstraction of Graph Transformation Systems via Type Graphs

Von der Fakultät für Ingenieurwissenschaften,
Abteilung Informatik und Angewandte Kognitionswissenschaft
der Universität Duisburg-Essen

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
(Dr. rer. nat.)

genehmigte Dissertation

von

Dennis Nolte
aus
Willich

1. Gutachter:	Prof. Dr. Barbara König
2. Gutachter:	Prof. Dr. Andrea Corradini
Tag der mündlichen Prüfung:	25. Juni 2019

DOI: 10.17185/duepublico/70359
URN: urn:nbn:de:hbz:464-20190821-100424-8

To my family and friends

This thesis is based on the following original publications:

✎ **Chapter 1. Introduction:**

[Nol17] D. Nolte. “Analysis and Abstraction of Graph Transformation Systems via Type Graphs”. In: *STAF 2017 Doctoral Symposium.* Vol. 1955. CEUR Workshop Proceedings. 2017.

✎ **Part I. Preliminaries and Foundations:**

[KN+18] B. König, D. Nolte, J. Padberg, and A. Rensink. “A Tutorial on Graph Transformation”. In: *Graph Transformation, Specifications, and Nets – In Memory of Hartmut Ehrig.* Ed. by R. Heckel and G. Taentzer. LNCS 10800. Springer, 2018, pp. 1–22. DOI: 10.1007/978-3-319-75396-6_5.

✎ **Part II. Termination Analysis of Graph Transformation Systems:**

[BK+15] H. J. S. Bruggink, B. König, D. Nolte, and H. Zantema. “Proving Termination of Graph Transformation Systems Using Weighted Type Graphs over Semirings”. In: *Proc. of ICGT ’15 (International Conference on Graph Transformation).* 2015, pp. 52–68. DOI: 10.1007/978-3-319-21145-9_4. arXiv: 1505.01695 [cs.LO].

[ZNK16] H. Zantema, D. Nolte, and B. König. “Termination of Term Graph Rewriting”. In: *Proc. of WST ’16 (Workshop on Termination).* 2016.

✎ **Part III. Specifying Graph Languages:**

[CKN17] A. Corradini, B. König, and D. Nolte. “Specifying Graph Languages with Type Graphs”. In: *Proc. of ICGT ’17 (International Conference on Graph Transformation).* LNCS 10373. Springer, 2017, pp. 73–89. DOI: 10.1007/978-3-319-61470-0_5. arXiv: 1704.05263 [cs.FL].

[CKN19] A. Corradini, B. König, and D. Nolte. “Specifying Graph Languages with Type Graphs”. In: *Journal of Logical and Algebraic Methods in Programming* Vol. 104 (2019), pp. 176–200. DOI: 10.1016/j.jlamp.2019.01.005.

✎ **Part IV. Abstract Object Rewriting:**

[CH+19] A. Corradini, T. Heindel, B. König, D. Nolte, and A. Rensink. “Rewriting Abstract Structures: Materialization Explained Categorically”. In: *Foundations of Software Science and Computation Structures.* Ed. by M. Bojańczyk and A. Simpson. Cham: Springer International Publishing, 2019, pp. 169–188. DOI: 10.1007/978-3-030-17127-8_10. arXiv: 1902.04809 [cs.LO].

A complete list of the author’s publications can be found on page 215.

Abstract

The aim of this thesis is to analyse *graph specification frameworks* based on *type graphs* and show their applicability for verification techniques based on formal language theory. In order to specify and analyse the behaviour of dynamically evolving systems it is important to use suitable specification languages which support practical verification methods. Many concurrent and distributed systems can be modelled by graphs and graph transformation rules. However, graph-like structures introduce an additional level of complexity, compared to rule-based systems where states have either a word or tree structure. In particular, many complex models induce infinite state spaces, such that explicit verification often fails, since this requires the construction of the entire state space. Therefore, it is necessary to also put a focus on abstraction mechanisms.

Graph abstractions are used in many frameworks for over-approximation of graph transformation systems. These frameworks use abstractions that implicitly specify graph languages and there exist various approaches which try to find an abstraction that is fine enough to enable successful verification, but coarse enough to efficiently implement abstract graph rewriting. This work introduces a general framework which can be suitably instantiated, in order to obtain methods usable in practice.

First, a basic graph specification framework based on type graphs is introduced. The framework is subsequently refined to an approach based on *weighted type graphs* which can be used for termination analysis of graph transformation systems.

Second, it is shown how three different refinements of the basic framework influence decidability, expressiveness and closure properties of type graph specification languages. Among these refinements, *multiply annotated type graphs* are discussed which build the foundation for graph specification frameworks expressive enough to specify strongest postconditions.

Finally, by exploiting universal properties from category theory, a general framework for *abstract object rewriting* is introduced in which strongest postconditions can be computed for annotated objects in an arbitrary topos. A concrete instance of the framework, namely the multiply annotated type graph, is implemented in a prototype tool to substantiate the practicability of the framework.

Preface

Over the last four years, I had the pleasure to conduct research as a PhD student in the theoretical computer science group of *Barbara König* at the University of Duisburg-Essen. Most of the results which were worked out, presented and published during this time are summarized in this thesis. Before I present these results, I would like to make some short remarks concerning the origins of the material as well as express my gratitude to all those wonderful people who accompanied me on this adventure called research.

Origins of the Material

The tutorial chapter for graph transformation (Chapter 3) as well as the three main parts (Parts II to IV) of this thesis are based on some of my joint publications with other authors. Except for our work presented in [BK+15] and [KN+18], I created initial drafts for all remaining publications, which later served as a basis for the respective final versions. During the writing process of each paper I contributed the majority of examples and structured the text to generate an easy to follow flow of thoughts. In this thesis, I streamlined the material with respect to a fixed notation. In every chapter, I added several examples, added more explanations and rewrote text passages whenever I had the feeling that they were written too densely in the respective publication due to the page restrictions. In every main part, I added an additional preliminary section such that this thesis becomes self-contained i.e. the consultation of other literature is not mandatory, to be able to understand the corresponding chapters.

I will now comment on the three main parts and my contributions in the creation process of the referenced publications in more detail.

Part II: Termination Analysis of Graph Transformation Systems

The termination analysis approach based on weighted type graphs, as presented in Chapter 5, is a collaboration with *Sander Bruggink*, *Barbara König* and *Hans Zantema*. It was the first topic I started contributing on and is a generalization of an approach introduced by my co-authors in [BKZ14]. The theoretical background was already worked out in detail, however, the implementation in the tool GREZ lacked the capability to produce results whenever non-linear arithmetic expressions were involved. I extended the SMT encodings used by GREZ to overcome this issue and generated examples and termination proofs for several graph transformation systems. Later on, in [ZNK16], we extended the weighted type graph approach to work for term rewriting systems as well. I contributed substantially in the development of the basic version interpretation for term graph productions and in the two transformation encodings, namely the number and function encoding. The results of this work are summarized in Chapter 6, though, compared to the original publication, I added a section to properly introduce term rewriting systems. Furthermore, I was responsible for all results of the experiments summarized in Appendix B.

Part III: Specifying Graph Languages

Part III of this thesis is based on a joint work with *Andrea Corradini* and *Barbara König*. During a research stay in Pisa, we worked out several frameworks based on type graphs. I investigated decidability and closure properties for these frameworks and wrote down our results, which were then accepted as a conference paper [CKN17]. Later, in a special issue journey version [CKN19], we integrated the proofs in the main text, added additional examples/explanations and I provided a non-trivial proof for the non-closure of annotated type graphs under the complement operation.

Part IV: Abstract Object Rewriting

The idea for a general abstract rewriting framework arose during discussions between *Barbara König* and *Arend Rensink* over a decade ago. At some point, during our own search for an abstraction mechanism suitable for the verification of systems, sketches from these discussions reappeared, including additional notes from *Tobias Heindel*. Together with *Andrea Corradini* we were able to work out a materialization category for objects in an arbitrary topos. Before we got the general framework, I already had worked out a concrete instance in form of the materialization construction for graphs. Later on, I wrote down the connection between the terminal object of our materialization category and the notion of partial map classifiers. As a follow up, we then generalised our work on annotated type graphs to the abstract rewriting of annotated objects. The results, which are explained in Part IV of this thesis, got accepted as a conference paper [CH+19]. The paper was nominated for the EATCS best paper award.

Acknowledgements

Research is a continuous process. It can be educational, exciting, frustrating, satisfying, tedious and illuminating, but in the end it is a path chosen by those who seek to learn and share knowledge beyond the already discovered. The sharing aspect of research teaches us an important lesson: The journey to wisdom is one that the researcher does not travel alone. Many astounding people accompanied and supported me during my adventure and I want to mention them here.

First and foremost, I would like to express my gratitude to *Barbara König* for her support and guidance as well as for offering me the opportunity to conduct research under her supervision. She introduced me to graph transformation and provided me the freedom to follow my intuition and develop my own ideas. I rarely got stuck in this process since she was always there to answer my questions. Our fruitful discussions played a major role in every breakthrough result achieved with respect to the theoretical problems I tackled. Thanks to her, I got to know many other researchers and I got the chance to work with them too.

Among these researchers, I am greatly indebted to *Andrea Corradini* for acting as the second assessor of this thesis. I will never forget the inspiring one month research stay in Pisa, where we worked on the foundations of this topic that subsequently led to multiple successful publications. I admire the patience he showed in answering my questions, while he simultaneously introduced me to various categorical notions.

Furthermore, I would like to thank all my co-authors with which I have collaborated in several publications, namely (in alphabetical order) *Sander Bruggink*, *Tobias Heindel*, *Maxime Nederkorn*, *Julia Padberg*, *Arend Rensink* and *Hans Zantema*. I enjoyed working with every one of them.

I consider myself lucky that I was given the chance to be member of a theoretical computer science group with an awesome working atmosphere and lots of illuminating discussions during lunch. Therefore, I would like to acknowledge my former and present colleagues: *Harsh Beohar*, *Benjamin Cabrera*, *Richard Eggert*, *Matthias Hülsbusch*, *Henning Kerstan*, *Christina Mika-Michalski*, *Sebastian Küpper*, *Lars Stoltenow* and *Jan Stückrath*.

Special thanks go to my friend and former colleague *Christoph Blume* for sharing his experience in the graduation process with me. Including him, all of my colleagues were so kind to provide valuable comments on early drafts of this thesis.

Additionally, I would like to thank my elite tutor team: *Rebecca Bernemann*, *Philip Garus*, *Marleen Matjeka* and *Matthias Schaffeld*. They reliably supported me in my teaching assistance duties for several years, such that I had enough time to efficiently conduct my research.

Finally, I want to express my gratitude to *Tamara*, our *families* and *friends* for their encouragement and assistance during the whole time. I will be forever grateful to you. Without your support I would not be who I am today.

Thank you all.

Contents

“For there is nothing either good or bad, but thinking makes it so.”

William Shakespeare (1564-1616)

1 Introduction

Due to the rising complexity of systems it is natural to ask for ways to model them on an intuitive level and analyse them efficiently. Many concurrent and distributed systems, especially those with a dynamically evolving topology, can be modelled by graphs and graph transformation rules. While graph transformation leads to a natural way to model dynamically evolving systems, the question arose, how to verify these systems. Work on the verification of dynamic, graph-like structures has shown that they introduce an additional level of complexity, compared to rule-based systems where states have either a word or tree structure. But since these latter structures posses a well-established theory that has been successfully used for verification in the past, the main idea is to generalize rewriting techniques from strings and trees in the theory of formal languages to the setting of graph-like structures.

1.1. Context

The theory of formal languages plays an important role in computer science and there exists a large number of applications for this theory, for instance in the design of communication protocols, compiler construction and parsing. The theory can be efficiently used to represent states as sets of words or trees and it offers symbolic manipulation techniques, for instance in form of grammars, to rewrite and analyse the specified context. In this way the theory can also be applied for verification purposes. Here we concentrate on automata/formal-language based verification techniques, i.e. analysis techniques used in the class of regular languages. In verification some typical methods are (non-)termination analysis [EZ15; GHW04], reachability analysis [FO97], regular model checking [BJ+00] and counterexample-guided abstraction refinement [CG+03]. Using reachability analysis for example, one can prove the absence of erroneous states.

While the theory of formal languages is worked out very well in string and tree/term rewriting, it is often non-trivial to solve the same problems when it comes to graph rewriting. Therefore, it is natural to ask for generalizations of these verification techniques to the framework of graph rewriting and additionally to a theory of graph languages, where these techniques can be applied. The analysis of pointer structures, in the research field of heap analysis, is just one example, where the adequate specification of sets of graphs in combination with verification techniques is needed. For this purpose, one needs a specification formalisms for graph languages with suitable closure properties, positive results for decidability problems (such as membership, language inclusion and emptiness) and computable pre- and postconditions. Instead of just tinkering with fitting existing specification formalisms for any given verification problem, we try to achieve a different main goal here: One contribution of this thesis is help to understand the essence of some selected graph specification languages, which grant them the possibility to adapt the verification techniques.

The Type Graph Framework

We focus on specification languages based on *type graphs*, where the language of a type graph T consists of all graphs that can be mapped homomorphically into T (with potentially extra constraints to extend the framework). Many specification formalisms that are usually used in abstract graph transformation [SWW11] and verification, are based on type graphs. Usually, one assumes that the rules and the graphs to be rewritten are typed. This idea serves the purpose of introducing constraints on the applicability of the rules and therefore type graphs can be understood as a form of labelling. However, this is different from the point of view used throughout this thesis, where graphs and rules remain untyped (even while working with labelled graphs) and the type graphs are simply meant to represent a possibly infinite set of graphs. Type graphs retain a nice intuition from regular languages when it comes to specifying graph languages. The language of a given finite state automaton M can be interpreted as the set of all string graphs that can be mapped homomorphically to M (respecting initial and final states).

Double-Pushout Graph Rewriting

The rewriting formalism for graphs and graph-like structures that we use throughout this thesis is the double-pushout (DPO) approach [CM+97]. Although it was originally introduced for graphs [EPS73], it is well-defined in any category. However, certain standard results for graph rewriting require that the category has "good" properties. The category of graphs is an elementary topos—an extremely rich categorical structure—but weaker conditions on categories, for instance adhesivity, have been studied [LS05; EH+04; EGH+13]. Since we are interested in the verification of graphs which may model specific systems, the advantage in using DPO lies in the fact that deletion in unknown contexts is forbidden per default. Therefore, by using DPO instead of other approaches like single-pushout (SPO), we can ensure that the application of our rules never cause unwanted side-effects, which could lead to inadequate models of the described system.

Application Scenarios for Graph Specification Languages

In order to better motivate our approach, we will explain how specification languages for graphs can help in system verification. Assume that we are given a graph transformation system, specified by a set $\mathcal{R}$ of DPO rules [Ehr79], which generates a transition system on graphs. A transition between two graphs G, H is denoted by $G \Rightarrow_{\mathcal{R}} H$. Then we can consider the following application scenarios:

Invariant Analysis Assume that we are given a graph language $\mathcal{L}$. The aim is to show that for every transition $G \Rightarrow_{\mathcal{R}} H$ with $G \in \mathcal{L}$ it always holds that $H \in \mathcal{L}$. We also say that $\mathcal{L}$ is *closed under rewriting.*

One way to show this is to compute the strongest postcondition of $\mathcal{L}$ wrt. $\mathcal{R}$, i.e., $Post_{\mathcal{R}}(\mathcal{L}) = \{H \mid \exists G \in \mathcal{L} \colon G \Rightarrow_{\mathcal{R}} H\}$ and prove that $Post_{\mathcal{R}}(\mathcal{L}) \subseteq \mathcal{L}$.

Reachability Analysis Given a fixed language $\mathcal{I}_0$ of initial graphs, the aim is to compute all graphs which are reachable from $\mathcal{I}_0$ in any number of steps. One can compute $\mathcal{I}_{i+1} = \mathcal{I}_i \cup Post_{\mathcal{R}}(\mathcal{I}_i)$ for successive indices i and terminate whenever $\mathcal{I}_{m+1} = \mathcal{I}_m$ for some m. Since in an infinite state space such analyses usually do not terminate in a finite number of steps, it is necessary to use widening respectively overapproximation techniques to ensure termination.

Termination Analysis The aim is to check if a given graph transformation system $\mathcal{R}$ is uniformly terminating. For this we specify the language $\mathcal{L}$ of all possible graphs and assign weights, which are elements of a well-founded relation, to the graphs $G \in \mathcal{L}$ to be rewritten. Afterwards, one shows that the assigned weight of the graph decreases with every rule application.

Non-Termination Analysis Here we ask whether there exists any graph G from which there is an infinite sequence of rewriting steps, i.e., whether there are non-terminating computations. A solution to this problem given in [EZ15] (for term rewriting systems) is to find a non-empty language $\mathcal{L}$ of graphs such that (i) every $G \in \mathcal{L}$ contains at least one left-hand side, i.e., a rewriting step is possible; (ii) for every $G \in \mathcal{L}$, whenever $G \Rightarrow_{\mathcal{R}} H$ then $H \in \mathcal{L}$ holds, i.e., the language $\mathcal{L}$ is closed under rewriting for all rules in the rewriting system $\mathcal{R}$. With these two properties one can prove that from every graph in $\mathcal{L}$ there exists a non-terminating sequence of rewriting steps.

Counterexample-Guided Abstraction Refinement The well-known CEGAR approach (see for instance [HJ+04]) is a static analysis technique which starts with a coarse initial abstraction which is then refined step-by-step by eliminating spurious counterexamples. One starts with a finite set of local formulas to abstract the state space. Then one looks for spurious counterexamples (i.e. runs that exist in the abstraction, but not in reality) in order to generate additional logical formulas and to refine the abstract state space. Instead of a logic it is in principle also possible to use other specification mechanisms. We will not go into detail, but CEGAR requires computation of strongest postconditions/weakest preconditions, inclusion and so-called Craig interpolation.

In order to actually implement a scenario as above, one needs the required constructions (computation of postconditions, ...), decision procedures (closure under rewriting, inclusion, ...) and closure properties (union, ...). On the other hand, if a specification language with the required properties is provided, one can implement all the procedures described above where the methods are called as “black boxes”, without any need to know what is going on under the hood.

Needless to say that expressiveness, decidability and efficiency are also a major issue. The current state-of-the-art is such that for graph transformation there is no single specification language that is suitable for all such purposes.

1.2. Contributions

In the following we illustrate the main contributions of this thesis:

– *(Part II) Termination Analysis of Graph Transformation Systems.* Proving the termination property of a rewriting system, e.g. the absence of rewriting or derivation sequences of infinite length, is an undecidable problem in general [Plu98]. Nonetheless, given a rewriting system (for instance in graph rewriting), one can try several methods in parallel to possibly find a solution for the specific termination problem. One possible approach of proving termination is to construct a monotone function that measures structural properties of the graphs to be rewritten. Afterwards one shows that the value of such a function (or weight assigned to the graph) decreases with every rule application. This is usually achieved by computing the weights directly on the left-hand side and right-hand side of every rule in the rewriting system.

We introduce a technique based on type graphs which are weighted over different kinds of semirings, to check if a given graph transformation system is uniformly terminating, i.e. independently of the initial graph the rules of the system can only be applied a finite number of times. This technique was inspired by an existing method based on matrix interpretations for proving termination in string, cycle and term rewriting systems [EWZ08]. The type graph is used to specify the set of all possible graphs by finite means and at the same time assign weights to the graphs to be rewritten. Depending on the semiring chosen for the computation, we are able to prove termination for graph transformation systems consisting of rules that can be applied up to an exponential number of times.

The new termination analysis technique has been implemented (among others) in a prototype Java-based tool named *Grez*. The tool concurrently runs several algorithms to prove the termination of a given graph transformation system. *Grez* is able to employ an SMT solver, to solve inequalities resulting from this method. The inequalities encode all possible morphisms from the given rule graphs (both left- and right-hand side) into potential weighted type graph candidates. The variables, used in these encodings, represent weights for each element of the type graph. Therefore, whenever the SMT solver returns a valid solution for the inequalities, it gives rise to the weights assigned to the type graph such that it becomes a witness for the termination proof.

Furthermore, we translate term rewriting systems from the *Termination Problems Database* (TPDB) into graph transformation systems and let *Grez* automatically prove termination. We investigate two different encodings (namely the function and number encoding) in two possible rewriting interpretations (called basic and extended version) of term rewrite rules into graph transformation rules that preserve the termination property, i.e. whenever the graph transformation system terminates, so does the term rewriting system.

– *(Part III) Specifying Graph Languages.* We analyse decidability and closure properties for graph languages specified by type graphs. While not being as expressive as recognizable graph languages, we prove positive results with respect to decidability problems for the two simplest cases of specification formalisms, namely *type graph languages* and *restriction graph languages.* A *type graph language* contains all graphs which allow a homomorphism into a given type graph, whereas a *restriction graph language* includes all graphs that do *not* contain an homomorphic image of a given type graph. We extend the formalism in two different ways: First, we introduce boolean connectives between type graphs to generate a type graph logic and second, we increase the expressiveness of the type graph itself, by adding annotations to the type graph elements.

In case of the *type graph logic*, one already obtains the desired closure properties for free since they are semantically given by the logical conjunction, disjunction and negation operators. However, it is still impossible to compute postconditions within this formalism. This is due to the fact that one can not express the existence of a subgraph (here the right hand-side graph from a graph transformation rule) in every graph contained in the specified graph language.

Therefore, we define a framework of *annotated type graphs*, to generate an abstract framework, from which formalisms based on type graphs can be instantiated. Each type graph is enriched with a set of annotations, and annotations can be parametrized. For instance, in one of the settings, the annotations are used to globally count all elements that can be mapped to the elements in the type graph. This is different from UML multiplicities, which are locally specified on the edges.

By adding annotations to the type graph, the expressiveness is too powerful, such that the language inclusion problem becomes hard to decide. We only obtain positive results for the language inclusion problem by restricting the analysed graph languages to graphs up to a given pathwidth (equivalent to [Blu14]). However, by adding annotations to the formalism, we are able to compute postconditions of rule applications, which was impossible in the other refinements of the type graph specification language. In addition, we investigate closure under rule application, i.e. invariant checking for our frameworks.

– *(Part IV) Abstract Object Rewriting.* Finally, by exploiting universal properties from category theory, we introduce a *materialization* construction (similar to [SRW02]) for our annotated type graph framework. A basic observation is that in most specification frameworks an abstract rewriting step is performed by computing the (strongest) postcondition in two steps: by first materializing the left-hand side of the rule to be applied (also called shift in some specification formalisms), followed by adding the right-hand side (existentially quantified). Therefore, the notion of annotated type graphs is lifted to a more general framework of *annotated abstract objects* in an arbitrary *topos.*

For the purely structural aspects of our materialization construction we will use partial map classifiers in a topos and its slice categories. We furthermore relate the construction to the fundamental construction of final pullback complements [DT87]. Even though the materialization fully specifies the set of objects which explicitly contain a copy of the left-hand side of a rule, there still can be objects in this set which can not be rewritten. Therefore, we refine the materialization into a *rewritable materialization*, which specifies the set of all rewritable objects with respect to the rule to be applied. However, the abstract object retrieved by the abstract derivation step can only be used to specify the strongest postcondition as long as the information about the explicit copy of the right hand-side is given. To solve this issue we explain how all objects in the construction can be endowed with annotations and how these annotations can be rewritten.

We give properties for the annotations which need to be satisfied to be able to compute the strongest postconditions in this generalized abstract setting.

Finally, being able to compute postconditions for the specification of graph languages by using annotated type graphs, we implement verification techniques for this formalism in a prototype Java-tool called DrAGoM. We benchmark the techniques of the tool with respect to some worked examples.

1.3. Structure of this Thesis

The structure of this thesis is as follows[1]:

The thesis begins with a preliminaries and foundations part (Part I) to remind the reader of basic mathematical definitions. The notation that is going to be used is fixed and concepts related to category theory and graph transformation are recalled as a preparation for later parts to come. The main contributions of this thesis are split into the three main Parts II-IV, each having their own motivation and conclusion section. This way, the parts can be read independently[2] and in any order, while each of them contribute to a larger research context, namely the analysis and abstraction of graph transformation systems via type graphs. The theoretical results, introduced in the Chapters 8-10, have been implemented into DrAGoM, a tool which is introduced and evaluated in Part V. In Part VI the thesis ends with a conclusion where all contributions are evaluated in the overall context. All proofs, experimental results on termination analysis, a nomenclature and an index are given in the appendix of this thesis.

The content of the chapters (grouped by their respective part) is the following:

Part I – Preliminaries and Foundations

Chapter 2 – Foundations

In Chapter 2 we recall and fix mathematical notations. The definitions in this chapter are used throughout the thesis, whereas additional preliminary sections in the different parts might extend these basic concepts. In order to establish the background theory (especially needed in Part IV), we additionally give an introduction to some basic notions of category theory at the end of the chapter.

Chapter 3 – Graphs and Graph Transformation

This chapter introduces the kind of graphs that will be used in this thesis. Furthermore we give a tutorial on graph transformation that explains the so-called double-pushout approach to graph transformation in a rigorous, but non-categorical way, using a gluing construction. Afterwards, we relate this gluing construction to its categorical counterpart.

Chapter 4 – Type Graph Languages

In this chapter we learn how type graphs can be used to specify (possibly infinite) sets of graphs by finite means. We are interested in (pure) type graphs, where the corresponding language consists of all graphs that can be mapped homomorphically to a given type graph. After giving the formal definition of type graph languages we present several examples of special cases of these languages.

[1] See also the dependency graph in Figure 1.1 on page 9

[2] Exception: It is recommended (but not required) to read Chapter 8 of Part III before reading Chapter 10 of Part IV, as the introduced concepts are related to each other.

Part II – Termination Analysis of Graph Transformation Systems

Chapter 5 – Weighted Type Graphs over Semirings

This chapter presents techniques, based on so-called weighted type graphs, for proving uniform termination of graph transformation systems. These type graphs can be used to assign weights to graphs and to show that these weights decrease in every rewriting step in order to prove termination. We present an example involving counters and discuss the implementation in a tool called Grez.

Chapter 6 – Terms, Term Rewriting and Term Graph Encodings

In Chapter 6 we discuss two natural ways to interpret term rewrite rules as term graph rewrite productions. Afterwards we introduce an approach to transform term graph rewriting to graph transformation, in such a way that termination of the term graph rewrite system can be concluded from termination of the resulting graph transformation system, to be proved by Grez. We propose two such transformations: the function encoding and the number encoding. We discuss the two transformations and report about experiments.

Part III – Specifying Graph Languages

Chapter 7 – Pure Type Graphs, Restriction Graphs and Type Graph Logic

In this chapter we investigate two more formalisms for specifying type graph languages, i.e. sets of graphs, based on type graphs. First, we study languages specified by restriction graphs and their relation to type graphs. Second, we extend this basic approach to a type graph logic. We present decidability results and closure properties for both formalisms.

Chapter 8 – Annotated Type Graphs

In Chapter 8 we endow type graphs with annotations, thus making graph languages more expressive. In particular we will use ordered monoids in order to annotate graphs. Similar to the previous chapter, we present decidability results and closure properties. This time we put a little more focus on the language inclusion problem, for which we introduce the notion of a counting cospan automaton functor to solve the problem.

Part IV – Abstract Object Rewriting

Chapter 9 – Materialization Category

In this chapter we focus on the so-called materialization of left-hand sides from abstract objects, a central concept in abstract rewriting. We have a look at an accessible, general explanation of how materializations arise from universal properties and categorical constructions, in particular partial map classifiers, in a topos. Furthermore, we refine the materialization construction to a rewritable materialization by exploiting the notion of final pullback complements.

Chapter 10 – Rewriting Annotated Objects

In Chapter 10 we combine the previously introduced materialization construction with the concept of enriching graphs with annotations from ordered monoids to create a framework of abstract rewriting for annotated objects. We define properties of annotations which can be used to give a precise characterization of strongest postconditions, which are effectively computable under certain assumptions.

Part V – Tools and Applications

Chapter 11 – DrAGoM

This chapter gives an overview over the prototype tool DrAGoM which is a tool to handle and manipulate multiply annotated type graphs. The main application of DrAGoM is to automatically compute strongest postconditions to check invariants of graph transformation systems in the framework of abstract graph rewriting. We have a look at the basic functionalities and give an overview of a typical user interaction with the software. Afterwards, we investigate how the categorical notions of the previous chapters can be implemented and have a look at other tool approaches.

Chapter 12 – Evaluation

In this chapter we present different case studies which we have conducted using the tool DrAGoM. First, we compare the generated output with the expected results of examples from previous chapters. Second, we provide runtime results for several invariant checks and stress test the tool with respect to language inclusion checks of increasing graph sizes. At the end of the chapter, an overview of the case study results and a summary of the tool's practicability are provided.

Part VI – Conclusion

Chapter 13 – Conclusion and Future Work

In Chapter 13 we draw the thesis to a close. We summarize the main theoretical contributions and discuss how they fit into the broader scientific context of this thesis. Finally, we provide suggestions of potential next steps for future work, resulting from the contribution.

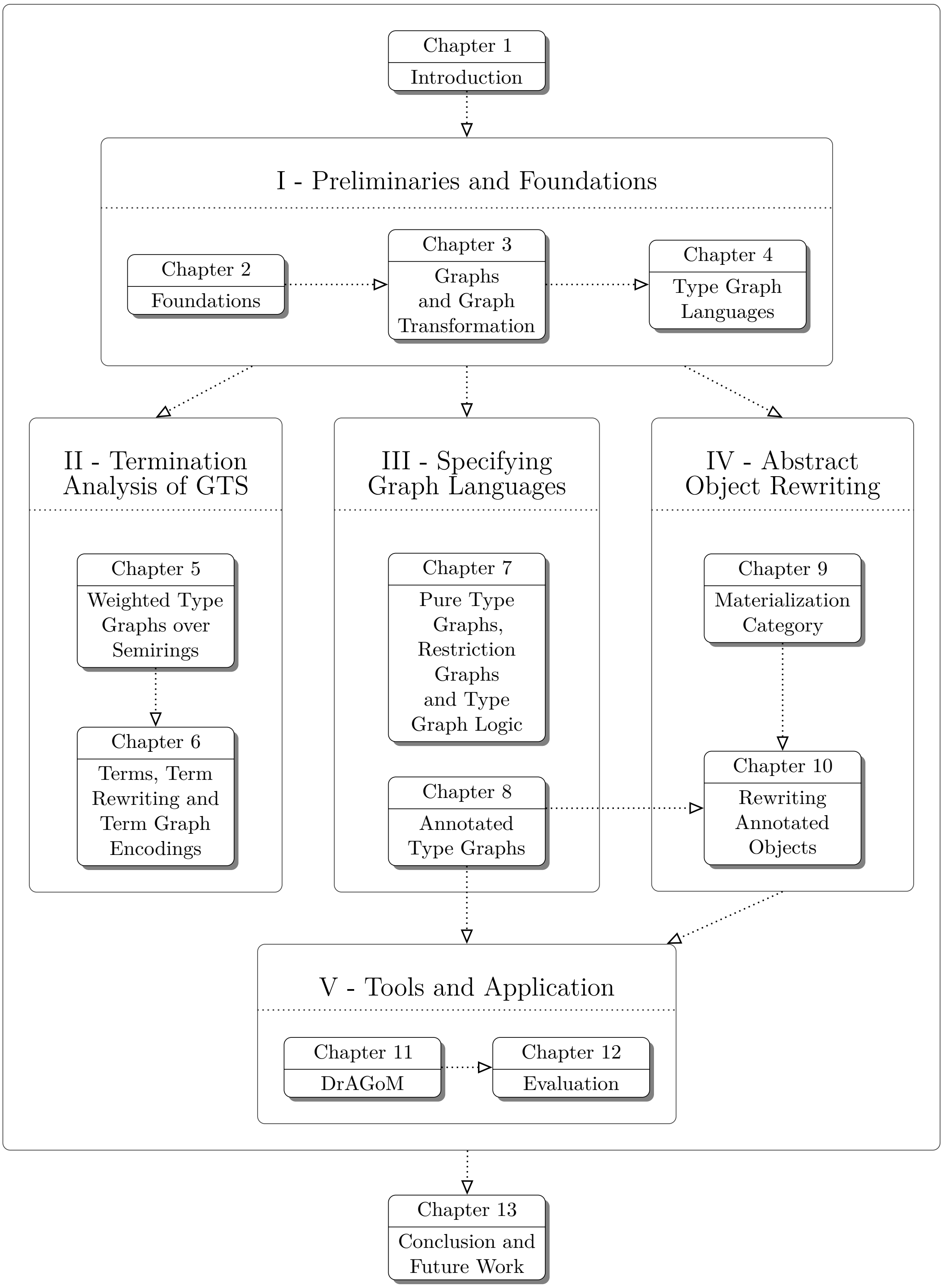

Chapter 1
Introduction
I - Preliminaries and Foundations
Chapter 2
Foundations
Chapter 3
Graphs and Graph Transformation
Chapter 4
Type Graph Languages
II - Termination Analysis of GTS
Chapter 5
Weighted Type Graphs over Semirings
Chapter 6
Terms, Term Rewriting and Term Graph Encodings
III - Specifying Graph Languages
Chapter 7
Pure Type Graphs, Restriction Graphs and Type Graph Logic
Chapter 8
Annotated Type Graphs
IV - Abstract Object Rewriting
Chapter 9
Materialization Category
Chapter 10
Rewriting Annotated Objects
V - Tools and Application
Chapter 11
DrAGoM
Chapter 12
Evaluation
Chapter 13
Conclusion and Future Work

Figure 1.1.: Dependency graph of this thesis. The dotted arrows visualize prerequisites with respect to the chapters. It is recommended to read the chapter(s) at the source of an arrow first before reading its target.

Part I.

Preliminaries and Foundations

"Do not worry about your difficulties in Mathematics.
I can assure you mine are still greater."
Albert Einstein (1879-1955)

2
Foundations

In this chapter we will remind the reader of basic mathematical definitions and afterwards give a short introduction to a mathematical formalism called category theory, which is an alternative to set theory.

Please note, that the purpose of this chapter is not to give a full introduction to basic mathematical concepts nor to category theory, but rather fix the notation that is going to be used. Therefore, it is assumed that the reader at least has a thorough mathematical and computer science background on the level of an undergraduate degree at a university. At the same time, by describing the necessary concepts, the thesis becomes self-contained such that the consultation of other literature is not mandatory, to be able to understand the upcoming chapters.

Likewise, for readers who are familiar with standard mathematical/categorical concepts and the basics of graph rewriting, it is possible to skip this Part I of the thesis and immediately proceed to one of the main Parts II-IV, without missing any important results. A complete list of used symbols together with an index is given at the end of the thesis.

2.1. Basic Notation

We start by fixing the notation of the fundamental logical statements which are being used in computer science. Afterwards, we recall the basic concepts of set theory and conclude this subsection with a reminder about the definitions of some mathematical structures.

Logical Operators

For the standard logical operations we will use the following symbols:

$\wedge$	for *conjunction*	$\vee$	for *disjunction*
$\Longrightarrow$	for *implication*	$\Longleftrightarrow$	for *bi-implication*
$\exists$	for *existential quantification*	$\forall$	for *universal quantification.*

Set

The *membership relation* is denoted by $\in$, i.e. whenever an element x is a member of a set X we will simply write $x \in X$. We use $X = Y$ to denote that two sets are equal, $X \subseteq Y$ to denote that X is a *subset* of Y or equal and we use $X \subset Y$ to denote that X is a *strict subset* of Y, i.e. inclusion holds but the sets are not equal. Their corresponding negations are denoted by $\notin, \neq, \not\subset$ and $\nsubseteq$. We denote by $\mathbb{N}_0$ the set of *natural numbers* $\{0, 1, 2, \ldots\}$, including 0, and by $\mathbb{N}$ the natural numbers without 0. The symbol $\emptyset$ is used to denote the *empty set*, i.e. a set without any element. The *powerset* of X is denoted by $\mathcal{P}(X)$.

For two sets X and Y, the *relative complement* is denoted by $X \setminus Y$, i.e. $X \setminus Y = \{x \in X \mid x \notin Y\}$, the *union* is denoted by $X \cup Y$, i.e. $X \cup Y = \{z \mid z \in X \vee z \in Y\}$ and the *intersection* is denoted by $X \cap Y$, i.e. $X \cap Y = \{z \mid z \in X \wedge z \in Y\}$. The *cartesian product* $X \times Y$ is the set $X \times Y = \{(x, y) \mid x \in X \wedge y \in Y\}$ of ordered pairs, which are denoted by round parentheses. For $n \in \mathbb{N}$, $X^n = X \times \ldots \times X$, denotes the *n-ary cartesian product* of X. The *disjoint union* $X_1 \biguplus X_2$ for two sets X_1 and X_2 is the set $X_1 \biguplus X_2 = \bigcup\limits_{i \in I} \{(x, i) \mid x \in X_i\}$ with $I = \{1, 2\}$.

Relation

For an arbitrary set X a subset $R \subseteq X \times X$ is called *(homogeneous) relation* on X. We denote by R^{-1} the *inverse* relation of R. The relation R is *reflexive* if and only if for all $x \in X$ we have $(x, x) \in R$; it is *transitive* if and only if for all $x, y, z \in X$ we have $(x, y) \in R \wedge (y, z) \in R \implies (x, z) \in R$; it is *symmetric* if and only if for all $x, y \in X$ we have $(x, y) \in R \implies (y, x) \in R$ and it is *antisymmetric* if and only if for all $x, y \in R$ we have $(x, y) \in R \wedge (y, x) \in R \implies x = y$. The *transitive closure* of R is denoted R^+, i.e. the smallest relation on X that contains R and is transitive.

Order

For an arbitrary set X, a *preorder* $\leq$ on X is a binary relation on X which is reflexive and transitive. A preorder $\leq$ is *total* if for all $x, y \in X$ either $x \leq y$ or $y \leq x$ (or both) holds. If a preorder $\leq$ is antisymmetric it is called a *partial order*. If a preorder $\leq$ is symmetric it is called an *equivalence* and will be denoted by $\equiv$. We denote by $X/\equiv$ the *quotient set* of all equivalence classes of X by $\equiv$. Furthermore $[x]_\equiv$ denotes the *equivalence class* of $x \in X$ with respect to $\equiv$, i.e. $[x]_\equiv = \{y \in X \mid x \equiv y\}$. In the following we omit the subscripts in the equivalence class $[x]_\equiv$ and simply write $[x]$ whenever $\equiv$ is clear from the context. If $\leq$ is an order, then we denote by $<$ its strict subrelation e.g. $x < y$ if and only if $x \leq y \wedge x \neq y$. An order is *well-founded* if it does not allow infinite, strictly decreasing sequences $x_0 > x_1 > x_2 > \cdots$.

Function

A binary relation $f \subseteq X \times Y$, which satisfies the requirement that for all $x \in X$ there exists a unique $y \in Y$ such that $(x, y) \in f$ holds, is called a (total) *function* from X to Y and is denoted by $f\colon X \to Y$. For a function $f\colon X \to Y$ we call X the *domain*, Y the *codomain* and given any $x \in X$ we write $f(x)$ (instead of y) for the unique element satisfying $(x, f(x)) \in f$. We call $f(x)$ the *image* of $x \in X$ under $f\colon X \to Y$ and say that $y \in Y$ has a *preimage* under $f\colon X \to Y$ if there exists an $x \in X$ with $f(x) = y$. A function $f\colon X \to Y$ is *injective* if for all $x_1, x_2 \in X$ we have $f(x_1) = f(x_2) \implies x_1 = x_2$ and it is *surjective* if every $y \in Y$ has a preimage under f. A function $f\colon X \to Y$ is *bijective* if it is both, injective and surjective, and we denote by $f^{-1}\colon Y \to X$ the inverse of a bijective function. The *domain restriction* of a function $f\colon X \to Y$ to a set $Z \subseteq X$ is denoted by the function $f|_Z\colon Z \to Y$ with $f|_Z = f \cap Z \times Y$.

Monoid

For a non-empty set X, a binary operator $\oplus\colon X \times X \to X$ and an element $e \in X$ a 3-tuple $(X, \oplus, e)$ is called *monoid* if the operator $\oplus$ is associative, i.e. for all $x, y, z \in X$ we have $x \oplus (y \oplus z) = (x \oplus y) \oplus z$ and e is the *unit* with respect to $\oplus$, i.e. for all $x \in X$ we have $e \oplus x = x \oplus e = x$. A monoid $(X, \oplus, e)$ is called a *commutative monoid* if the operator $\oplus$ is commutative, i.e. for all elements $x, y \in X$ we have $x \oplus y = y \oplus x$.

Lattice

Let $\leq$ be a preorder and X, Y be two sets with $Y \subseteq X$. An *upper bound* of Y is an element $x \in X$ such that for all $y \in Y$ we have $y \leq x$. An upper bound $x \in X$ is called *least upper bound* (or *join*; or *supremum*) if for each upper bound $u \in X$ we have $x \leq u$. Likewise, a *lower bound* is an element $x \in X$ such that for all $y \in Y$ we have $x \leq y$. A lower bound $x \in X$ is called *greatest lower bound* (or *meet*; or *infimum*) if for each lower bound $\ell \in X$ we have $\ell \leq x$. We denote the least upper bound of a set by $\bigvee Y$ and the greatest lower bound by $\bigwedge Y$ if it exists. For a set consisting of only two elements $y_1, y_2 \in Y$ we will sometimes write $y_1 \vee y_2$ instead of $\bigvee\{y_1, y_2\}$ for the least upper bound and $y_1 \wedge y_2$ instead of $\bigwedge\{y_1, y_2\}$ for the greatest lower bound. Note that a set may have many upper/lower bounds, or none at all, but at most one least upper/greatest lower bound.

A preordered set $(X, \leq)$ is called a *lattice* if for all subsets $Y \subseteq X$ there exists a least upper bound $\bigvee Y$ and a greatest lower bound $\bigwedge Y$. Moreover, if X is finite, then X has a unique minimal element $\bot = \bigwedge X$ (called *bottom*) and a unique maximal element $\top = \bigvee X$ (called *top*). For the special case $Y = \emptyset$ we have $\bigvee \emptyset = \bot$ and $\bigwedge \emptyset = \top$.

2.2. Basic Category Theory

Category theory is a mathematical framework which is used to describe abstract structures. The theory does not focus on elements, like it is done in set theory, but rather focuses on collections of elements (here called *objects*) and the relations between these collections (also called *arrows* or *morphisms*). In this chapter we have a look at the basic categorical concepts which are used throughout this thesis. Please note that the definitions and explanations given in this chapter are not meant to give a full overview to the topic of category theory but rather covers the essential constructions needed in this thesis. At the end of this chapter we give some literature recommendation for the interested reader, who would like to learn more about category theory. We start by defining what a category is.

Definition 2.1 (Category)**.** A *category* is a 4-tuple $\mathbf{C} = (\mathcal{O}, \mathcal{M}, \circ, id)$ which consists of

- a class $\mathcal{O}$ whose elements are called *objects* (or **C**-*objects*).
- a class $\mathcal{M}(A, B)$ for all objects $A, B \in \mathcal{O}$ whose elements are called *arrows* or *morphisms* (sometimes referred to as **C**-*arrows*/**C**-*morphisms*). Each morphism $f \in \mathcal{M}(A, B)$ has a *domain* $dom(f) = A$ alongside a *codomain* $cod(f) = B$ and we will write $f \colon A \to B$.
- a *composition function* $\circ \colon \mathcal{M}(B, C) \times \mathcal{M}(A, B) \to \mathcal{M}(A, C)$ for all objects $A, B, C \in \mathcal{O}$ which assigns to any morphisms $f \in \mathcal{M}(A, B)$ and $g \in \mathcal{M}(B, C)$ their composed morphism $g \circ f \colon A \to C$. Furthermore, the composition of morphisms $f \colon A \to B, g \colon B \to C$ and $h \colon C \to D$ must be associative, i.e. $(h \circ g) \circ f = h \circ (g \circ f)$.
- a class id of *identity* morphisms with $id_A \in \mathcal{M}(A, A)$ for all objects $A \in \mathcal{O}$. The identity morphisms must be the neutral elements with respect to composition, i.e. for all morphisms $f \colon A \to B$ it holds that $f \circ id_A = f$ and $id_B \circ f = f$.

Example 2.2. *The classic example of a category is* **Set***. This category consists of sets as objects and functions as morphisms, and the composition operation is the usual composition of functions. Another example is the category* **Rel** *where the objects are sets, but in contrast to* **Set***, the morphisms are (binary) relations which do not need to be functions.*

We will use the category **Set** as a running example in this chapter to provide the reader, who is not familiar with category theory, some intuition behind the concepts which are described in the following.

Definitions and proofs in category theory extensively use the notion of *commuting diagrams*. Diagrams can be visualized as directed graphs, which makes it easier for the reader to "chase" required commuting properties within the visualisation.

Definition 2.3 (Diagram)**.** Let **C** be a category. A *diagram* in **C** is a subclass of **C**-objects $\mathcal{O}'$ and a subclass of **C**-morphisms $\mathcal{M}'$, where for every $f \in \mathcal{M}'$ we have $dom(f) \in \mathcal{O}'$ and $cod(f) \in \mathcal{O}'$. A diagram *commutes* if for every two well defined sequences of compositions $f_1 \circ \ldots \circ f_n$ and $g_1 \circ \ldots \circ g_m$ of morphisms in $\mathcal{M}'$ where $dom(f_n) = dom(g_m)$ and $cod(f_1) = cod(g_1)$ we have $f_1 \circ \ldots \circ f_n = g_1 \circ \ldots \circ g_m$.

Example 2.4. *Let the following diagram $\mathcal{D}$ consist of the objects $\{A, B, C, D, E\}$ and arrows $\{f \colon A \to B, g \colon A \to C, h \colon B \to D, i \colon C \to D, j \colon B \to E, k \colon E \to D\}$. The diagram $\mathcal{D}$ (depicted below right) commutes if and only if the following two equations hold:*

$$h \circ f = i \circ g \qquad \textit{and} \qquad k \circ j = h.$$

As a consequence from above equations we get

$$i \circ g = h \circ f = k \circ j \circ f$$

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] & B \arrow[d, "h"'] \arrow[dr, "j"] & \\
C \arrow[r, "i"] & D & E \arrow[l, "k"]
\end{tikzcd}$$

With category theory focusing on morphisms rather than elements, there exist special types of morphisms, which play important roles in their categories.

Definition 2.5 (Monomorphism, Epimorphism, Isomorphism)**.** Let **C** be a category and $f\colon A \to B$ be a **C**-morphism.

- f is called *monomorphism* (or *mono*), if for all morphisms $g_1\colon X \to A$ and $g_2\colon X \to A$ with $f \circ g_1 = f \circ g_2$ it follows that $g_1 = g_2$.
- f is called *epimorphism* (or *epi*), if for all morphisms $h_1\colon B \to Y$ and $h_2\colon B \to Y$ with $h_1 \circ f = h_2 \circ f$ it follows that $h_1 = h_2$.
- f is called *isomorphism* (or *iso*), if there exists a morphism $g\colon B \to A$ such that $g \circ f = id_A$ and $f \circ g = id_B$.

Example 2.6. *In the category* **Set***, the monomorphisms are injective functions, epimorphisms are surjective functions and isomorphisms are bijective functions.*

We will denote monomorphisms by $A \rightarrowtail B$, epimorphisms by $A \twoheadrightarrow B$ and isomorphisms by $A \xrightarrow{\sim} B$.

Category theory gives rise to categorical constructions which can be used to either create new categories based on given categories or define objects via the concept of *universal properties*. By using universal properties for object specifications, category theory remains abstract in the sense that it describes objects by their properties instead by the way how they are constructed. This leads to the ability to reuse constructions in several categories.

In this thesis, we will use the notion of *products* for some constructions. Intuitively, the product of two objects is the most general object which admits a morphism to each of them.

Definition 2.7 (Product)**.** Let **C** be a category with some objects A_1 and A_2. A *product* of A_1 and A_2 is an object $A_1 \times A_2$ together with a pair of morphisms $\pi_1\colon (A_1 \times A_2) \to A_1, \pi_2\colon (A_1 \times A_2) \to A_2$ (called *projection morphisms*) such that the following universal property is fulfilled:
For every object X and pair of morphisms $f_1\colon X \to A_1, f_2\colon X \to A_2$ there exists a unique morphism $f\colon X \to (A_1 \times A_2)$ such that $\pi_1 \circ f = f_1$ and $\pi_2 \circ f = f_2$, i.e., the diagram to the right commutes.

$$\begin{tikzcd}
& X \arrow[ld, "f_1"'] \arrow[d, dashed, "f"] \arrow[rd, "f_2"] & \\
A_1 & A_1 \times A_2 \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & A_2
\end{tikzcd}$$

Example 2.8. *In the category* **Set***, the product is the* cartesian product*.*

The category-theoretical dual notion of the product is the *coproduct*. Dual notions in category theory usually have the same definitions as their corresponding counterpart, with the difference that all morphisms are reversed. Essentially, the coproduct of two objects is the least specific object to which each of them admits a morphism. Products and coproducts, if they exist, are unique up to isomorphism.

Definition 2.9 (Coproduct)**.** Let $\mathbf{C}$ be a category with some objects A_1 and A_2. A *coproduct* of A_1 and A_2 is an object $A_1 \oplus A_2$ if there exist morphisms $i_1 \colon A_1 \to (A_1 \oplus A_2), i_2 \colon A_2 \to (A_1 \oplus A_2)$ (called *embedding morphisms* or *injection morphisms*) such that the following universal property is fulfilled: For every object X and pair of morphisms $f_1 \colon A_1 \to X, f_2 \colon A_2 \to X$ there exists a unique morphism $f \colon (A_1 \oplus A_2) \to X$ such that $f \circ i_1 = f_1$ and $f \circ i_2 = f_2$, i.e., the diagram to the right commutes.

$$\begin{tikzcd}
 & X & \\
A_1 \arrow[ur, "f_1"] \arrow[r, "i_1"'] & A_1 \oplus A_2 \arrow[u, dashed, "f"'] & A_2 \arrow[ul, "f_2"'] \arrow[l, "i_2"]
\end{tikzcd}$$

Example 2.10. *In the category* **Set***, the coproduct is the* disjoint union*.*

Another example for a special kind of object satisfying a universal property is given in the following definition of *initial*, and *terminal objects*.

Definition 2.11 (Terminal object and initial object)**.** A *terminal object* in a category $\mathbf{C}$ is an object $\mathbf{1}$ of $\mathbf{C}$ satisfying the following universal property: For every $\mathbf{C}$-object A, there exists a unique morphism $!_A \colon A \to \mathbf{1}$.

The dual concept to terminal objects are *initial objects*. An *initial object* in a category $\mathbf{C}$ is an object $\mathbf{0}$ of $\mathbf{C}$ satisfying the following universal property: For every $\mathbf{C}$-object A, there exists a unique morphism $?_A \colon \mathbf{0} \to A$.

Example 2.12. *In the category* **Set** *the terminal object is any one-element set and the initial object is the empty set $\emptyset$ (which only admits the empty function).*

An initial and a terminal object, if they exist, are unique up to unique isomorphism. Therefore, we can speak of *the* initial and *the* final object of a category. The terminal object will play an important role in several chapters of this thesis as the membership of an object to a language, i.e. a set of objects, is determined by the existence of a morphism, as we will see in Chapter 4. Therefore, using a terminal object which admits a morphism from any object, one can easily specify the set consisting of all objects.

As described above, category theory focuses on morphisms instead of plain objects. This concept can be used to abstract the already abstract notion by category theory itself. Thus, instead of just analysing the morphisms between objects in a category, category theory introduces *functors* as structure preserving morphisms between categories.

Definition 2.13 (Functor)**.** Let $\mathbf{C}$ and $\mathbf{D}$ be two categories. A *functor* $\mathcal{F} \colon \mathbf{C} \to \mathbf{D}$ from $\mathbf{C}$ to $\mathbf{D}$ assigns to each $\mathbf{C}$-object A a $\mathbf{D}$-object $\mathcal{F}(A)$ and to each $\mathbf{C}$-morphism $f \colon A \to B$ a $\mathbf{D}$-morphism $\mathcal{F}(f) \colon \mathcal{F}(A) \to \mathcal{F}(B)$ such that the following conditions are satisfied:

- $\mathcal{F}$ preserves composition, i.e. for all composable morphisms f and g it holds that $\mathcal{F}(f \circ g) = \mathcal{F}(f) \circ \mathcal{F}(g)$.
- $\mathcal{F}$ preserves identities, i.e. for all $\mathbf{C}$-objects it holds that $\mathcal{F}(id_A) = id_{\mathcal{F}(A)}$.

The *identity functor* of a category $\mathbf{C}$ is denoted by $Id_{\mathbf{C}}$.

The notion of functors can be abstracted again to the notion of *natural transformation*, i.e., morphisms between functors. We will define and use the notion of natural transformation in Chapter 9.

One of the main concepts of this thesis are graph transformation systems, i.e., sets of graph transformation rules. In Section 3.2 we will learn how a *gluing construction* can be used to rewrite graphs. The gluing construction can be generalized to the idea of *pushouts* in the sense of category theory. Therefore, we now introduce pushouts, being the fundamental constructions on which graph rewriting is based on. The relation between the notion of pushouts and the gluing construction will be explained in Section 3.2.2.

Pushouts are constructed from pairs of morphisms $f\colon A \to B$ and $g\colon A \to C$ which share the same domain. Such a pair of morphisms is also called *span* and we will sometimes simply write $B \xleftarrow{f} A \xrightarrow{g} C$ to indicate the span. The universal property of pushouts essentially states that the pushout is the most general way to complete a commutative square with two given morphisms.

Definition 2.14 (Pushout (PO))**.** Given morphisms $f\colon A \to B$ and $g\colon A \to C$ in a category $\mathbf{C}$, a *pushout* (D, f', g') over f and g is defined by a pushout object D and two morphisms $f'\colon C \to D$ and $g'\colon B \to D$ with $f' \circ g = g' \circ f$, such that the universal property is fulfilled: For all objects X and morphisms $h\colon B \to X$ and $k\colon C \to X$ with $k \circ g = h \circ f$, there is a uniqe morphism $x\colon D \to X$ such that $x \circ g' = h$ and $x \circ f' = k$, i.e. the diagram to the right commutes.

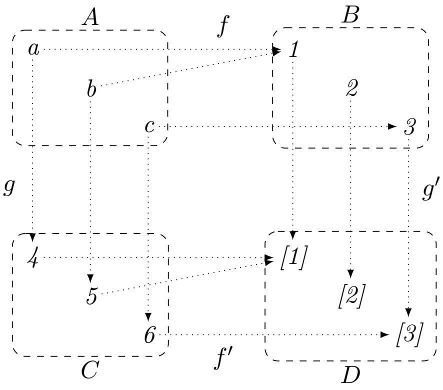
$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] & B \arrow[d, "g'"'] \arrow[ddr, "h"] & \\
C \arrow[r, "f'"] \arrow[drr, "k"'] & D \arrow[dr, dashed, "x"] & \\
& & X
\end{tikzcd}$$

Example 2.15. *In the category* $\mathbf{Set}$*, the pushout object* D *over the morphisms* $f\colon A \to B$ *and* $g\colon A \to C$ *can be constructed as the quotient set* $B \uplus C/\equiv$*, where* $\equiv$ *is the smallest equivalence relation with* $f(a) \equiv g(a)$ *for all* $a \in A$*. The pair of morphisms* $f'\colon C \to D$ *and* $g'\colon B \to D$ *is defined by* $f'(c) = [c]$ *for all* $c \in C$ *and* $g'(b) = [b]$ *for all* $b \in B$*.*

For instance, let the sets $A = \{a, b, c\}, B = \{1, 2, 3\}$ *and* $C = \{4, 5, 6\}$ *be given. Furthermore let* $f\colon A \to B$ *and* $g\colon A \to C$ *be defined as*

$$\begin{aligned}
f(a) &= 1 & \qquad g(a) &= 4\\
f(b) &= 1 & \qquad g(b) &= 5\\
f(c) &= 3 & \qquad g(c) &= 6.
\end{aligned}$$

The smallest equivalence relation $\equiv$ *yields the classes* $[1] = [4] = [5] = \{1, 4, 5\}$, $[2] = \{2\}$ *and* $[3] = [6] = \{3, 6\}$*. Therefore, the pushout object can be defined as the set* $D = \{[1], [2], [3]\}$ *alongside the two morphisms* $f'\colon C \to D$ *with* $f'(x) = [x]$ *and* $g'\colon B \to D$ *with* $g'(x) = [x]$*. The resulting pushout is depicted in the commuting diagram to the right.*



The generic definition of pushouts via universal properties can take various forms. For instance, another prototypical example is the supremum or join, where – given two elements x, y of a partially ordered set $(X, \leq)$ – we ask for a third element z with $x \leq z$, $y \leq z$ and such that z is the *smallest* element which satisfies both inequalities. There is at most one such z, namely $z = x \vee y$, the join of x, y.

The dual notion of pushouts are *pullbacks*. Pullbacks can be constructed from pairs of morphisms $f\colon C \to D$ and $g\colon B \to D$ which share the same codomain. These pairs are called *cospan* and are the dual notion of spans. Likewise, we will sometimes write $B \xrightarrow{g} D \xleftarrow{f} C$ to indicate the cospan. Pushouts and pullbacks, if they exist, are unique up to isomorphism.

Definition 2.16 (Pullback (PB))**.** Given morphisms $f\colon C \to D$ and $g\colon B \to D$ in a category $\mathbf{C}$, a *pullback* (A, f', g') over f and g is defined by a pullback object A and two morphisms $f'\colon A \to B$ and $g'\colon A \to C$ with $f \circ g' = g \circ f'$, such that the universal property is fulfilled: For all objects X and morphisms $h\colon X \to B$ and $k\colon X \to C$ with $f \circ k = g \circ h$, there is a uniqe morphism $x\colon X \to A$ such that $f' \circ x = h$ and $g' \circ x = k$, i.e. the diagram to the right commutes.

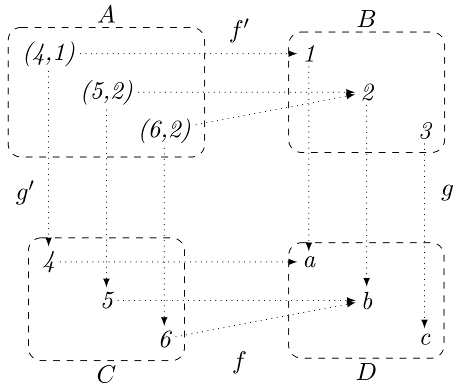
$$\begin{tikzcd}
X \arrow[rd, dashed, "x"] \arrow[rrd, "h"] \arrow[rdd, "k"'] & & \\
& A \arrow[r, "f'"] \arrow[d, "g'"] & B \arrow[d, "g"] \\
& C \arrow[r, "f"'] & D
\end{tikzcd}$$

Example 2.17. *In the category* **Set***, the pullback object A over the morphisms $f\colon C \to D$ and $g\colon B \to D$ is the set $A = \{(c, d) \mid f(c) = g(b)\} \subseteq C \times B$, i.e., A can be constructed as a subset of the cartesian product $C \times B$. The pair of morphisms $f'\colon A \to B$ and $g'\colon A \to C$ is defined as the corresponding projections, e.g., $f'((c, b)) = b$ and $g'((c, b)) = c$.*

For example, let the sets $B = \{1, 2, 3\}, C = \{4, 5, 6\}$ and $D = \{a, b, c\}$ be given. Furthermore let $f\colon C \to D$ and $g\colon B \to D$ be defined as

$$\begin{aligned} f(4) &= a & g(1) &= a \\ f(5) &= b & g(2) &= b \\ f(6) &= b & g(3) &= c. \end{aligned}$$

Then the pullback object is defined as the set $A = \{(4, 1), (5, 2), (6, 2)\}$ alongside the two morphisms $f'\colon A \to B$ with $f'((x, y)) = y$ and $g'\colon A \to C$ with $g'((x, y)) = x$. The pullback is depicted in the commuting diagram to the right.



A generalization of a pullback yields the notion of *limits*, as the limit of a cospan (if it exists) is a pullback. In the context of this thesis we do not use the notion of limit but will require the existence of finite limits in Chapter 9 to define a special class of categories named *Topos* (see Definition 9.4).

Another class of important categories are so-called *adhesive categories*. Many types of graphical structures which are being used in computer science are known to be examples of adhesive categories, including a category of graphs which we will extensively use later in this thesis. Adhesive categories provide a vast amount of structure ensuring properties, guaranteed to hold in any adhesive category, which can be used to analyse the structures within such a category more easily.

Definition 2.18 (Adhesive category)**.** A category $\mathbf{C}$ is *adhesive* if and only if the following three conditions are satisfied:

- $\mathbf{C}$ has pullbacks
- $\mathbf{C}$ has pushouts along monomorphisms
- pushouts of monomorphisms are pullbacks and pushouts are stable under pullbacks, i.e., given the cube depicted to the right, where the bottom face is a pushout along monomorphisms (also called *van Kampen square*) and the back faces are pullbacks: the front faces are pullbacks if and only if the top face is a pushout.

$$\begin{tikzcd}
& A' \arrow[rr, tail] \arrow[dl, tail] \arrow[dd, tail] & & B' \arrow[dl, tail] \arrow[dd, tail] \\
C' \arrow[rr, tail, crossing over] \arrow[dd, tail] & & D' & \\
& A \arrow[rr, tail] \arrow[dl, tail] & & B \arrow[dl, tail] \\
C \arrow[rr, tail] & & D \arrow[from=uu, tail, crossing over] &
\end{tikzcd}$$

Graph rewriting (which we will discuss in Section 3.2.2) is an instance of a generalised notion of rewriting defined categorically. This rewriting mechanism, that we will be using, is the well known notion of the *double-pushout rewriting*, which can be used in arbitrary adhesive categories [CM+97; LS05]. A *production* (or *rule*) is a span $L \leftarrow I \rightarrow R$ in a category $\mathbf{C}$.

Definition 2.19 (Double-pushout rewriting)**.** Let $p\colon L \leftarrow I \rightarrow R$ be a production in a category $\mathbf{C}$ and let $m\colon L \rightarrow X$ be a morphism (also called *match*) for an object X in $\mathbf{C}$. Then the object X rewrites to an object Y in $\mathbf{C}$ via production p (and match m), written $X \overset{p,m}{\Longrightarrow} Y$, if there exists a diagram consisting of morphisms (shown to the right) in which both squares are pushouts. The morphism n is called *co-match*.

$$\begin{tikzcd}
L \arrow[d, "m"'] & I \arrow[l] \arrow[r] \arrow[d] & R \arrow[d, "n"] \\
X & C \arrow[l] \arrow[r] & Y
\end{tikzcd}$$

Note that in the situation above there is not necessarily an object C, making the left-hand square a pushout. If the object C exists, we say that the *gluing condition* is satisfied. The gluing condition alongside the gluing construction for graph rewriting will be discussed in the next chapter.

If $\mathbf{C}$ is an adhesive category (and thus if it is a topos [LS06]) and the production consists of monos, then all remaining arrows of double-pushout diagrams of rewriting are monos [LS05] and the result of rewriting—be it the object Y or the co-match n—is unique (up to a canonical isomorphism).

This ends the summary of basic categorical definitions and brief explanations needed later in this thesis. The interested reader, who would like to learn more about the topic of category theory, is invited to have a look into *A Taste of Category Theory for Computer Scientists* by Benjamin C. Pierce [Pie88], which provides a good starting point for computer scientists who would like to explore the topic. Afterwards, to further consolidate the readers knowledge, one can consult the free available book *Abstract and Concrete Categories - The Joy of Cats* [AHS09] or the book *Categories for the Working Mathematician* by Saunders Mac Lane [Mac71]. Of course, there exist many more good introductions.

"As for everything else, so for a mathematical theory: beauty can be perceived but not explained."

Arthur Cayley (1821-1895)

3 Graphs and Graph Transformation

A substantial part of computer science is concerned with the transformation of structures, the most well-known example being the rewriting of words via Chomsky grammars, string rewriting systems [DJ90] or transformations of the tape of a Turing machine. The focus of this thesis is on systems where transformations are rule-based and rules consist of a left-hand side (the structure to be deleted) and a right-hand side (the structure to be added).

If we increase the complexity of the structures being rewritten, we next encounter trees or terms, leading to term rewriting systems (see also Chapter 6). The next level is concerned with graph rewriting [Roz97], which – as we will see below – differs from string and term rewriting in the sense that we need a notion of interface between left-hand and right-hand side, detailing how the right-hand side is to be glued to the remaining graph.

Graph rewriting is a flexible and intuitive, yet formally rigorous, framework for modelling and reasoning about dynamical structures and networks. Such dynamical structures arise in many contexts, be it object graphs and heaps, UML diagrams (in the context of model transformations [EE+15]), computer networks, the world wide web, distributed systems, etc. They also occur in other domains, where computer science methods are employed: social networks, as well as chemical and biological structures. Specifically concurrent non-sequential systems are well-suited for modelling via graph transformation, since non-overlapping occurrences of left-hand sides can be replaced in parallel. For a more extensive list of applications see [EE+99].

Note that in the context of this chapter we use the terms *graph rewriting* and *graph transformation* interchangeably. We will avoid the term *graph grammar*, since that emphasizes the use of graph transformation to generate a graph language, here the focus is just on the rewriting aspect. The specification of graph languages will be investigated in Chapter 4.

In the following, we first recall the basic notion of directed edge-labelled multi-graphs and graph morphisms. Afterwards, graph transformation systems and their correspondence to category theory will be explained in detail.

3.1. Graphs and Graph Morphisms

We start by defining graphs, where we choose to consider directed, edge-labelled graphs where parallel edges are allowed. Other choices would be to use hypergraphs (where an edge can be connected to any number of nodes) or to add node labels. Both versions can be easily treated by our rewriting approach. Throughout the thesis, we assume the existence of a fixed set Λ from which we take our edge labels.

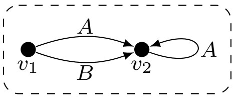
Definition 3.1 (Graph)**.** Let Λ be a fixed set of edge labels. A Λ-*labeled graph* is a tuple $G = (V, E, \mathit{src}, \mathit{tgt}, \mathit{lab})$, where V is a finite set of *nodes*, E is a finite set of *edges*, $\mathit{src}, \mathit{tgt} \colon E \to V$ assign to each edge a source and a target node, and $\mathit{lab} \colon E \to \Lambda$ is a labeling function.

Given a graph G, we denote its components by $V_G, E_G, \mathit{src}_G, \mathit{tgt}_G, \mathit{lab}_G$, unless otherwise indicated. Given an edge $e \in E_G$, the nodes $\mathit{src}_G(e), \mathit{tgt}_G(e)$ are called *incident* to e. The empty graph, i.e. a graph with $V = E = \emptyset$, is denoted $\varnothing$.

Example 3.2. *Let $V_G = \{v_1, v_2\}, E_G = \{e_1, e_2, e_3\}$ and $\Lambda = \{A, B\}$ be given. A graphical representation of the graph G with*

$$\begin{array}{lll} \mathit{src}_G(e_1) = v_1 & \mathit{src}_G(e_2) = v_1 & \mathit{src}_G(e_3) = v_2 \\ \mathit{tgt}_G(e_1) = v_2 & \mathit{tgt}_G(e_2) = v_2 & \mathit{tgt}_G(e_3) = v_2 \\ \mathit{lab}_G(e_1) = A & \mathit{lab}_G(e_2) = B & \mathit{lab}_G(e_3) = A \end{array}$$

$G =$



is depicted to the right.

A central notion in graph rewriting is a graph morphism. Just as a function is a mapping from a set to another set, a graph morphism is a mapping from a graph to a graph. It maps nodes to nodes and edges to edges, while preserving the structure of a graph. This means that if an edge is mapped to an edge, there must be a mapping between the source and target nodes of the two edges. Furthermore, labels must be preserved.

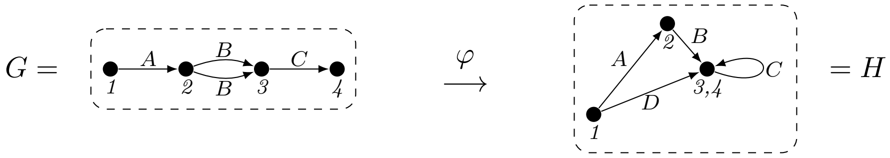
Definition 3.3 (Graph morphism)**.** Let G, H be two graphs. A *graph morphism* $\varphi \colon G \to H$ is a pair of mappings $\varphi_V \colon V_G \to V_H$, $\varphi_E \colon E_G \to E_H$ such that for all $e \in E_G$ it holds that

- $\mathit{src}_H(\varphi_E(e)) = \varphi_V(\mathit{src}_G(e))$,
- $\mathit{tgt}_H(\varphi_E(e)) = \varphi_V(\mathit{tgt}_G(e))$ *and*
- $\mathit{lab}_H(\varphi_E(e)) = \mathit{lab}_G(e)$.

A graph morphism φ is called *injective (surjective)* if both mappings φ_V, φ_E are injective (surjective). Whenever φ_V and φ_E are bijective, φ is called an *isomorphism*. If there exists an isomorphism $\varphi \colon G_1 \to G_2$, we say that G_1, G_2 are isomorphic and write $G_1 \cong G_2$. The composition of two graph morphisms is again a graph morphism. Graph morphisms are composed by composing both component mappings. *Composition of graph morphisms* is denoted by $\circ$.

In the following we omit the subscripts in the functions φ_V, φ_E and simply write φ. Furthermore, the negation of $G_1 \cong G_2$ will be denoted by $G_1 \ncong G_2$.

Example 3.4. *Consider the following graphs G and H. Note that the numbers written at the nodes are not part of the graph: they are just there to indicate the morphism from G to H.*



Here the edges of G are mapped with respect to their corresponding source and target node mappings. Note that the graph morphism φ is not surjective, since the D-labelled edge in H is not targeted. Furthermore, the morphism φ is not injective since the nodes 3 *and* 4 *of the graph G are mapped to the same node in H and the two B-labelled edges in G are mapped to the same edge in H.*

Graph morphisms are needed to identify the match of a left-hand side of a rule in a (potentially larger) host graph. As we will see next, they are also required for other purposes, such as graph gluing and graph transformation rules.

3.2. Graph Transformation Systems

Graph rewriting has been introduced in the early 1970's, where one of the seminal initial contributions was the paper by Ehrig, Pfender and Schneider [EPS73]. Since then, there have been countless articles in the field: many of them are foundational, describing and comparing different graph transformation approaches and working out the (categorical) semantics. Others are more algorithmic in nature, describing for instance methods for analysing and verifying graph transformation. Furthermore, as mentioned earlier at the beginning of this chapter, there have been a large number of contributions on applications, many of them in software engineering [EE+99], but in other areas as well, such as the recent growing interest from the area of biology in connection with the Kappa calculus [DF+12].

The aim of this subchapter is not to give a full overview over all possible approaches to graph transformation and all application scenarios. Instead, we give a condensed version that can be easily and concisely defined and explained. For basic graph rewriting, we rely on the *Double-Pushout Approach* (DPO) [CM+97; EPS73], which is one of the most well-known approaches to graph transformation, although clearly not the only one, and which will be explained in Section 3.2.2.

The central construction for the rewriting process, that we here call *graph gluing*, is an alternative way to describe a *pushout*. We will stick with the name graph gluing for now and in the definition we do not use the notion of pushouts, although we will afterwards explain the relation to the categorical notion.

3.2.1. Graph Rewriting via Graph Gluing

An intuitive explanation for the graph gluing construction is to think of two graphs G_1, G_2 with an overlap I. Now we glue G_1 and G_2 together over this common interface I, obtaining a new graph $G_1 +_I G_2$. This intuition is adequate in the case where the embeddings of I into the two graphs (called φ_1, φ_2 below) are injective, but not entirely when they are not. In this case one can observe some kind of merging effect that is illustrated in the examples below.

Graph gluing is described via factoring through an equivalence relation.

Definition 3.5 (Graph gluing)**.** Let I, G_1, G_2 be graphs with graph morphisms $\varphi_1 \colon I \to G_1$, $\varphi_2 \colon I \to G_2$, where I is called the *interface*. We assume that all node and edge sets are disjoint.

Let $\equiv$ be the smallest equivalence relation on $V_{G_1} \cup E_{G_1} \cup V_{G_2} \cup E_{G_2}$ which satisfies $\varphi_1(x) \equiv \varphi_2(x)$ for all $x \in V_I \cup E_I$.

The *gluing* of G_1, G_2 over I (written as $G = G_1 +_{\varphi_1,\varphi_2} G_2$, or $G = G_1 +_I G_2$ if the φ_i morphisms are clear from the context) is a graph G with:

$$V_G = (V_{G_1} \cup V_{G_2})/\equiv \qquad E_G = (E_{G_1} \cup E_{G_2})/\equiv$$

$$src_G([e]_\equiv) = \begin{cases} [src_{G_1}(e)]_\equiv & \text{if } e \in E_{G_1} \\ [src_{G_2}(e)]_\equiv & \text{if } e \in E_{G_2} \end{cases}$$

$$tgt_G([e]_\equiv) = \begin{cases} [tgt_{G_1}(e)]_\equiv & \text{if } e \in E_{G_1} \\ [tgt_{G_2}(e)]_\equiv & \text{if } e \in E_{G_2} \end{cases}$$

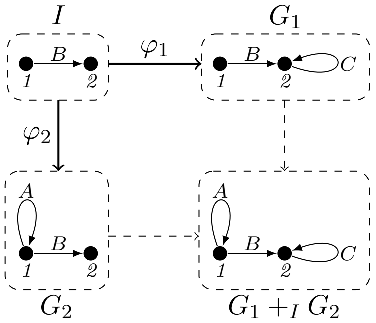
$$lab_G([e]_\equiv) = \begin{cases} lab_{G_1}(e) & \text{if } e \in E_{G_1} \\ lab_{G_2}(e) & \text{if } e \in E_{G_2} \end{cases}$$

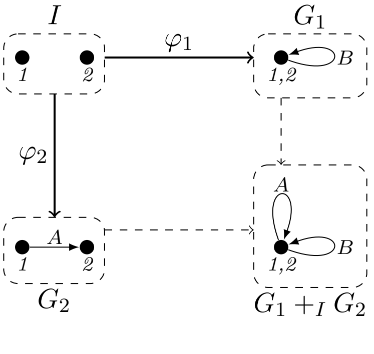
where $e \in E_{G_1} \cup E_{G_2}$.

Note that the gluing is well-defined, which is not immediately obvious since the mappings src_G, tgt_G, lab_G are defined on representatives of equivalence classes. The underlying reason for this is that φ_1, φ_2 are morphisms.

Example 3.6. *We now explain this gluing construction via some examples.*

Let the two graph morphisms $\varphi_1 \colon I \to G_1$ and $\varphi_2 \colon I \to G_2$ to the right be given, where both φ_1 and φ_2 are injective. Since the interface I is present in both graphs G_1 and G_2, we can glue the two graphs together to construct a graph $G_1 +_I G_2$ depicted on the bottom right of the square.





Now let the graph morphisms $\varphi_1 \colon I \to G_1$ and $\varphi_2 \colon I \to G_2$ to the left be given, where only φ_2 is injective. In the graph G_1, the interface nodes of I are merged via φ_1. The gluing graph $G_1 +_I G_2$ is constructed by merging all nodes in G_1, G_2, resulting in an A-labelled loop, together with the original B-labelled loop. This graph is depicted at the bottom right of the square.

We are now ready to define graph transformation rules, also called productions. Such a rule consists of a left-hand side graph L and a right-hand side graph R.

However, as indicated in the introduction, this is not enough. The problem is that, if we simply remove (a match of) L from a host graph, we would typically have dangling edges, i.e., edges where either the source or the target node (or both) have been deleted. Furthermore, there would be no way to specify how the right-hand side R should be attached to the remaining graph.

Hence, there is also an interface graph I related to L and R via graph morphisms, which specify what is preserved by a rule.

Definition 3.7 (Graph transformation rule)**.** A *(graph transformation) rule* ρ consists of three graphs L, I, R and two graph morphisms $L \xleftarrow{\varphi_L} I \xrightarrow{\varphi_R} R$.

Given a rule ρ, all nodes and edges in L that are not in the image of φ_L are called *obsolete*. Similarly, all nodes and edges in R that are not in the image of φ_R are called *fresh*.

After finding an occurrence of a left-hand side L in a host graph (a so-called match), the effect of applying a rule is to remove all obsolete elements and add all fresh elements. As indicated above, the elements of I are preserved, providing us with well-defined attachment points for R.

While this explanation is valid for injective matches and rule morphisms, it does not tell the full story in case of non-injective morphisms. Here, rules might split or merge graph elements. Using the graph gluing defined earlier, it is easy to give a formal semantics of rewriting.

The intuition is as follows: given a rule as in Definition 3.7 and a graph G, we ask whether G can be seen as a gluing of L and an (unknown) context C over interface I, i.e., whether there exists C such that $G \cong L +_I C$. If this is the case, G can be transformed into $H \cong R +_I C$.

Definition 3.8 (Graph transformation)**.** Let $\rho = (L \xleftarrow{\varphi_L} I \xrightarrow{\varphi_R} R)$ be a rule. We say that a graph G is transformed via ρ into a graph H (symbolically: $G \Rightarrow_\rho H$) if there is a graph C (the so-called *context*) and a graph morphism $\psi\colon I \to C$ such that:

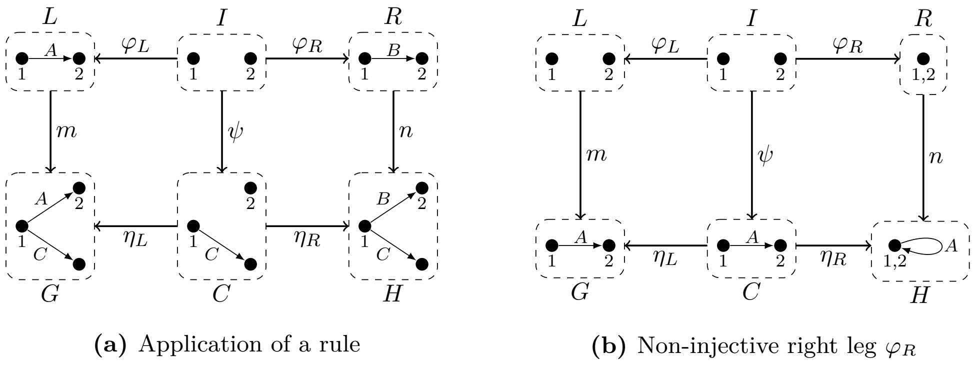
$$G \cong L +_{\varphi_L,\psi} C \qquad H \cong R +_{\varphi_R,\psi} C$$

This situation can be depicted by the diagram to the right. The morphism m is called the *match*, n the *co-match.*

$$\begin{tikzcd}
L \arrow[d, "m"'] & I \arrow[l, "\varphi_L"'] \arrow[r, "\varphi_R"] \arrow[d, "\psi"] & R \arrow[d, "n"] \\
G & C \arrow[l, "\eta_L"] \arrow[r, "\eta_R"'] & H
\end{tikzcd}$$

Depending on the morphisms φ_L and φ_R one can obtain different effects: whenever both φ_L and φ_R are injective, we obtain standard replacement. Whenever φ_L is non-injective we specify splitting, whereas a non-injective φ_R results in merging.

We now consider some examples. First, we illustrate the straightforward case where indeed the obsolete items are removed and the fresh ones are added, see Figure 3.1a. Somewhat more elaborate is the case when the right leg φ_R of a rule is non-injective, which causes the merging of nodes, see Figure 3.1b.



(a) Application of a rule

(b) Non-injective right leg φ_R

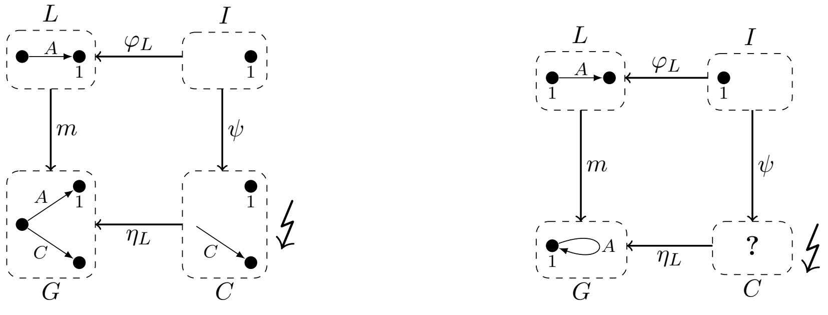
Figure 3.1.: Graph transformation rule examples

Different from string or term rewriting, in graph rewriting it may happen that we find a match of the left-hand side, but the rule is not applicable, because no context as required by Definition 3.8 exists. There are basically two reasons for this: either the rule removes a node, without removing all edges connected to that node (dangling edge condition, see Figure 3.2a), or the match identifies two graph elements which are not preserved (identification condition) (see Figure 3.2b).

Fact 3.9 (Gluing condition [Ehr79])**.** Let $L \xleftarrow{\varphi_L} I \xrightarrow{\varphi_R} R$ be a graph transformation rule and let $m\colon L \to G$ be a match. Then a context C and a morphism $\psi\colon I \to C$ such that $G \cong L +_{\varphi_L,\psi} C$ exist if and only if the following holds:

- *Dangling edge condition*: Every node $v \in V_L$, whose image $m(v)$ is incident to an edge $e \in E_G$ which is not in the image of m, is not obsolete (i.e. in the image of φ_L).
- *Identification condition*: Whenever two elements $x, y \in V_L \cup E_L$ with $x \neq y$ satisfy $m(x) = m(y)$, then neither of them is obsolete.

However, even if the context exists, there might be cases where it is non-unique.

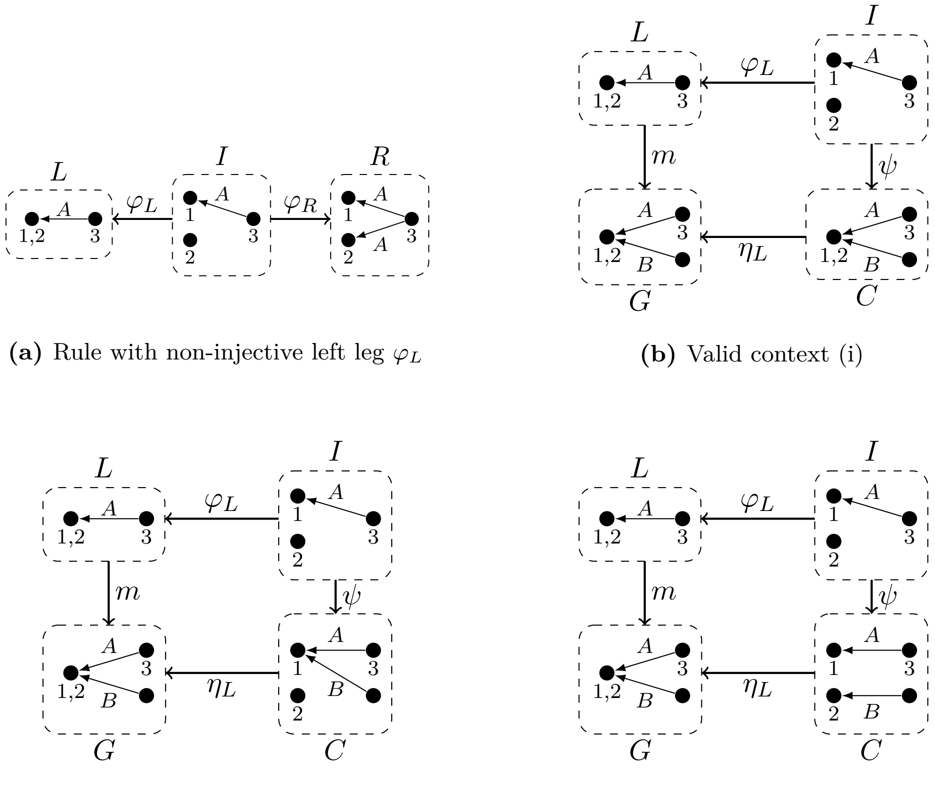


(a) Dangling edge condition example

(b) Identification condition example

Figure 3.2.: Gluing condition examples

This happens in cases where φ_L, the left leg of a rule, is non-injective. In this case one can for instance split nodes (see the rule in Figure 3.3a) and the question is what happens to the incident edges. By spelling out the definition above, one determines that this must result in non-determinism. Either, we do not split (Figure 3.3b) or we split and each edge can non-deterministically "choose" to stay either with the first or the second node (Figures 3.3c–3.3d). Each resulting combination is a valid context and this means that a rule application may be non-deterministic and generate several (non-isomorphic) graphs. In many papers such complications are avoided by requiring the injectivity of φ_L.



(a) Rule with non-injective left leg φ_L

(b) Valid context (i)

(c) Valid context (ii)

(d) Valid context (iii)

Figure 3.3.: Non-injective left leg rule with three valid contexts

Finally, we can introduce the notion of a graph transformation system that is used in this thesis.

Definition 3.10 (Graph transformation system)**.** A *graph transformation system* $\mathcal{R}$ is a set of *graph transformation rules.*

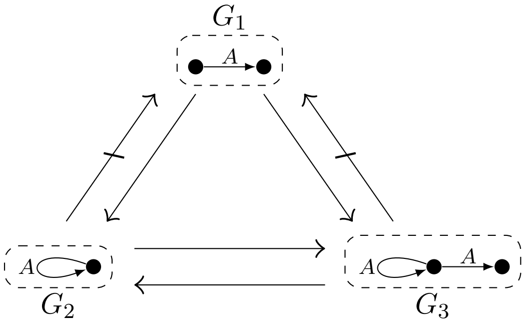
In the next subsection, we relate the two notions of graph gluing and pushouts. Afterwards we introduce one of the most popular formalisations of graph transformation systems, namely the *Double-Pushout Approach.*

3.2.2. Graph Transformation the Categorical Way

A considerable part of graph transformation theory is concerned with making the results independent of the specific graph structure under consideration (see [LS05; Löw93]). This however depends on the use of category theory. Therefore, we will consider the category $\mathbf{Graph}_\Lambda$ having Λ-labeled graphs as objects and graph morphisms as arrows. The set of its objects will be denoted by $|\mathbf{Graph}_\Lambda|$. The categorical structure induces an obvious preorder on graphs, defined as follows.

Definition 3.11 (Homomorphism preorder)**.** Given graphs G and H, we write $G \to H$ if there exists a graph morphism from G to H in $\mathbf{Graph}_\Lambda$. The relation $\to$ is a preorder (i.e. it is reflexive and transitive) and we call it the *homomorphism preorder* on graphs. We write $G \not\to H$ if $G \to H$ does not hold. Graphs G and H are *homomorphically equivalent*, written $G \sim H$, if both $G \to H$ and $H \to G$ hold.

Example 3.12. *Consider the following three graphs G_1, G_2 and G_3:*



As evident from the picture we get $G_2 \sim G_3$, and both $G_1 \to G_2, G_1 \to G_3$ but $G_2 \not\to G_1, G_3 \not\to G_1$. Furthermore it holds that $G_i \to G_i$ for all $i \in \{1, 2, 3\}$. Note that, the fusion of two nodes (while turning in-between binary edges into loops) and the graph expansion in form of adding additional graph structures, lead to a graph which is larger (or equal) in the homomorphism preorder, i.e. these steps preserve the existence of a morphism between the old and the modified graph.

We will revisit the homomorphism preorder in Chapter 7 when we investigate the notion of *core graphs*. There, we show that two graphs can only be homomorphically equivalent if they share a common subgraph structure.

As described in Section 2.2, category theory relies on so-called universal properties where, given some objects, one defines another object that is in some relation to the given object and is – in some sense – the most general object which is in this relation. In the case of graphs, the order relation is given by the homomorphism preorder $\to$, i.e., graph morphisms. Graph gluing can alternatively be characterized via the categorical notion of pushout (cf. Definition 2.14).

Fact 3.13 (Pushouts and the gluing construction [EE+06])**.** Let I, G_1, G_2 be graphs with graph morphisms $\varphi_1 \colon I \to G_1$, $\varphi_2 \colon I \to G_2$ as in Definition 3.5. Then the equation $G \cong G_1 +_{\varphi_1,\varphi_2} G_2$ holds if and only if G is a pushout.

Intuitively, the pushout characterization says that G should be a graph where the "common" parts of G_1, G_2 must be merged (since the square commutes), but it should be obtained in the most general way by merging only what is absolutely necessary and adding nothing superfluous. This corresponds to saying that for any other merge H, G is more general and H can be obtained from G by further merging or addition of graph elements (expressed by a morphism from G to H).

Please note that in $\mathbf{Graph}_\Lambda$, pushouts can be constructed componentwise for nodes and edges in the category **Set**. The source and target functions of the pushout graph are then uniquely determined by the pushout property of the node set in the pushout graph.

Graph transformation systems are widely used for modeling the evolution of parallel and distributed systems where the states are represented by graphs, and the behaviour is modeled by the application of local transformation rules to the state graph. One of the most popular formalisations of graph transformation systems is the *Double-Pushout Approach (DPO)* [Ehr79], where a transformation rule is represented as a pair of graph morphisms with the same source, and the application of a rule to a given graph is modeled with a construction involving two pushouts in the category of graphs. Therefore, our double-pushout rewriting is an instance of Definition 2.19 (also compare with Definition 3.8) in the category $\mathbf{Graph}_\Lambda$. We do not impose any injectivity constraint on the rule morphisms or to the match, hence graph transformation is non-deterministic.

The specifics such as the dangling edge condition in Proposition 3.9 are typical to the double-pushout approach that we are following here, i.e., it is forbidden to remove a node that is still attached to an edge, which is not deleted and in this case, the rule is not applicable. In other approaches, such as the single-pushout (or SPO) approach [Löw93] however, the deletion of a node to which an undeleted edge is attached, is possible. In this case all incident edges are deleted as well. In contrast to DPO, SPO is based on partial graph morphisms. This is also called *deletion in unknown contexts.*

For more information on the topic of graph transformation a standard reference is the "Handbook of Graph Grammars and Computing by Graph Transformation", which appeared in three volumes (foundations [Roz97] – applications, languages and tools [EE+99] – concurrency, parallelism and distribution [EK+99]). Strongly related to our definitions is the chapter on DPO rewriting by Corradini et al. [CM+97], which is based on the categorical definitions. The well-known book "Fundamentals of Algebraic Graph Transformation" by Ehrig et al. [EE+06] revisits the theory of graph rewriting from the point of view of adhesive categories, a general categorical framework for abstract transformations. In an introductory section it defines the construction of pushouts via factorization, equivalent to our notion of graph gluing.

"Coffee is a language in itself."

Jackie Chan (1954-present)

4 Type Graph Languages

In this chapter we study a specification formalism based on type graphs, where a type graph T represents all graphs that can be mapped homomorphically to T, potentially taking into account some extra constraints. Type graphs are common in graph rewriting [CMR96; Roz97]. Usually, one assumes that all items, i.e., rules and graphs to be rewritten, are typed, introducing constraints on the applicability of rules. Hence, type graphs are in a way seen as a form of labelling. This is different from our point of view, where graphs (and rules) are – a priori – untyped (but labeled) and type graphs are simply a means to represent sets of graphs.

There are various reasons for studying languages based on type graphs: First, they are reasonably simple with many positive decidability results and they have not yet been extensively studied from the perspective of specification formalisms. Second, other specification mechanisms – especially those used in connection with verification and abstract graph transformation [Ren04a; SRW02; SWW11] – are based on type graphs: abstract graphs are basically type graphs with extra annotations. Third, while not being as expressive as recognizable graph languages, they retain a nice intuition from regular languages: given a finite state automaton M one can think of the language of M as the set of all string graphs that can be mapped homomorphically to M (respecting initial and final states).

4.1. Type Graphs and Graph Languages

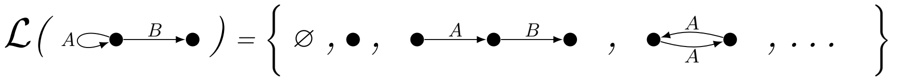
A *type graph language* contains all graphs that can be mapped homomorphically to a given *type graph*.

Definition 4.1 (Type graph language)**.** Let T be a Λ-labelled graph. Its *type graph language* $\mathcal{L}(T)$ is defined as:

$$\mathcal{L}(T) = \{G \mid G \to T\}.$$

Even if the graphs used for specifying type graph languages are just ordinary graphs, we will in the following call them *type graphs* in order to emphasize their role.

Example 4.2. *The following type graph T over the edge label set $\Lambda = \{A, B\}$ specifies a type graph language $\mathcal{L}(T)$ consisting of infinitely many graphs where no graph in the language contains a B-loop and every target node of a B-edge is not incident to an A-edge or the source of another B-edge.*



Specifying graph languages using type graphs gives us the possibility to forbid certain graph structures by not including them into the type graph. For example, no graph in the language of Example 4.2 can contain a B-loop or an A-edge incident to the target of a B-edge. However, it is not possible to force some structures to exist in all graphs of the language, since the morphism to the type graph need not be surjective. This point will be addressed with the notion of *annotated type graph* in Chapter 8.

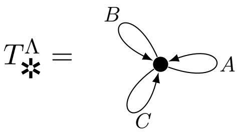
4.2. Examples

We have a look at some special cases, to convey a better understanding for the specification of graph languages via our notion of type graphs. We start with some type graphs which play an important role in the category $\mathbf{Graph}_\Lambda$, namely the corresponding initial and final object.

Example 4.3. *The category $\mathbf{Graph}_\Lambda$ has an initial object containing no nodes and edges, i.e. it is the empty graph $\varnothing$. For any non-empty graph object G in $\mathbf{Graph}_\Lambda$ we get $G \nrightarrow \varnothing$, i.e. if $G \neq \varnothing$ it implies $G \notin \mathcal{L}(\varnothing)$. However, in case of $G = \varnothing$, there exists a unique morphisms $\varphi\colon \varnothing \to \varnothing$ and therefore $\mathcal{L}(\varnothing) = \{\varnothing\}$. Please note, since the empty graph is the initial object in $\mathbf{Graph}_\Lambda$ it immediately follows that for any type graph T there exists a unique morphisms $\varphi\colon \varnothing \to T$, i.e. for all T it follows that $\varnothing \in \mathcal{L}(T)$. Due to this fact, it is impossible to specify an empty graph language in the framework of pure type graphs. Especially $T \in \mathcal{L}(T)$ always holds since $T \cong T$ guarantees us the existence of an isomorphism.*

While the initial object specifies a type graph language consisting only of one graph (the empty graph itself), its dual notion in form of the terminal object can be used to specify the language consisting of all graphs.

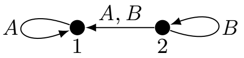
Example 4.4. *The category $\mathbf{Graph}_\Lambda$ has a final object, which we denote by $T^{\Lambda}_{✻}$, consisting of one node (called* flower node *✻) and one loop for each label in Λ. For any graph object G in $\mathbf{Graph}_\Lambda$ there exists a unique morphism $\varphi\colon G \to T^{\Lambda}_{✻}$ which maps all nodes in V_G to the flower node ✻ and all edges to the corresponding loop with the same label. Therefore $\mathcal{L}(T^{\Lambda}_{✻}) = |\mathbf{Graph}_\Lambda|$. The graph $T^{\Lambda}_{✻}$ for $\Lambda = \{A, B, C\}$ is depicted to the right.*



We will use the flower node ✻ extensively in Chapter 5 where we want to prove uniform termination of a graph transformation system, i.e. the absence of any infinite derivation sequence independent of the starting graph. For this purpose our type graphs will always contain a flower node ✻ to be able to specify the set of all possible starting graphs.

Another interesting class of graph languages are type graph languages which can be described in a dual notion by specifying that a certain subgraph structure is not contained in any graph of the language.

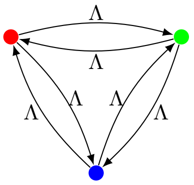
Example 4.5. *Consider the following type graph T depicted to the below right. This type graph plays a special role as it specifies a graph language that could alternatively be described by the following property: $\mathcal{L}(T)$ contains all graphs G over the edge label set $\Lambda = \{A, B\}$ that do not contain such a node, which is both, the target of an A-labelled edge and the source of a B-labelled edge at the same time. In fact, any graph $G \in \mathcal{L}(T)$ can be mapped to the type graph T by using the following function $\varphi_V \colon V_G \to V_T$ for each $v \in V_G$:*

$T =$ 

$$\varphi_V(v) = \begin{cases} 1, & \textit{if } v \textit{ is the target of an } A\textit{-labeled edge} \\ 2, & \textit{if } v \textit{ is the source of a } B\textit{-labeled edge} \\ 2, & \textit{otherwise} \end{cases}$$

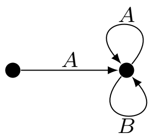
Type graph languages which can be described in a dual notion are specified by a type graph T, for which there exists a partner graph R, such that the pair of R and T forms a so-called *duality pair*. We will consider these pairs and further explain the special role of the second graph R (also called *restriction graph*) in Chapter 7.

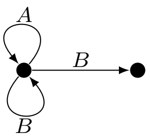
Example 4.6. *Consider the following type graph T depicted to the below right. This type graph plays a special role for complexity reasoning. The type graph language $\mathcal{L}(T)$ consists of all 3-colorable graphs, a problem which is known to be NP-complete. Every Λ-labeled edge represents a set of edges, one for each label in Λ. The color of each type graph node indicates one of the three colors for the coloring of the graph node that is mapped into it.*

$T =$ 

Finally, as a last example we show that different type graphs can specify the same graph language.

Example 4.7. *Let the following two type graphs T_1 and T_2 over $\Lambda = \{A, B\}$ be given.*

$T_1 =$ 

$T_2 =$ 

Due to the fact that both type graphs T_1 and T_2 consist of a flower node ✲ over the label set Λ, we can deduce that $\mathcal{L}(T_1) = \mathcal{L}(T_2)$. Please note, that T_1 and T_2 are not isomorphic ($T_1 \not\cong T_2$) but they are homomorphically equivalent ($T_1 \sim T_2$).

The last example gives rise to the question, if there exists a unique minimal representation alongside homomorphically equivalent type graphs which specifies the same language. This idea is similar to asking for a minimal deterministic finite automata in the field of regular languages. The answer to this question alongside an investigation of decidability and closure properties for type graph languages will be discussed in Chapter 7.

Part II.

Termination Analysis of Graph Transformation Systems

Motivation of Part II

The question of termination is one of the most fundamental problems studied in every computational formalism, for instance the halting problem for Turing machines. For graph transformation systems there has been some work on termination, but this problem has received less attention than, e.g., confluence or reachability analysis. There are several applications where termination analysis is essential: one scenario is termination of graph programs, especially for programs operating on complex data structures. Furthermore, model transformations, for instance of UML models, usually require functional behaviour, i.e., every source model should be translated into a unique target model. This requires termination and confluence of the model transformation rules.

There is a huge body of termination results in string and term rewriting [Ter03] from which one can draw inspiration. Still, adapting these techniques to graph transformation is often non-trivial. A helpful first step is often to modify these techniques to work with cycle rewriting [ZBK14; SZ15], which imagines the two ends of a string to be glued together, so that rewriting is performed on a cycle.

In [BKZ14] it was shown how to adapt methods from string rewriting [Zan95; KW08] which resulted in a technique based on weighted type graphs, that was implemented in the tool Grez. Despite its simplicity the method is quite powerful and finds termination arguments also in cases which are difficult for humans to comprehend. However, there are some examples (see for instance the example discussed in Section 5.4) where this technique fails. The corresponding techniques in string rewriting can be seen as matrix interpretations of strings in certain semirings, more specifically in the tropical and arctic semiring. Those semirings can be replaced by the arithmetic semiring (the natural numbers with addition and multiplication) in order to obtain a powerful termination analysis method for string rewriting [HW06; EWZ08].

Here we generalize this method to graphs. Due to their non-linear nature, we have to abandon matrices and instead state a different termination criterion that is based on weights of morphisms of the left-hand and right-hand sides of rules into a type graph.

By introducing weighted type graphs we generalize the matrix-based interpretations for string rewriting in two ways: first, we transform graphs instead of strings and second, we consider general semirings. Our techniques work for so-called strictly and strongly ordered semirings, which have to be treated in a slightly different way.

For termination issues an extensive database of benchmarks has been developed, the so-called *Termination Problems Database* (short: TPDB). The TPDB consists of a great number of term rewriting systems which can be used to benchmark new termination proof concepts and is often used in workshops aimed at termination competitions. Therefore, for testing our weighted type graph approach, it is natural to filter a suitable selection from this database and try to convert them into graph transformation systems while preserving termination proving arguments.

We investigate how to interpret a (left-linear non-collapsing) term rewrite rule as a term graph production. We argue that there are two natural ways to do so. One is called the *basic version* and the other one is called the *extended version*, coinciding with the version as studied in [CD11].

We then observe that conceptually there is a strong relationship between term graph rewriting and graph transformation systems. We propose two transformations from term graph productions to graph transformation rules, namely the function encoding and the number encoding. For both we prove soundness, for the latter also completeness under mild conditions.

Last, for a selection of 201 term rewriting systems, we apply Grez on four variants: both the function encoding and the number encoding on both the basic and the extended version of interpreting term rewriting systems on term graphs.

Outline:
After recalling the basic theory in Section 5.1 we introduce termination analysis based on weighted type graphs in Section 5.2 and Section 5.3. We will discuss an extended example in Section 5.4, followed by a presentation of the implementation in the termination tool Grez in Section 5.5. In Section 6.1 we recall terms and term graph rewriting. In Section 6.2 we present and discuss the basic and extended version to interpret term rewriting rules as term graph rewriting productions. In Section 6.3 we present the function encoding and the number encoding to transform term graph rewriting to graph transformation, and prove the relevant properties. Finally, we present our experiments on the term rewriting systems from the TPDB in Section 6.4. All proofs can be found in Appendix A.1 and A.2.

"It is the weight, not numbers of experiments that is to be regarded."

Isaac Newton (1642-1727)

5 Weighted Type Graphs over Semirings

In this chapter we introduce techniques for proving uniform termination of graph transformation systems, based on matrix interpretations for string rewriting. We generalize this technique by adapting it to graph rewriting instead of string rewriting and by generalizing to ordered semirings. In this way we obtain a framework which includes three variants of type graphs; namely, the tropical type graphs, the arctic type graphs and the arithmetic type graphs. These weighted type graphs can be used to assign weights to graphs and to show that these weights decrease in every rewriting step in order to prove termination.

5.1. Additional Preliminaries - Termination and Semirings

The weighted type graph technique is strongly influenced by matrix interpretations for proving termination in string, cycle and term rewriting systems [HW06; SZ15; EWZ08]. We will generalize this technique, resulting in a technique for graph transformation systems that has a distinctly different flavour than the original method. In order to point out the differences later and motivate the choices, we will recall the basic concept of proving termination for a rewriting system and introduce matrix interpretations for string rewrite systems first. Afterwards, we introduce ordered semirings which are used to assign weights to the rule morphisms of a graph transformation system.

5.1.1. Termination Analysis of Rewriting Systems

Termination, i.e. the absence of infinite computations, is a desirable property that is required in many applications. Algorithms that manipulate data structures or model transformation are just two applications where termination of the computation process plays an important role. Both of these example applications can be modelled using graph transformation systems in a natural way. The termination property for graph transformation systems is defined as follows.

Definition 5.1 (Termination and uniform termination)**.** A graph transformation system is *terminating* if it does not allow infinite transformation sequences from a fixed set of initial graphs and it is called *uniformly terminating* if it is terminating for all graphs.

With respect to graph transformation systems, all variants of the termination problem, termination on all graphs as well as termination on a fixed set of initial graphs, are undecidable [Plu98]. This fact follows quite directly from the halting problem. Nonetheless, it is important to develop semi-decision procedures which are able to decide as many instances as possible of the termination problem. One basic approach to prove termination of a rewriting system, is to assign the rewritable objects to elements of a well-founded set.

Definition 5.2 (Termination analysis)**.** Let $\mathcal{R}$ be a set of rewriting rules. Then $\mathcal{R}$ is terminating if and only if there exists an evaluation function $|\cdot|$ and a well-founded order $\leq$ such that for any object A which can be rewritten to another object B (short $A \Rightarrow_{\mathcal{R}} B$) it holds that $|A| > |B|$.

Since the rewritten object's value decreases with every rule application, we can deduce the uniform termination property from the fact that the order is well-founded. Therefore, there does not exist an infinity, strictly decreasing sequence and the objects assigned to the lowest values can not be rewritten any further.

5.1.2. Matrix Interpretations for String Rewriting

We are working in the context of *string rewrite systems*, where a rule is of the form $\ell \to r$, where ℓ, r are both strings over a given alphabet Σ. For instance, consider the alphabet $\Sigma = \{a, b\}$ and a string rewriting rule $aa \to aba$. Starting from an example word aaa we can do the following derivation steps:

$$\begin{aligned} &\underline{aa}a \Rightarrow ab\underline{aa} \Rightarrow ababa \quad \text{or} \\ &a\underline{aa} \Rightarrow \underline{aa}ba \Rightarrow ababa \end{aligned}$$

However, in both derivations the word $ababa$ can not be rewritten anymore ($ababa \not\Rightarrow$). Therefore, the rewriting rule $aa \to aba$ is terminating for the initial word aaa. We would like to know if the rewriting rule (or a rewriting system consisting of several such rules) is terminating independent of a given initial word, i.e., uniformly terminating.

In termination analysis of string rewriting, strings are assigned to elements in a well-founded set and it has to be shown that each rule application leads to a decrease with respect to this order (see Definition 5.2). In [HW06] the well-founded order is based on square matrices over the natural numbers $\mathbb{N}_0$. The order is defined as follows:

Let A, B be two square matrices A, B over $\mathbb{N}_0$ of equal dimension n. We write $A > B$ if $A_{1,1} > B_{1,1}$ and $A_{i,j} \geq B_{i,j}$ for all indices i, j with $1 \leq i, j \leq n$, i.e., we require that the entries in the upper left corner are strictly ordered, whereas the remaining entries may also be equal. It holds that $A > B$ implies $A \cdot C > B \cdot C$ and $C \cdot A > C \cdot B$ for a matrix[1] $C > \mathbf{0}$ of appropriate dimension.

Every letter of the alphabet $a \in \Sigma$ is associated with a square matrix $A = [a] > \mathbf{0}$ (where all matrices have the same dimension n). Similarly every word $w = a_1 \dots a_n$ is mapped to a matrix $[w] = [a_1] \cdot \ldots \cdot [a_n]$, which is obtained by taking the matrices of the single letters and multiplying them. If we can show $[\ell] > [r]$ for every rule

[1] Here $\mathbf{0}$ denotes the matrix with all entries zero.

$\ell \to r$, then termination is implied by the considerations above and by the fact that the order $\leq$ on $\mathbb{N}_0$ is well-founded.

Example 5.3. *For the rule $aa \to aba$ take the following matrices (as in [HW06]):*

$$[a] = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \qquad [b] = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \qquad \textit{with}$$

$$[aa] = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix} \qquad \textit{and}$$

$$[aba] = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$$

Since we found matrices $[a]$ and $[b]$ such that $[aa] > [aba]$ holds, the rewriting rule $aa \to aba$ is uniformly terminating.

For *cycle rewriting* a similar argument can be given, which is based on the idea that the trace, i.e., the sum of the diagonal, of a matrix decreases [SZ15].

A natural question to ask is how such matrices can be obtained. We will discuss in Section 5.5 how SMT solvers can be employed to automatically generate the required weights.

In the following, we will generalize this method in two ways: we will replace the natural numbers by an arbitrary semiring – an observation that has already been made in the context of string rewriting – and we will make the step from string to graph rewriting.

5.1.3. Ordered Semirings

We continue by defining semirings, the algebraic structures in which we will evaluate the graphs occurring in transformation sequences, and orders on them. We are interested in two types of ordered semirings: strongly ordered semirings and strictly ordered semirings.

Definition 5.4 (Semiring)**.** A *semiring* is a 5-tuple $(S, \oplus, \otimes, 0, 1)$, where S is the (finite or infinite) carrier set, $(S, \oplus, 0)$ is a commutative monoid, $(S, \otimes, 1)$ is a monoid, $\otimes$ distributes over $\oplus$ and 0 is an annihilator for $\otimes$. That is, the following laws hold for all $x, y, z \in S$:

$$\begin{array}{lll}
(x \oplus y) \oplus z = x \oplus (y \oplus z) & 0 \oplus x = x & x \otimes 0 = 0 \\
(x \otimes y) \otimes z = x \otimes (y \otimes z) & x \oplus 0 = x & 0 \otimes x = 0 \\
(x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z) & 1 \otimes x = x & x \oplus y = y \oplus x \\
z \otimes (x \oplus y) = (z \otimes x) \oplus (z \otimes y) & x \otimes 1 = x &
\end{array}$$

A semiring $\langle S, \oplus, \otimes, 0, 1 \rangle$ is *commutative* if $\otimes$ is commutative.

We will often confuse a semiring with its carrier set, that is, S can refer to both the semiring $\langle S, \oplus, \otimes, 0, 1 \rangle$ itself and its carrier set. In order to come up with termination arguments, we need a partial order on the semirings that has to be compatible with its operations.

Definition 5.5 (Ordered semiring)**.** A structure $(S, \oplus, \otimes, 0, 1, \leq)$ is an *ordered semiring* if $(S, \oplus, \otimes, 0, 1)$ is a semiring and $\leq$ is a partial order on S such that for all $x, y, u, z \in S$:

- $x \leq y$ implies $x \oplus z \leq y \oplus z$, $x \otimes z \leq y \otimes z$ and $z \otimes x \leq z \otimes y$ for $z \geq 0$.

The ordered semiring S is *strongly ordered*, if

- $x < y$, $z < u$ implies $x \oplus z < y \oplus u$; and
- $z > 0$, $x < y$ implies $x \otimes z < y \otimes z$ and $z \otimes x < z \otimes y$.

The strongly ordered semiring S is *strictly ordered*, if in addition $x < y$ implies $x \oplus z < y \oplus z$ for all $z \in S$.

We will denote by Σ ($\prod$) the generalised sum of $\oplus$ (generalised product of $\otimes$).

Example 5.6. *We consider examples of semirings used in termination proving:*

Strictly ordered semirings:

- *The natural numbers form a semiring* $(\mathbb{N}_0, +, \cdot, 0, 1, \leq)$*, where* $\leq$ *is the standard ordering of the natural numbers. We will call this semiring the arithmetic semiring (on the natural numbers). This is a strictly ordered semiring because both* $<$ *and* $\leq$ *are monotone in* $+$ *and* $\cdot$

Strongly ordered semirings:

- *The tropical semiring (on the natural numbers) is:*

$$T_{\mathbb{N}_0} = (\mathbb{N}_0 \cup \{\infty\}, \min, +, \infty, 0, \leq),$$

where $\leq$ *is the usual ordering of the natural numbers. The tropical semiring is not strictly ordered, because, for example,* $2 < 3$ *but* $\min(1, 2) \not< \min(1, 3)$*. It is however still strongly ordered.*

- *The arctic semiring (on the natural numbers) is*

$$T_{\mathbb{N}_0} = (\mathbb{N}_0 \cup \{-\infty\}, \max, +, -\infty, 0, \leq),$$

where $\leq$ *is the usual ordering of the natural numbers. Like the tropical semiring, the arctic semiring is not strictly ordered, but strongly ordered.*

All semirings above are commutative.

We will in the following restrict ourselves to commutative semirings, since we are assigning weights to graphs by multiplying weights of nodes and edges, and nodes and edges are typically unordered. Furthermore, in the following chapters, we focus exclusively on uniform termination, i.e., there is only a set of graph transformation rules, but no fixed initial graph, and the question is whether the rules terminate on *all* graphs. To abstract the set of all weighted graphs we will extend the concept of type graph languages from Chapter 4.

5.2. Weighted Type Graphs

Similar to mapping a word to a matrix, we will associate weights to graphs, by typing them over a type graph with weights from a semiring.

Definition 5.7 (Weighted type graph)**.** Let an ordered semiring S be given. A *weighted type graph* T over S is a graph with a weight function $w_T \colon E_T \to S$ and a designated flower node $✻_T \in V_T$, such that for each label $A \in \Lambda$ there exists a designated edge e_A with $src_T(e_A) = ✻_T$, $tgt_T(e_A) = ✻_T$, $lab_T(e_A) = A$ and $w_T(e_A) > 0$.

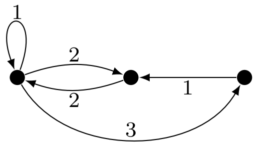
For a graph G, we denote with $fl_T(G)$ (or just $fl(G)$ if T is clear from the context) the unique morphism from G to T that maps each node $v \in V_G$ of G to the flower node $✻_T$ and each edge $e \in E_G$, with $lab_T(e) = A$, to e_A. Note that, for a morphism $c \colon G \to H$, it is always the case that $fl_T(H) \circ c = fl_T(G)$.

Note that every matrix A of dimension n can be associated with an (unlabelled) type graph with nodes $1 \dots n$, where an edge from node i to j is assigned weight $A_{i,j}$ (or does not exist if $A_{i,j} = 0$). Hence our idea of weighted type graphs is strongly related with the matrices of Section 5.1.2.

Example 5.8. *The matrix A shown below left specifies the weighted type graph T shown below right. In our approach, only matrix elements with a weight greater than 0 induce an edge in the weighted type graph T.*

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

$T =$



The node $✻_T$ is also called the *flower node*, since the loops attached to it look like a flower. Those loops correspond to the matrix entries at position $(1,1)$ and similar to those entries they play a specific role. Note that the flower structure also ensures that *every* graph can be typed over T (compare with the terminal object in the category $\mathbf{Graph}_\Lambda$ in Example 4.4, which is exactly such a flower).

Based on the weighted type graphs we are now ready to define how the weight of a graph $G \in \mathcal{L}(T)$ can be computed. To do so, we first assign weights to the typing morphisms. With a bit of notation overloading, we assign a weight to each morphism $t \colon G \to T$ with codomain T and arbitrary domain G as follows.

Definition 5.9 (Weight of a morphism)**.** Let T be a weighted type graph and let $t \colon G \to T$ be a graph morphism. Then the weight of t is obtained by multiplying the weights of the edges in the image of t, i.e.,

$$w_T(t) = \prod_{e \in E_G} w_T(t(e)).$$

That is, we multiply the weights of all edges in the image of t with respect to the $\otimes$-operator of the underlying semiring.

Finally, the weight of a graph G with respect to T is defined by summing up the weights of all morphisms from G to T with respect to $\oplus$.

Definition 5.10 (Weight of a graph)**.** Let G be a graph. The weight of G is the sum of the weights of the morphisms from G into the weighted type graph T, i.e.,

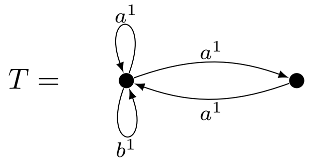
$$w_T(G) = \sum_{t_G\colon G \to T} w_T(t_G).$$

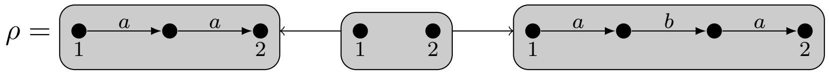
That is, we sum up the weights of all morphisms from G to T with respect to the $\oplus$-operator of the underlying semiring.

The subscript T of w_T will be omitted if it is clear from the context.

Example 5.11. *We give a small example for the weight of a graph.*

Consider for instance the type graph T to the right. Edges are labelled a, b and the weights, in this case natural numbers, are given as superscripts. Consider also the left-hand side L of rule ρ below, consisting of two a-edges (the graph rewriting analogue of the string rewriting rule $aa \to aba$ considered in Section 5.1.2).





There are five morphisms $t_i\colon L \to T$ with $0 \leq i \leq 4$, each having weight 1, as they are calculated by multiplying the weights of two a-edges which also have weight 1.

- *$t_0 = fl(L)$ is the flower morphism and maps all nodes to the left node of T. In this situation we have $w_T(t_0) = 1 \cdot 1 = 1$.*
- *t_1 is the morphism that maps the first interface node and the middle node to the left node of T and the second interface node to the right node of T. In this case we have $w_T(t_1) = 1 \cdot 1 = 1$.*
- *t_2 is the morphism that maps the first interface node to the right node of T and the second interface node together with the middle node to the left node of T. In this case we have $w_T(t_2) = 1 \cdot 1 = 1$.*
- *t_3 is the morphism that maps the first and second interface node to the right node of T and the middle node to the left node of T. In this case we have $w_T(t_4) = 1 \cdot 1 = 1$.*
- *t_4 is the morphism that maps the first and second interface node to the left node of T and the middle node to the right node of T. In this case we have $w_T(t_3) = 1 \cdot 1 = 1$.*

Hence the weight of L with respect to T is

$$\begin{aligned} w_T(L) &= w_T(t_0) + w_T(t_1) + w_T(t_2) + w_T(t_3) + w_T(t_4) \\ &= 1 + 1 + 1 + 1 + 1 \\ &= 5 \end{aligned}$$

More details on the usage of these weights are given in Example 5.15.

If we glue two graphs G_1, G_2 in order to obtain G, the weight of G can be obtained from the weights of G_1, G_2.

Lemma 5.12. Let S be an ordered commutative semiring and T a weighted type graph over S.

(i) Whenever S is strongly ordered, for all graphs G, $\mathit{fl}_T(G)\colon G \to T$ exists and $w_T(\mathit{fl}_T(G)) > 0$.

(ii) Given the following diagram, where the square is a pushout and G_0 is discrete, it holds that $w_T(t) = w_T(t \circ \varphi_1) \otimes w_T(t \circ \varphi_2)$.

$$\begin{tikzcd}
& G_1 \arrow[dr, "\varphi_1"] & & \\
G_0 \arrow[ur, "\psi_1"] \arrow[dr, "\psi_2"'] & (\mathrm{PO}) & G \arrow[r, "t"] & T \\
& G_2 \arrow[ur, "\varphi_2"'] & &
\end{tikzcd}$$

Since property (ii) above only holds if G_0 is discrete we restrict to discrete graphs I in the rule interface.[2]

While the process of obtaining the weight of a graph corresponds to calculating the matrix of a word and summing up all its entries, we also require a way to be more discriminating, i.e., to access separate matrix entries. Evaluating a string-like graph would mean to fix its entry and exit node within the type graph (similarly to fixing two matrix indices). However, in graph rewriting, we have interfaces of arbitrary size. Hence, we do not index over pairs of nodes, but over arbitrary interface graphs, and compute the weight of a graph L with respect to a typed interface I.

Definition 5.13 (Weight of a morphism wrt. an interface morphism)**.** Let $\varphi\colon I \to L$ and $t\colon I \to T$ be graph morphisms, where T is a weighted type graph. We define:

$$w_t(\varphi) = \sum_{\substack{t_L\colon L \to T \\ t_L \circ \varphi = t}} w_T(t_L).$$

$$\begin{tikzcd}
L \arrow[dr, dashed, "t_L"'] & I \arrow[l, "\varphi"'] \arrow[d, "t"] \\
& T
\end{tikzcd}$$

Finally, we can define what it means that a rule is decreasing, analogous to the condition $[\ell] > [r]$ introduced in Section 5.1.2. In addition we also introduce non-increasingness, a concept that will be needed for the so-called relative termination arguments.

Definition 5.14 (Non-increasing and decreasing rules)**.** Let a graph transformation rule $\rho = L \xleftarrow{\varphi_L} I \xrightarrow{\varphi_R} R$, an ordered commutative semiring S and a weighted type graph T over S be given.

(i) The rule ρ is *non-increasing* if for all $t_I\colon I \to T$ it holds that $w_{t_I}(\varphi_L) \geq w_{t_I}(\varphi_R)$.

(ii) The rule ρ is *decreasing* if it is non-increasing, and $w_{\mathit{fl}(I)}(\varphi_L) > w_{\mathit{fl}(I)}(\varphi_R)$.

Example 5.15. *We come back to Example 5.11 and check whether rule ρ is decreasing. For this we have to consider the following four interface morphisms $t\colon I \to T$ from the two-node interface into the weighted type graph T:*

[2] Compare also with the “stable under pushouts” property of [BKZ14].

- *The flower morphism $\mathit{fl}(I)$ which maps both interface nodes to the left node of T. In this case we have $w_{\mathit{fl}(I)}(\varphi_L) = w_T(t_0) + w_T(t_4) = 2 > 1 = w_{\mathit{fl}(I)}(\varphi_R)$.*
- *Furthermore there are three other interface morphisms $t_1, t_2, t_3 \colon I \to T$ mapping the two interface nodes either both to the right node of T, or the first interface node to the left and the second interface node to the right node of T, or vice versa. In all these cases we have $w_{t_i}(\varphi_L) = w_T(t_i) = 1 = w_{t_i}(\varphi_R)$.*

Hence, the rule is decreasing. Note also that these weights correspond exactly to the weights of the multiplied matrices in Example 5.3.

Finally, we have to show that applying a decreasing rule also decreases the overall weight of a graph. For a non-increasing rule the weight might remain the same.

Lemma 5.16. Let S be a strictly ordered commutative semiring and T a weighted type graph over S. Furthermore, let ρ be a rule such that $G \Rightarrow_\rho H$.

(i) If ρ is non-increasing, then $w_T(G) \geq w_T(H)$.

(ii) If ρ is decreasing, then $w_T(G) > w_T(H)$.

From this lemma we can prove our main theorem that is based on the well-known concept of *relative termination* [Ges90; Zan03]: if we can find a type graph for which some rules are decreasing and the rest is non-increasing, we can remove the decreasing rules without affecting termination. We are then left with a smaller set of rules for which termination can either be shown with a different type graph or with some other technique entirely.

Theorem 5.17 (Relative termination based on strictly ordered semirings)**.** Let S be a strictly ordered commutative semiring with a well-founded order $\leq$ and T a weighted type graph over S. Let R be a set of graph transformation rules, partitioned in two sets $R^<$ and $R^=$. If all rules of $R^<$ are decreasing and all rules of $R^=$ are non-increasing then R is terminating if and only if $R^=$ is terminating.

A special case of the theorem is when $R^= = \emptyset$. Then the statement of the theorem is that a graph transformation system R is terminating if all its rules are decreasing with respect to a strictly ordered commutative semiring S and type graph T over S.

5.3. Using Strongly Ordered Semirings

In the last section the semirings were required to be strictly ordered. In this section we consider what happens when we weaken this requirement and also allow non-strictly ordered semirings, which must however be strongly ordered. This allows us to work with the tropical and arctic semiring from Example 5.6. It turns out that we obtain similar results to Theorem 5.17 if we strengthen the notion of decreasingness.

Definition 5.18 (Strongly decreasing rules)**.** Let a rule $\rho = L \xleftarrow{\varphi_L} I \xrightarrow{\varphi_R} R$, an ordered commutative semiring S and a weighted type graph T over S be given. The rule ρ is *strongly decreasing* (with respect to T) if for all $t_I \colon I \to T$ it holds that $w_{t_I}(\varphi_L) > w_{t_I}(\varphi_R)$.

Using this notion of decreasingness we can also formulate a termination argument, which is basically equivalent to the termination argument presented in [BKZ14].

Lemma 5.19. Let S be a strongly ordered commutative semiring and T a weighted type graph over S. Furthermore, let ρ be a rule such that $G \Rightarrow_\rho H$.

(i) If ρ is non-increasing, then $w_T(G) \geq w_T(H)$.

(ii) If ρ is strongly decreasing, then $w_T(G) > w_T(H)$.

Now it is easy to prove a theorem analogous to Theorem 5.17, using Lemma 5.19 instead of Lemma 5.16.

Theorem 5.20 (Relative termination based on strongly ordered semirings)**.** Let S be a strongly ordered commutative semiring with a well-founded order $\leq$ and T a weighted type graph over S. Let R be a set of graph transformation rules, partitioned in two sets $R^<$ and $R^=$. If all rules of $R^<$ are strongly decreasing and all rules of $R^=$ are non-increasing, then R is terminating if and only if $R^=$ is terminating.

In this way we have recovered the termination analysis from [BKZ14], however spelt out differently. In order to explain the connection, let us consider what it means for a rule $\rho = L \xleftarrow{\varphi_L} I \xrightarrow{\varphi_R} R$ to be non-increasing in the tropical semiring where $\oplus$ is min and $\otimes$ is $+$ i.e., for each $t \colon I \to T$ into a weighted type graph T it must hold that

$$\min_{\substack{t_L \colon L \to T \\ t_L \circ \varphi_L = t}} w_T(t_L) \geq \min_{\substack{t_R \colon R \to T \\ t_R \circ \varphi_R = t}} w_T(t_R)$$

$w_T(t_L)$ is the weight of the morphism t_L, obtained by summing up (via $+$) the weights of all edges in the image of t_L.

A different way of expressing that the minimum of the first set is larger or equal than the minimum of the second set, is to say that for each morphism $t_L \colon L \to T$ with $t_L \circ \varphi_L = t$ there exists a morphism $t_R \colon R \to T$ with $t_R \circ \varphi_R = t$ and $w_T(t_L) \geq w_T(t_R)$.

Similarly, a rule $\rho = L \xleftarrow{\varphi_L} I \xrightarrow{\varphi_R} R$ is non-increasing in the arctic semiring where $\oplus$ is max and $\otimes$ is $+$ whenever for each $t \colon I \to T$ into a weighted type graph T it holds that

$$\max_{\substack{t_L \colon L \to T \\ t_L \circ \varphi_L = t}} w_T(t_L) \geq \max_{\substack{t_R \colon R \to T \\ t_R \circ \varphi_R = t}} w_T(t_R)$$

where again, $w_T(t_L)$ is the weight of the morphism t_L, obtained by summing up (via $+$) the weights of all edges in the image of t_L.

A different way of expressing that the maximum of the first set is larger or equal than the maximum of the second set, is to say that for each morphism $t_R\colon R \to T$ with $t_R \circ \varphi_R = t$ there exists a morphism $t_L\colon L \to T$ with $t_L \circ \varphi_L = t$ and $w_T(t_L) \geq w_T(t_R)$. These are exactly the notions of tropically and arctically non-increasing of [BKZ14].

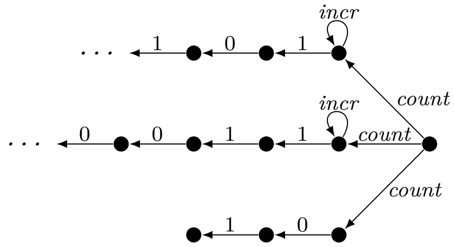
In hindsight, comparing the results of Theorems 5.17 and 5.20 we notice the following: as underlying semiring S we can take either a strictly ordered or a strongly ordered one, but if we choose a strongly ordered semiring, the termination argument becomes slightly harder to prove because for every morphism from the left-hand side to the type graph there must exist a compatible, strictly smaller morphism from the right-hand side to the type graph.

5.4. Examples

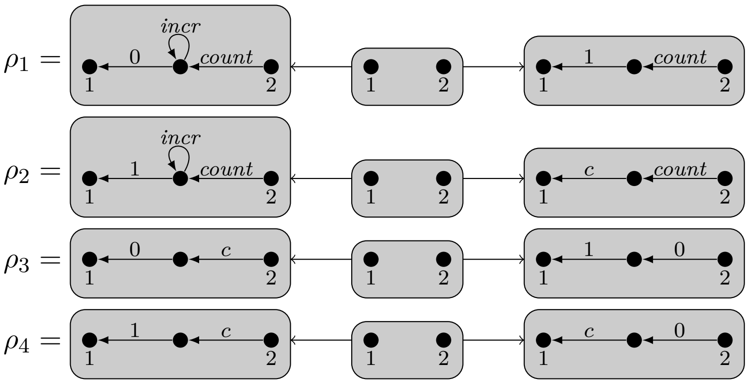
We give examples to show that with a weighted type graph over a strictly ordered semiring (such as the arithmetic semiring), we can prove termination on some graph transformation systems where strongly ordered semirings fail. We start with a graph transformation system for which a termination argument can be found using both variants. Then we will modify some rules and explain why weighted type graphs over strongly ordered semirings can not find a termination argument for the modified system.

Example 5.21. *As an example we take a system consisting of several counters, which represent their current value by a finite number of bits. Each counter may possess an* incr *marker, that can be consumed to increment the counter by* 1. *One possible graph describing a state of such a system is given by* G. *This is just one possible initial graph, since we really show uniform termination, i.e., termination on all initial graphs, even those that do not conform to the schema indicated by* G.

$G =$

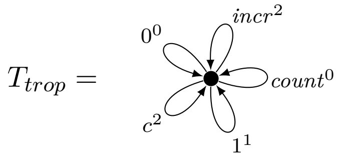


We consider the graph transformation system $\{\rho_1, \rho_2, \rho_3, \rho_4\}$, *adapted from [SZ15], consisting of the following four rules:*



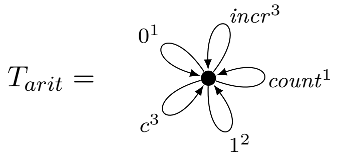
Each counter may increment at most once. Rules ρ_1 and ρ_2 specify that a counter (represented by a count*-labelled edge) may increment its least significant bit by* 1 *if an* incr *marker was not consumed yet. If the least significant bit is* 1*, the bit is marked by a label* c*, to remember that a carry bit has to be passed to the following bit. Rule ρ_3 increments the next bit of the counter by* 1 *(if it was* 0 *before), while rule ρ_4 shifts the carry bit marker over the next* 1*.*

The fact that this graph transformation system is uniformly terminating can be shown using a weighted type graph over either a strictly or strongly ordered semiring. For example, using a non-relative termination argument, we evaluate the rules with respect to the weighted type graph T_{trop} over the tropical semiring.



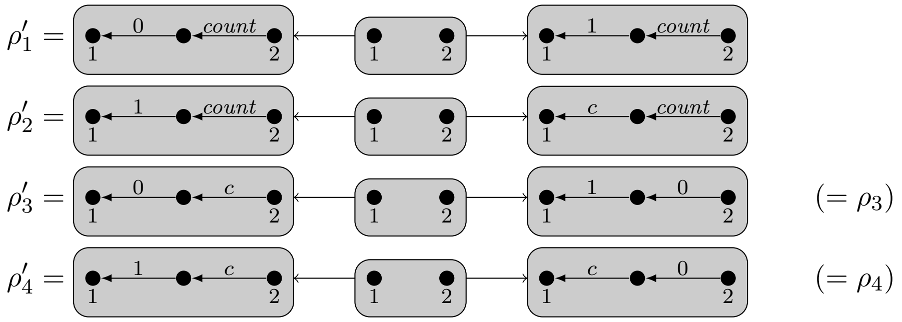
A relative termination argument is even easier: the rules ρ_1 and ρ_2 can be removed due to the decreasing number of incr*-labelled edges. Then we can remove ρ_3 due to the decreasing number of* c*-labelled edges (which remain constant in ρ_4) and afterwards remove ρ_4 since it decreases* 1*-labelled edges. With all rules removed, the graph transformation system has been shown to terminate uniformly.*

We now consider the arithmetic semiring and again use a non-relative termination argument: we evaluate the rules with respect to the weighted type graph T_{arit}, where all weights are just increased by one with respect to T_{trop}. That is due to the fact, that we are working in the arithmetic semiring and hence have to make sure that all weights of flower edges are strictly larger than 0*.*



Example 5.22. *We will now modify rules ρ_1 and ρ_2 in order to give an example where weighted type graphs over tropical and arctic semirings fail to find a termination argument.*

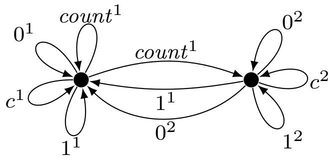
Consider the graph transformation system $\{\rho'_1, \rho'_2, \rho'_3, \rho'_4\}$ consisting of rules ρ_3 and ρ_4 from Example 5.21 with two additional new rules:



With respect to Example 5.21, the counter may increment its value not only once but several times, until the least significant bit is permanently marked by the carrier bit label c. This will eventually happen, since counters are never extended by additional digits and carry bits finally accumulate and can not be processed.

We now give a relative termination argument, to show uniform termination of this graph transformation system. The termination of this system is not obvious as the numbers of the labels c, 0 *and* 1 *increase and decrease depending on the* *rules used for the derivation. First, we evaluate the rules with respect to the following weighted type graph T' over the arithmetic semiring. Consider for instance rule ρ'_1 and the following four interface morphisms:*

$T' =$



- $t_0 = fl(I)\colon I \to T'$ *is the flower morphism and maps both interface node to the left node of* T'*. In this situation we have* $w_{t_0}(\varphi_L) = 1 \cdot 1 + 1 \cdot 2 = 3 > 2 = 1 \cdot 1 + 1 \cdot 1 = w_{t_0}(\varphi_R)$ *(there are two ways to map the left-hand side in such a way that both interface nodes are mapped to the left node, resulting in weight* 3*; similar for the right-hand side, where we obtain weight* 2*).*

- $t_1\colon I \to T'$ *is the morphism that maps the first interface node to the right node of* T' *and the second interface node to the left node of* T'*. In this case we have* $w_{t_1}(\varphi_L) = 1 \cdot 2 = 2 \geq 2 = 1 \cdot 2 = w_{t_1}(\varphi_R)$*.*

- $t_2\colon I \to T'$ *is the morphism that maps the first interface node to the left node of* T' *and the second interface node to the right node of* T'*. In this case we have* $w_{t_2}(\varphi_L) = 0 \geq 0 = w_{t_2}(\varphi_R)$*, since there are no possibilities to map either the left-hand or the right-hand side.*

- $t_3\colon I \to T'$ *is the morphisms that maps both interface node to the right node of* T'*. Here we have* $w_{t_3}(\varphi_L) = 0 \geq 0 = w_{t_3}(\varphi_R)$ *(again, there are no fitting matches of the left-hand and right-hand side).*

Hence ρ'_1 *is decreasing. Similarly we can prove that* ρ'_2 *is decreasing and* ρ'_3, ρ'_4 *are non-increasing, which means that* ρ'_1, ρ'_2 *can be removed. To show termination of the remaining rules* ρ'_3, ρ'_4 *we can simply use the weighted type graph* T_{arit} *from Example 5.21 again.*

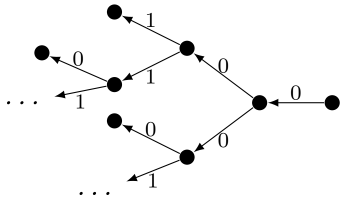
We found a relative termination argument for Example 5.22 using a weighted type graph over the arithmetic semiring. However, there is no way to obtain a termination argument with a weighted type graph over either tropical or arctic semirings: in these cases the weight of any graph is linear in the size of the graph (since we use only addition and minimum/maximum to determine the weight of a graph). If we have an interpretation where at least one rule is decreasing, and the other rules are non-increasing, then in any derivation, the number of applications of the decreasing rules is at most linear in the size of the initial graph.

However, if we start with a counter which consists of n *bits (all set to* 0*), we obtain a derivation in which* all *of the rules are applied at least* 2^n *times. This means that it is principally impossible to find a proof with weighted type graphs over the tropical or arctic semiring, even using relative termination.*

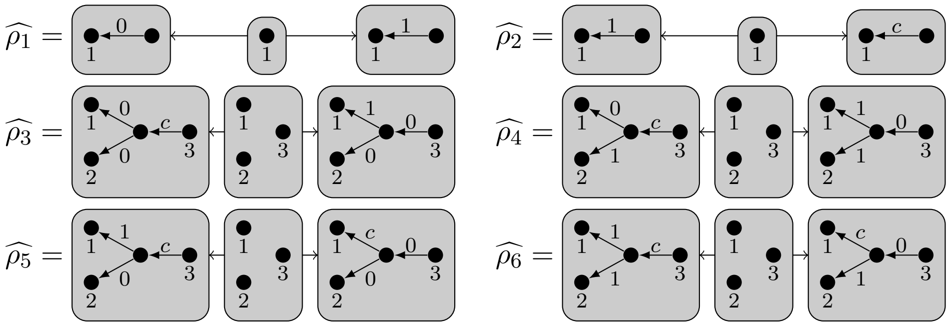
The last two examples were inspired by string rewriting and the example rules could easily be encoded into a string grammar. We give another final example and prove termination using a weighted type graph over the arithmetic semiring. We now switch from strings to trees, staying with a scenario where reductions of exponential length are possible. In addition we discard the *count*-label as each counter will be represented by a node with no incoming edge and we will exploit the dangling edge condition.

Example 5.23. *We interweave our counters into a single treelike structure. Each path from a root node to a leaf can be interpreted as a counter.*
One possible graph describing a state of the modified system is given by $\widehat{G}$. Each counter shares a number of bits with other counters, where the least significant bit is shared by all counters. Again this is just one possible initial graph, since we prove uniform termination.

$\widehat{G} =$

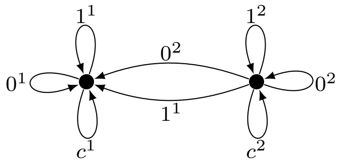


Let the following graph transformation system $\{\widehat{\rho_1}, \widehat{\rho_2}, \widehat{\rho_3}, \widehat{\rho_4}, \widehat{\rho_5}, \widehat{\rho_6}\}$ be given:



The rules $\widehat{\rho_1}$ and $\widehat{\rho_2}$ increment the shared least significant bit by 1*. These two rules can only be applied at the root of the tree (due to the dangling edge condition of the DPO approach), as long as the edge is either labelled* 0 *or* 1*. By applying the rules $\widehat{\rho_3}, \ldots, \widehat{\rho_6}$, a carrier bit can be passed to the next bit. Proving termination of this graph transformation system is non-trivial. By applying for instance $\widehat{\rho_6}$, the value of the counters containing interface node* 1 *does not change, while the other counter values decrease. We evaluate the rules with respect to the following weighted type graph $\widehat{T}$ over the arithmetic semiring. We can prove that $\widehat{\rho_1}$ and $\widehat{\rho_2}$ are decreasing and $\widehat{\rho_3}, \ldots, \widehat{\rho_6}$ are non-increasing, which means that $\widehat{\rho_1}$, $\widehat{\rho_2}$ can be removed using a relative termination argument.*

$\widehat{T} =$



The rules $\widehat{\rho_3}$ and $\widehat{\rho_4}$ can be removed due to the decreasing number of c-labelled edges, which remain constant in $\widehat{\rho_5}$ and $\widehat{\rho_6}$. Afterwards we can remove $\widehat{\rho_5}$, $\widehat{\rho_6}$ since they decrease the number of 1-labelled edges. The graph transformation system has been shown to terminate uniformly, since there are no rules left.

5.5. Grez

The erstwhile question of how to find suitable weighted type graphs has been left open so far. Instead of manually searching for a suitable type graph we employ a *satisfiable modulo theories* (SMT) solver (in this case Z3) that can solve inequations over the natural numbers.

We fix a number n of nodes in the type graph and proceed as follows: take a complete graph T with n nodes, i.e., a graph with an edge for every pair $i, j \in \{1, \ldots, n\}$ of nodes and every edge label $a \in \Lambda$. Every edge e in this graph is associated with a variable x_e. The task is to assign weights to those variables such that rules can be shown as either decreasing or non-increasing.

Now, for every rule $\rho = L \xleftarrow{\varphi_L} I \xrightarrow{\varphi_R} R$ and every map $t\colon I \to T$ we obtain an inequation:

$$\sum_{\substack{t_L\colon L \to T\\ t_L \circ \varphi_L = t}} \prod_{e \in E_L} x_{t_L(e)} \geq \sum_{\substack{t_R\colon R \to T\\ t_R \circ \varphi_R = t}} \prod_{e \in E_R} x_{t_R(e)}$$

If we want to show that ρ is decreasing and t is the flower morphism $\geq$ has to be replaced by $>$.

Doing this for each rule and every map t gives us equations that can be used as input for an SMT-solver. We consider the weights as natural numbers only up to a given bound by restricting the length of the corresponding bit-vectors. Note that we would be outside the decidable fragment of arithmetics otherwise since the equations would contain multiplication of variables (as opposed to multiplication of constants and variables). By using a bit-vector encoding the SMT-solver Z3 can reliably find a solution (if it exists) and especially such solutions are found for the examples discussed in Section 5.4. Any solution gives us a valid weighted type graph.

A prototype Java-based tool, called Grez, has been written and was introduced in [BKZ14]. Given a graph transformation system $\mathcal{R}$, the tool tries to automatically find a proof for the uniform termination of $\mathcal{R}$. The tool supports relative termination and runs different algorithms (which are chosen by the user) concurrently to search a proof. If one algorithm succeeds in finding a termination argument for at least one of the rules, all processes are interrupted and the corresponding rule(s) will be removed from $\mathcal{R}$. The algorithms are then executed on the smaller set of rules and this procedure is repeated until all rules have been removed. Afterwards Grez generates the full proof which can be saved as a PDF-file.

Grez provides both a command-line interface and a graphical user interface. The tool supports the integration of external tools, such as other termination tools or SMT-solvers. Grez can use any SMT-solver which supports the SMT-LIB2 format [BST10]. Grez generates the inequation described above in this format and passes it, either through a temporary file or via direct output stream, to the SMT-solver. The results are parsed back into the termination proof, as soon as the SMT-solver terminates and produces a model for the formula.

We ran the tool on all examples of this chapter using a Windows workstation with a $2,67$ Ghz, 4-core CPU and 8 GB RAM. All proofs were generated in less than 1 second. The tool, a user manual [Bru15] and the examples from this chapter can be downloaded from the Grez webpage.[3]

[3] Grez homepage: www.ti.inf.uni-due.de/research/tools/grez

“Problems are often stated in vague terms... because it is quite uncertain what the problems really are.”

John von Neumann (1903-1957)

6 Terms, Term Rewriting and Term Graph Encodings

In the previous Chapter 5 we studied termination analysis inspired by matrix interpretations on string rewriting and cycle rewriting. In this chapter we will focus on term rewriting instead and describe ways to encode term rewriting systems into graph transformation systems while preserving termination proving arguments, i.e. termination of the term rewriting system can be concluded from the termination of the corresponding graph transformation system. Furthermore, we can benefit from an existing large database of term rewriting systems which we can use as case studies for our previously introduced weighted type graph technique.

We will introduce term graph rewriting as a natural way to apply term rewriting to graphs. Term graph rewriting provides an efficient way to implement term rewriting. We discuss two natural ways to interpret term rewrite rules as term graph rewrite productions: one in which implicit unraveling of the term graph is allowed and one in which it is not. We provide techniques to automatically prove termination of term graph rewriting and apply them to term rewriting systems in both ways. The main approach is to transform term graph rewriting to a graph transformation system and then apply the tool Grez from Section 5.5 for proving termination of the graph transformation system. We discuss two such transformations and report about experiments.

6.1. Additional Preliminaries - Terms and Term Graphs

When trying to express algorithms in specific domains (that have certain structures and/or properties), string rewriting systems are not suited very well for describing semantics in domains by means of homomorphisms. Like string rewriting, term rewriting can describe grammars, but it additionally offers mechanics to describe processes in functional programming or analyse logic programs [Mar94]. Therefore, term rewriting systems play an important role in various areas, such as abstract data type specifications. In order to understand the remainder of this chapter we introduce the basics of terms and term rewriting systems. A good introduction on terms and term rewriting is given in [DJ90] and a comprehensive overview of research in the area of term rewriting can be found in [Ter03].

6.1.1. Terms and Term Rewriting

In term rewriting the following three basic concepts are essential: terms, substitution and matching. We start by defining terms.

Definition 6.1 (Signature and terms)**.** Let $\mathcal{X}$ be a countable set of variables $x, y, z, \ldots$, let $\Box$ be a special symbol denoting an empty space and let $\mathcal{F}$ be a *signature*, i.e. a set of function symbols $\{f, g, \ldots\}$ each having a fixed arity given by a mapping $\mathsf{ar}\colon \mathcal{F} \to \mathbb{N}_0$.

The set of terms over the signature $\mathcal{F}$ is the least set $\mathcal{T}(\mathcal{F}, \mathcal{X})$ satisfying:

- If $x \in \mathcal{X}$, then $x \in \mathcal{T}(\mathcal{F}, \mathcal{X})$.
- If $a \in \mathcal{F}$ is a constant symbol (i.e., $\mathsf{ar}(a) = 0$), then $a \in \mathcal{T}(\mathcal{F}, \mathcal{X})$.
- If $f \in \mathcal{F}$ is a n-ary function symbol (i.e., $\mathsf{ar}(f) = n > 0$) and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, then $f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F}, \mathcal{X})$.

For a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ the set $\mathcal{V}\mathrm{ar}(t)$ contains all variables occurring in t.

- We say that a term t is *ground* if $\mathcal{V}\mathrm{ar}(t) = \emptyset$.
- The term t is called *linear* if it does not contains multiple occurrence of the same variable.

A *context* is a term containing one occurrence of $\Box$ and is denoted by $C[\,]$. If a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ is replacing $\Box$ the result is denoted by $C[t] \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and t is called a subterm of $C[t]$.

The function $\mathsf{root}\colon \mathcal{T}(\mathcal{F}, \mathcal{X}) \to \mathcal{F} \cup \mathcal{X}$ returns the symbol labeling the root of t, i.e., $\mathsf{root}(t) = t$ if $t \in \mathcal{X}$ and $\mathsf{root}(f(t_1, \ldots, t_n)) = f$ if $f \in \mathcal{F}$.

Example 6.2. *Let $\mathcal{X} = \{x, y\}$ and $\mathcal{F} = \{f, g, a\}$ with $\mathsf{ar}(f) = 2$, $\mathsf{ar}(g) = 1$ and $\mathsf{ar}(a) = 0$ be given. We give the following examples:*

- *The expression $f(f(a, g(a)))$ is not a term in $\mathcal{T}(\mathcal{F}, \mathcal{X})$ since the occurence of the outer function symbol f is used as if it had arity 1.*
- *$t = f(f(x, a), y)$ is a term ($t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$), which is linear but not ground and the root is $\mathsf{root}(t) = f$.*
- *The term $t' = f(x, a)$ is a subterm of t with the context $C[\,] = f(\Box, y)$. We have $t = C[t']$.*

In term rewriting, not only $\Box$ but also the variables of a term t, can be seen as abstract representations of subterms. These subterms do not have to be ground and could still consist of variables. Variables play an important role when it comes to matches between terms. In term rewriting, a match is a mapping between terms which preserves function symbols and identifies variables in the source term with subterms in the target term. The identification of a variable with a term is also called *substitution*.

Definition 6.3 (Substitution and homomorphic extension)**.** Let $\mathcal{T}(\mathcal{F},\mathcal{X})$ be a set of terms over the signature $\mathcal{F}$ with a set of variables $\mathcal{X}$. A *substitution* is a mapping $\sigma\colon \mathcal{X} \to \mathcal{T}(\mathcal{F},\mathcal{X})$. Substitutions are *homomorphically* extended to mappings from terms to terms $\bar{\sigma}\colon \mathcal{T}(\mathcal{F},\mathcal{X}) \to \mathcal{T}(\mathcal{F},\mathcal{X})$ by:

$$\bar{\sigma}(x) = \sigma(x) \quad \text{and} \quad \bar{\sigma}(f(t_1,\ldots,t_n)) = f(\bar{\sigma}(t_1),\ldots,\bar{\sigma}(t_n))$$

By using substitutions $\bar{\sigma}$, we can define matches between terms. Matches are used to identify an occurrence of a (possibly not ground) left-hand-side term ℓ, from a term rewriting rule $\ell \to r$, within a term t that is supposed to be rewritten.

Definition 6.4 (Match)**.** A term ℓ *matches* a term t if there exists a substitution $\bar{\sigma}\colon \mathcal{T}(\mathcal{F},\mathcal{X}) \to \mathcal{T}(\mathcal{F},\mathcal{X})$ such that $t = \bar{\sigma}(\ell)$. The substitution $\bar{\sigma}$ is also called *matcher* of t against ℓ.

Example 6.5. *Let $\mathcal{T}(\mathcal{F},\mathcal{X})$ be the set of terms from Example 6.2 and let the following three terms $\ell_1, \ell_2, t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ be given:*

$$\ell_1 = f(x, g(y)) \qquad \ell_2 = f(x, a) \qquad t = f(f(x,a), g(a))$$

Then the term ℓ_1 matches the term t by the substitution $\sigma(x) = f(x,a)$ and $\sigma(y) = a$ which leads to $t = \bar{\sigma}(\ell_1)$. The term ℓ_2 can not be matched with t due to the constant a (the second argument in $f(x,a)$) which can not be substituted to $g(a)$.

We are now ready to define term rewrite rules. Like in many rewrite systems these rules consist of two elements from the corresponding framework which are related by a rewriting relation.

Definition 6.6 (Term rewrite rule, redex and contractum)**.** A *term rewrite rule* is an ordered pair $\ell \to r$, where $\ell, r \in \mathcal{T}(\mathcal{F},\mathcal{X})$ are called the left-hand-side and right-hand-side respectively and the following two properties hold:

- $\ell \notin \mathcal{X}$ (the left hand side is not a variable), and
- $\mathcal{V}\mathrm{ar}(r) \subseteq \mathcal{V}\mathrm{ar}(\ell)$ (there is no extra variable in the right-hand-side)

A term rewrite rule $\ell \to r$ is called *left-linear* if ℓ is linear, *right-linear* if r is a linear and it is called *linear* if both ℓ and r are linear terms. The rule is called *collapsing* if the right-hand-side is a single variable ($r \in \mathcal{X}$).

Given a substitution $\bar{\sigma}$, an instance $\bar{\sigma}(\ell)$ of the left-hand-side ℓ of a term rewriting rule $\ell \to r$ is called a *redex* (reducible expression) of the rule and an instance $\bar{\sigma}(r)$ of the right-hand-side is called a *contractum*.

Example 6.7. *The term rewrite rule $f(x,x) \to f(g(x),a)$ is right-linear and non-collapsing. For the substitution $\sigma(x) = g(a)$ the ground term $\bar{\sigma}(\ell) = f(g(a),g(a))$ is a redex and the ground term $\bar{\sigma}(r) = f(g(g(a)),a)$ is a contractum.*

Finally, we define term rewriting systems. Term rewriting systems play an important role in various areas, for instance for the analysis and implementation of abstract data type specifications, i.e. checking the consistency of properties or used for theorem proving. Just like in other rewriting frameworks, a term rewriting system is a set of rewrite rules.

Definition 6.8 (Term rewriting system)**.** A *Term Rewriting System* is a pair $\mathcal{R} = (\mathcal{F}, R)$ such that $\mathcal{F}$ is a signature and R is a set of rewrite rules over the signature $\mathcal{F}$.

A term rewriting system $\mathcal{R} = (\mathcal{F}, R)$ is called left-linear, right-linear or linear if each rule $\ell \to r \in R$ has the corresponding property. On the other hand, $\mathcal{R}$ is called collapsing if at least one of the rules $\ell \to r \in R$ is collapsing.

In the following we will simply denote a term rewriting system $\mathcal{R}$ by its set of rewrite rules R whenever the signature $\mathcal{F}$ is clear from the context. A rewriting step is now based on replacing a redex by a corresponding contractum within the same context. Given a term rewriting system $\mathcal{R} = (\mathcal{F}, R)$ we define the (one-step) rewrite relation $\to_\mathcal{R}$ as the set of all possible rewriting steps over terms in $\mathcal{T}(\mathcal{F}, \mathcal{X})$.

Definition 6.9 (Rewriting relation)**.** Let a term rewriting system $\mathcal{R} = (\mathcal{F}, R)$ be given. A term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ can be rewritten to a term $u \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ (written $t \to_\mathcal{R} u$ and called a *rewriting step*) if there exists a term rewriting rule $\ell \to r \in R$ together with a substitution $\sigma \colon \mathcal{X} \to \mathcal{T}(\mathcal{F}, \mathcal{X})$ such that $t = C[\bar{\sigma}(\ell)]$ and $u = C[\bar{\sigma}(r)]$, i.e. a redex is replaced by a corresponding contractum within the same shared context.

Example 6.10. *Consider the following linear, non-collapsing term rewriting system $\mathcal{R} = (\mathcal{F}, R)$ which consists of the two rules $\rho_1, \rho_2 \in R$:*

$$\rho_1 = f(a, x, y) \to h(x, y) \qquad \text{and} \qquad \rho_2 = g(b, y) \to a$$

Let $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ be the term $t = f(g(b, a), h(z, z), b)$. Then the following rewriting steps are possible, where the matched subterm is underlined and the applied rule is written above the rewriting relation:

$$f(\underline{g(b, a)}, h(z, z), b) \quad \overset{\rho_2}{\to}_\mathcal{R} \quad \underline{f(a, h(z, z), b)} \quad \overset{\rho_1}{\to}_\mathcal{R} \quad h(h(z, z), b)$$

In implementing term rewriting, an obvious optimization is to share common subterms. In this way the objects to rewrite are not terms represented by trees, but by directed graphs. For finite terms the directed graphs are acyclic, but in many applications, in particular in functional programming ([PE93]), it makes sense to also allow cycles, by which after unfolding the represented term is infinite. These graphs are called *term graphs*, and rewriting on term graphs has been extensively studied, see e.g. [Plu99; CD11], sometimes under the name of jungle rewriting [HKP88; CR93].

6.1.2. Term Graph Rewriting

We will now focus on term graphs where every node is labeled by an operation symbol, and the outgoing edges of such a node are numbered from 1 to the arity of the operation symbol.

Definition 6.11 (Term graph)**.** A *term graph* over a signature $\mathcal{F}$ is a triple $(V, \mathsf{lb}, \mathsf{succ})$ in which

- V is a finite set of nodes (*vertices*),
- $\mathsf{lb} : V \to \mathcal{F}$ is a partial function, called *labeling*, and
- $\mathsf{succ} : V \to V^*$ is a partial function, called *successor*, having the same domain as lb, such that for every $v \in V$ for which $\mathsf{succ}(v)$ is defined, the length of $\mathsf{succ}(v)$ is equal to the arity of $\mathsf{lb}(v)$: $|\mathsf{succ}(v)| = \mathsf{ar}(\mathsf{lb}(v))$.

A *(term graph) morphism* $\varphi : (V, \mathsf{lb}, \mathsf{succ}) \to (V', \mathsf{lb}', \mathsf{succ}')$ is a function $\varphi : V \to V'$ such that for all $v \in V$ where $\mathsf{lb}(v)$ is defined, $\mathsf{lb}'(\varphi(v)) = \mathsf{lb}(v)$ and $\mathsf{succ}'(\varphi(v)) = \varphi(v_1) \cdots \varphi(v_n)$ with $\mathsf{succ}(v) = v_1 \cdots v_n$.

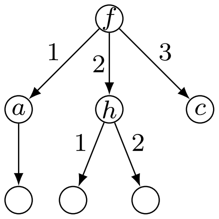
This definition coincides with the one given in [CD11]. If $\mathsf{succ}(v) = (v_1, \ldots, v_n)$ then we see $(v, v_1), \ldots, (v, v_n)$ as the n outgoing edges of v. Note that swapping two outgoing edges changes the term graph since edges are ordered. For specifying ground term graphs we are interested in the case where lb and succ are total, but for defining term graph rewriting it is convenient to allow also partial functions. Furthermore, in the various definitions of term graphs within the literature there are some syntactical variations, in particular whether or not a term graph is assumed to have a root node. In this chapter we will investigate term graphs that are not required to have a root. Term graphs are a direct extension of finite terms.

Definition 6.12 (Term graph of a term)**.** For a finite linear term t (that is, every variable occurs at most once in t) its term graph $\mathsf{TG}(t)$ is defined inductively together with $\mathsf{root}(t)$ as follows:

- For a constant a the term graph $\mathsf{TG}(a)$ consists of a single node v for which $\mathsf{succ}(v) = \epsilon$ and $\mathsf{lb}(v) = a$, and $\mathsf{root}(a) = v$.
- For a variable x the term graph $\mathsf{TG}(x)$ consists of a single node v for which $\mathsf{succ}(v)$ and $\mathsf{lb}(v)$ are both undefined, and $\mathsf{root}(x) = v$.
- For a term $t = f(t_1, \ldots, t_n)$ let V_i be the set of nodes of the term graph of t_i, and $\mathsf{root}(t_i) = v_i \in V_i$, for $i = 1, \ldots, n$. Then the set of nodes of the term graph $\mathsf{TG}(t)$ is defined to be the disjoint union of $V_1, \ldots, V_n$ and a fresh node v. For nodes in $V_1, \ldots, V_n$, the functions lb and succ are maintained, and $\mathsf{succ}(v) = v_1 \cdots v_n$, $\mathsf{lb}(v) = f$ and $\mathsf{root}(t) = v$.

For a finite non-linear term t its term graph $\mathsf{TG}(t)$ is defined in the same way, with the only difference that the occurrences of a variable are not taken disjoint, but shared.

Example 6.13. *We construct the term graph for the term $t = f(a(x), h(y, z), c)$. The corresponding term graph* $\mathsf{TG}(t)$ *is depicted on the right, where the numbers on outgoing edges from a node v indicate the order in* $\mathsf{succ}(v)$*.*



Conversely, term graphs can be *unraveled* to terms: starting from a node v, we build a term with $\mathsf{lb}(v)$ as its root, and for each of its $|\mathsf{succ}(v)|$ successors generate the corresponding argument by applying the same process on the corresponding element of $\mathsf{succ}(v)$. If the term graph is acyclic, this yields a finite term, but as soon as a node is visited that has been visited before, the resulting term will be infinite.

Just as in term rewriting, a term graph transformation rule consists of a left-hand side and a right-hand side, and the basic idea is that an occurrence of a left-hand side may be replaced by the corresponding right-hand side. Now left-hand sides and right-hand sides are term graphs themselves, and an occurrence of a left-hand side may be defined as an injective morphism from the left-hand side to the term graph to be rewritten. However, for a precise description some extra information is required: which nodes of the left-hand side correspond to nodes in the right-hand side, and what to do with the remainder of the left-hand side.

Definition 6.14 (Term graph production, term graph rewrite systems)**.** A term graph production $p = L \leftarrow_\ell I \rightarrow_r R$ is a span consisting of three term graphs L, I and R, called the left-hand side, the interface and the right-hand side respectively, together with two interface morphisms ℓ and r which specify the correspondence of the nodes in L and R.

A term graph rewrite system (TGRS) is a set of (term graph) productions.

In our setting for $v \in I$ the partial functions lb and succ are defined for $\ell(v)$ in L if and only if they are defined for $r(v)$ in R.[1]

Since the double-pushout approach is a standard approach for describing graph transformations in several settings (as discussed in Chapter 3) we can use it to describe term graph rewriting.

Definition 6.15 (Term graph rewriting step)**.** A term graph G transforms to a term graph H by a production $p = L \leftarrow_\ell I \rightarrow_r R$ (written $G \Rightarrow_p H$) if and only if an injective morphism $g : L \rightarrow G$, a term graph D and morphisms $h : R \rightarrow H$, $i : I \rightarrow D$, $dg : D \rightarrow G$ and $dh : D \rightarrow H$ exist such that both the left square and the right square in the diagram below are *pushouts*.

$$\begin{tikzcd}
L \arrow[d, "g"'] \arrow[dr, phantom, "(\text{PO})"] & I \arrow[l, "\ell"'] \arrow[r, "r"] \arrow[d, "i"] \arrow[dr, phantom, "(\text{PO})"] & R \arrow[d, "h"] \\
G & D \arrow[l, "dg"'] \arrow[r, "dh"] & H
\end{tikzcd}$$

[1] This condition is necessary to avoid non-local unification of terms that might occur otherwise.

6.2. Interpreting Term Rewriting in Term Graph Rewriting

We will now focus on left-linear (left-hand sides of all rules are linear) and non-collapsing (right-hand sides of rules are no variables) term rewriting systems and investigate natural ways to interpret them as term graph rewriting systems. They have to be non-collapsing, since we restrict to injective morphisms in the rule.

In order to apply a term rewriting rule, that is, a rule $t \to u$ in which t and u are finite terms, to a term graph, there are two natural ways to proceed. In both ways in the corresponding production $L \leftarrow_\ell I \to_r R$, the term graph L is the term graph of t. The right-hand side corresponds to the term graph of u, where nodes for every variable that occurs in t, but not in u, are added. The main difference is in the interface I: roughly speaking in the *basic* version it is as small as possible, while in the *extended* version it is nearly a full copy of L. It is necessary to also add the additional interface structure to R in the extended version.

To motivate and define the two versions, let us first investigate what is really needed. The basic idea is that a part of the graph to be rewritten coincides with t, and that this is replaced by u. For doing so, the interface should at least contain the root of t and the variables of t.

Definition 6.16 (Basic version)**.** A term graph production $L \leftarrow_\ell I \to_r R$ based on a term rewriting rule $t \to u$ is in the *basic version* if it satisfies the following properties:

- The left-hand side L is the term graph $\mathsf{TG}(t)$
- The interface I consists of the nodes of the term graph L of t that correspond to the root of t and to the variables in t.
- The functions lb and succ are undefined for the nodes in I.
- The right-hand side R is the union of $\mathsf{TG}(u)$ with the nodes of I representing variables.
- The mappings ℓ, r map each of the nodes in I to the corresponding copy in L respectively R.

Via the two rule morphisms ℓ and r the node $v \in I$ corresponding to the root of t is mapped to the node $v' \in L$ representing the same root of t via ℓ. Likewise v is mapped to the node $v'' \in R$ representing the root of u via r. Every node in I corresponding to a variable in t is mapped to the corresponding node in L and it is similar mapped to R if the variable occurs in u.

For string rewriting, that is, term rewriting in which all symbols are unary, termination of *cycle rewriting* as studied in [ZBK14; SZ15] coincides with termination of term graph rewriting systems in the basic version; the argument that symbols of other arity, not occurring in the rewrite system, do not influence the termination property, is similar to the argument given in [BKZ14].

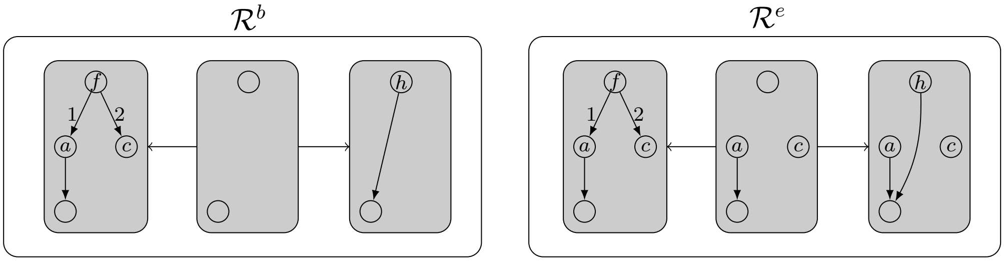
The other, *extended* version, exactly corresponds to the version as presented in [CD11], where it is shown that for orthogonal term rewriting systems, there is a correspondence between term graph rewriting and term rewriting on the corresponding unraveled (possibly infinite) terms.

Definition 6.17 (Extended version)**.** A term graph production $L \leftarrow_\ell I \rightarrow_r R$ based on a term rewriting rule $t \to u$ is in the *extended version* if it satisfies the following properties:

- The left-hand side L is the term graph $\mathsf{TG}(t)$
- The interface I is a copy of L, in which only the outgoing edges from the root are removed, that is, lb and succ are undefined for the node corresponding to the root of t
- For all nodes except the root of t, I is a copy of L in which for all nodes succ and lb is defined, and by $\ell : I \to L$ every node and edge is mapped to itself.
- The right-hand side R is the union of $\mathsf{TG}(u)$ with the interface I.
- The morphism $r : I \to R$ maps every node or edge of the interface to the corresponding item in R.

For a term rewriting system $\mathcal{R}$ we will denote by $\mathcal{R}^b(\mathcal{R}^e)$ the corresponding term graph rewriting systems in the basic (extended) version.

Example 6.18. *We interpret the term rewriting system $\mathcal{R} = \{\rho\}$ with the rule $\rho = f(a(x), c) \to h(x)$. Then $\mathcal{R}^b = \{\rho^b\}$ and $\mathcal{R}^e = \{\rho^e\}$ are the corresponding sets of productions where the interface morphisms are denoted by the node positions:*



Termination is also called strong normalization and is usually abbreviated to SN. For a term rewriting system $\mathcal{R}$ we will write:

- $\mathsf{SN}(\mathcal{R})$ if $\mathcal{R}$ is terminating on finite terms,
- $\mathsf{SN}^b(\mathcal{R})$ if $\mathcal{R}^b$ is terminating on finite term graphs,
- $\mathsf{SN}^e(\mathcal{R})$ if $\mathcal{R}^e$ is terminating on finite term graphs.

Since terms can be interpreted as term graphs, every infinite reduction of terms gives rise to an infinite reduction of term graphs in both versions. Further every step in the basic version can be mimicked by a step in the extended version. Hence if we have an infinite reduction in the basic version, then this also yields an infinite reduction in the extended version, in which left-over remainders of the left-hand sides, are ignored. This yields

$$\mathsf{SN}^e(\mathcal{R}) \implies \mathsf{SN}^b(\mathcal{R}) \implies \mathsf{SN}(\mathcal{R})$$

for all right-linear term rewriting systems $\mathcal{R}$.

The following example inspired by [Toy87] shows that right-linearity is essential for the last implication. The term rewriting system

$$\mathcal{R} = \{f(0,1,x) \to f(x,x,x), a \to 0, a \to 1\}$$

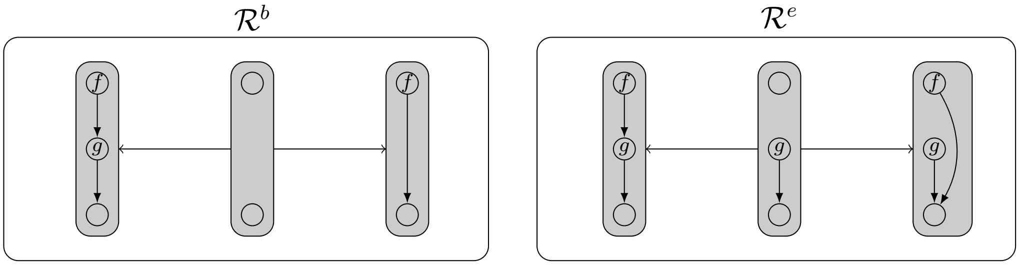
is non-terminating in term rewriting as $f(0,1,a)$ rewrites in three steps to itself:

$$\underline{f(0,1,a)} \quad \overset{\rho_1}{\to}_{\mathcal{R}} \quad f(\underline{a},a,a) \quad \overset{\rho_2}{\to}_{\mathcal{R}} \quad f(0,\underline{a},a) \quad \overset{\rho_3}{\to}_{\mathcal{R}} \quad f(0,1,a)$$

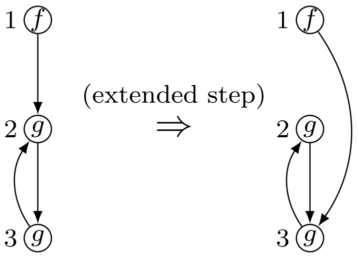
But this cannot be mimicked in term graph rewriting without doing unraveling: our techniques easily prove SN^b.

For both implications the converse does not hold, as we will show now by counterexamples. The single rule $f(g(x)) \to g(f(x))$ is the standard example of a string rewrite system that is terminating on strings but not on cycles, see [ZBK14], so this satisfies SN but not SN^b.

For the other implication, we consider the following single term rewrite rule $\rho = f(g(x)) \to f(x)$ interpreted in both, the basic version $\mathcal{R}^b = \{\rho^b\}$ (shown below left) and the extended version $\mathcal{R}^e = \{\rho^e\}$ (shown below right) where the interface morphisms are again denoted by the node positions:



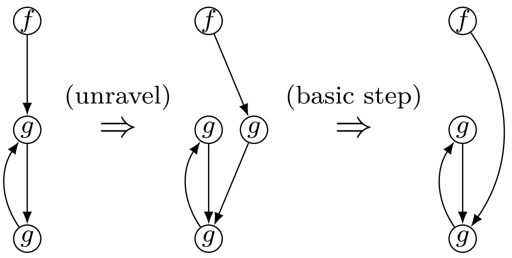
Now let the term graph depicted in the left part of the picture to the right be given. We have three nodes $1,2,3$ with $\mathsf{lb}(1) = f$, $\mathsf{lb}(2) = \mathsf{lb}(3) = g$, and $\mathsf{succ}(1) = 2$, $\mathsf{succ}(2) = 3$ and $\mathsf{succ}(3) = 2$. In the extended version, an injective morphism from L to this graph is obtained by mapping the root of L to 1, and the two nodes below it to 2 and 3 respectively. By applying the rule, the outgoing f-edge from 1 is removed, the rest of the left-hand side remains, and due to the right-hand side an edge from 1 to 3 is added, resulting in the graph depicted in the right part of the picture. As this graph is isomorphic to the original one, we see that this can be repeated forever, and the single rule $f(g(x)) \to f(x)$ does not satisfy SN^e.



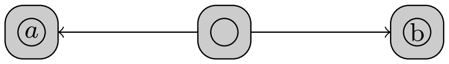
In contrast, in the basic version the rule does not apply, since the middle node 2 has an incoming edge that is not in the left-hand side, and is not part of the interface. Hence, due to the dangling edge condition, the rule cannot be applied.

An elementary argument for proving termination of the term rewriting system $f(g(x)) \to f(x)$ is by counting the number of g's: in every step the number of g's strictly decreases, so this cannot go on forever. As we will see later, this termination argument also holds for proving SN^b. Hence our single rule $f(g(x)) \to f(x)$ does not satisfy SN^e but satisfies SN^b.

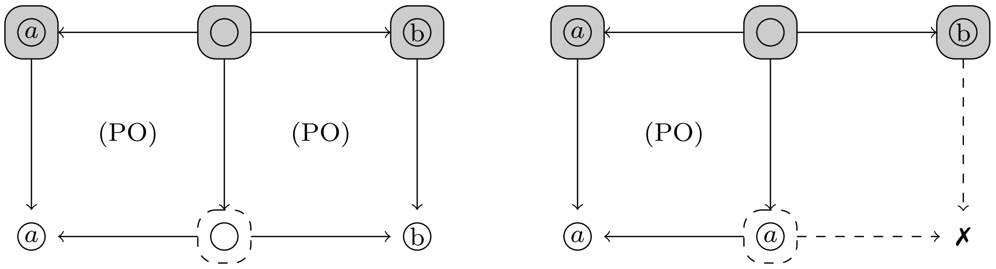
As we observed, every step in the basic version can be mimicked by a step in the extended version. Conversely, after doing a number of *unraveling* steps, a step in the extended version can be mimicked by a step in the basic version. Here unraveling means that if a node has more than one incoming edge, a new node may be created having the same outgoing edges as the original node, while one of the incoming edges points to the new node, and the others point to the original node. In the example this can be done for the middle node as depicted in the picture to the right, after which a step in the basic version can be done yielding the same result as when applying the step in the extended version directly.



It is worth to mention that the pushout complement for termgraphs, which is computed when a term graph production is applied on a term graph, may not be unique. Consider for instance the term rewrite rule $\rho = a \to b$ interpreted as the following term graph production (for this example, both versions look the same):



Applying this term graph production to the term graph representing a single constant a, results in two possible rewrite step scenarios. In the following, the term graphs of the pushout complements are highlighted in the dashed boxes:



Please note that both term graphs in the dashed boxes, shown in the diagrams above left and above right, make the left square a pushout. However, only for the unlabelled node there exists a corresponding right-hand side pushout diagram resulting in the term graph representing a single constant b. In [CD11] the authors provide properties which ensure the existence of pushouts and pushout complements for the application of term graph productions. These properties are guaranteed to hold by using so-called *evaluation rules*, which are term graph productions satisfying requirements that make them suitable to represent term rewrite rules. In the following, we restrict our term graph productions to be in the basic version or extended version. Therefore, similar to evaluation rules, our term graph productions are based on term rewrite rules (for which we want to show termination) and the existence of *minimal pushout complements* is guaranteed, i.e. those term graphs where as many nodes as possible remain unlabelled.

Our next step is to transform term graph productions in the basic version and extended version into graph transformation rules which can be analysed by the tool Grez (see Section 5.5). If the graph transformation system is proven to be terminating by Grez, we want to deduce termination of the original term rewriting system.

6.3. From Term Graph Rewriting to Graph Transformation Systems

We propose two transformations from term graphs to graphs now, both having the property that an infinite reduction in term graph rewriting translates to an infinite graph transformation reduction. The main goal is not to develop techniques specific for term graph rewriting systems, but to apply translations to graph transformation systems for which the tool Grez can be applied. Hence if a tool such as Grez can prove termination of the translated graph transformation system, then we have a proof that the original term graph rewrite system is terminating. The graphs in the graph transformation systems on which Grez can be applied differ in three ways from term graphs: nodes may have any number of outgoing edges, these outgoing edges are not numbered, and the labels are not in the nodes but on the edges. We now define two transformations from term graphs to graphs, more precisely, we will transform a term graph production to a graph transformation rule in such a way that reductions are preserved.

6.3.1. Function Encoding

The first transformation is called *function encoding*. The structure of the graph remains the same. The idea is that for a node labeled by a function symbol f of arity $n \geq 1$, the label of this node is removed, and the n ordered outgoing edges in the term graph are labeled by $f_1, \ldots, f_n$, respectively. In order to preserve constants, we introduce a fresh node $c(v)$, for every node $v \in V$ for which $\mathsf{lb}(v)$ is a constant; we write $c(V)$ for the set of all fresh nodes.

Definition 6.19 (Function encoding)**.** For a signature $\mathcal{F}$ we define the set $\mathcal{F}_F = \{f_i \mid f \in \mathcal{F}, 1 \leq i \leq \mathsf{ar}(f)\} \cup \{f \mid f \in \mathcal{F}, \mathsf{ar}(f) = 0\}$. Furthermore, for a term graph $(V, \mathsf{lb}, \mathsf{succ})$ over $\mathcal{F}$ we define the *function encoded* graph $F(V, \mathsf{lb}, \mathsf{succ}) = (V \cup c(V), E, \mathsf{src}, \mathsf{tgt}, \mathsf{lab})$ over $\mathcal{F}$, by

- $E = \{e_{v,i} \mid v \in V, 1 \leq i \leq \mathsf{ar}(\mathsf{lb}(v))\} \cup \{e_v \mid v \in V, \mathsf{ar}(\mathsf{lb}(v)) = 0)\}$,
- $\mathsf{src}(e_{v,i}) = v$ for $v \in V, 1 \leq i \leq \mathsf{ar}(\mathsf{lb}(v))$,
  $\mathsf{src}(e_v) = v$ if $\mathsf{ar}(\mathsf{lb}(v)) = 0$,
- $\mathsf{tgt}(e_{v,i}) = w_i$ for $v \in V, 1 \leq i \leq \mathsf{ar}(\mathsf{lb}(v))$ and $\mathsf{succ}(v) = w_1 w_2 \cdots w_n$,
  $\mathsf{tgt}(e_v) = c(v)$ if $\mathsf{ar}(\mathsf{lb}(v)) = 0$,
- $\mathsf{lab}(e_{v,i}) = \mathsf{lb}(v)_i$ for $v \in V, 1 \leq i \leq \mathsf{ar}(\mathsf{lb}(v))$,
  $\mathsf{lab}(e_v) = \mathsf{lb}(v)$ if $\mathsf{ar}(\mathsf{lb}(v)) = 0$.

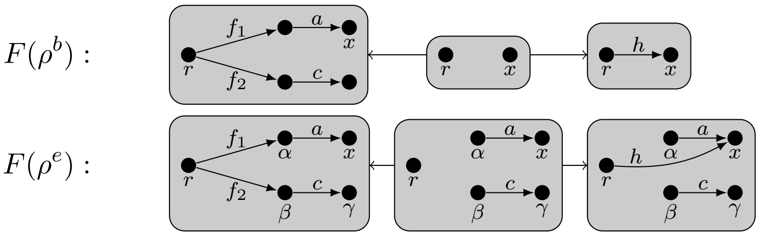
If $\varphi : (V, \mathsf{lb}, \mathsf{succ}) \to (V', \mathsf{lb}', \mathsf{succ}')$ is a term graph morphism, then the graph morphism $F(\varphi) : F(V, \mathsf{lb}, \mathsf{succ}) \to F(V', \mathsf{lb}', \mathsf{succ}')$ is defined by

- $F(\varphi)(v) = \varphi(v)$ and
  $F(\varphi)(c(v)) = c(\varphi(v))$ for $v \in V$,
- $F(\varphi)(e_{v,i}) = e_{\varphi(v),i}$ and
  $F(\varphi)(e_v) = e_{\varphi(v)}$ for every edge $e_{v,i}$ and e_v in $F(V, \mathsf{lb}, \mathsf{succ})$.

If $\rho = L \leftarrow_\ell I \rightarrow_r R$ is a term graph production, then we define the graph transformation rule by $F(\rho) = F(L) \xleftarrow{F(\ell)} F(I) \xrightarrow{F(r)} F(R)$.

It is straightforward to check that the above definition turns F into a functor.

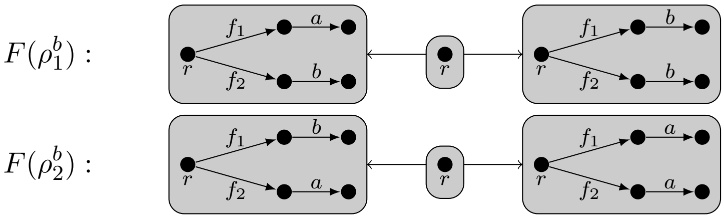
Example 6.20. *Let the term rewriting rule* $\rho = f(a(x), c) \to h(x)$ *and the term graph productions* ρ^b *and* ρ^e *from Example 6.18 be given. The graph transformation rules* $F(\rho^b)$ *and* $F(\rho^e)$ *look as follows, where the interface morphisms are denoted by the node labels:*



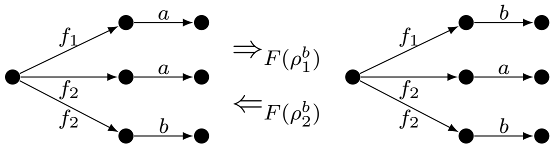
Theorem 6.21 (Termination argument via function encoding)**.** Let P be a set of term graph productions and let $\rho = L \leftarrow_\ell I \rightarrow_r R$ be a production from P. Let G, H be term graphs such that $G \Rightarrow_\rho H$, then $F(G) \Rightarrow_{F(\rho)} F(H)$. Hence, if $\{F(\rho) \mid \rho \in P\}$ is terminating, then P is terminating as well.

However, we cannot conclude termination of the graph transformation system obtained by the function encoding, if the corresponding term graph rewriting system is terminating: by definition in a term graph a node labeled by f of arity n has exactly n numbered outgoing edges that all get distinct labels when applying F. Applying the corresponding graph transformation system to a graph having nodes not satisfying this requirement may lead to an infinite reduction not having a counterpart in the term graph world. As an example consider the term rewriting system consisting of two rules $\rho_1 : f(a,b) \to f(b,b)$ and $\rho_2 : f(b,a) \to f(a,a)$. One can prove that both the basic version and the extended version of the corresponding term graph rewriting system is terminating, for instance by applying the tool Grez on the number encoding of these systems, to be defined in the next subsection.

However, the transformation for the basic version, using the function encoding, yields a graph transformation system $\{F(\rho_1^b), F(\rho_2^b)\}$ with an infinite reduction. The graph transformation system looks as follows, where the interface morphisms are denoted by the node labels:



Using the above graph transformation rules, the graph to the right yields an infinite derivation sequence as there exists a cycle. So we conclude that the function encoding for proving termination of term graph rewriting systems is sound by Theorem 6.21, but not complete.



6.3.2. Number Encoding

The reason why the function encoding is not complete is that in a graph the node corresponding to the root node of the left-hand side may have more than one outgoing f_i-edge for the same i, since this root node is part of the interface. Instead in the number encoding for non-unary symbols extra nodes and edges are added by which it is forced that the node with the numbered outgoing edges is not part of the interface, and hence is not allowed to have dangling edges.

Definition 6.22 (Number encoding)**.** For a signature $\mathcal{F}$ in which m is the greatest occurring arity, we define $\mathcal{F}_N = \mathcal{F} \cup \{1, 2, \dots, m\}$. Furthermore, for a term graph $(V, \mathsf{lb}, \mathsf{succ})$ over $\mathcal{F}$ we define the *number encoded* graph $N(V, \mathsf{lb}, \mathsf{succ}) = (V \times \{0, 1\}, E, \mathsf{src}, \mathsf{tgt}, \mathsf{lab})$ over $\mathcal{F}_N$ by

- $E = \{e_{v,i} \mid v \in V, 0 \leq i \leq \mathsf{ar}(\mathsf{lb}(v))\}$,
- $\mathsf{src}(e_{v,0}) = (v, 0)$ for $v \in V$,
  $\mathsf{tgt}(e_{v,0}) = (v, 1)$ for $v \in V$ if $\mathsf{ar}(\mathsf{lb}(v)) \neq 1$,
  $\mathsf{tgt}(e_{v,0}) = (\mathsf{succ}(v), 0)$ for $v \in V$ if $\mathsf{ar}(\mathsf{lb}(v)) = 1$,
  $\mathsf{lab}(e_{v,0}) = \mathsf{lb}(v)$ for $v \in V$,
- $\mathsf{src}(e_{v,i}) = (v, 1)$ for $v \in V$, $1 \leq i \leq \mathsf{ar}(\mathsf{lb}(v))$, $\mathsf{ar}(\mathsf{lb}(v)) \geq 2$,
  $\mathsf{tgt}(e_{v,i}) = (\mathsf{succ}(v)_i, 0)$ for $v \in V$, $1 \leq i \leq \mathsf{ar}(\mathsf{lb}(v))$, $\mathsf{ar}(\mathsf{lb}(v)) \geq 2$,
  $\mathsf{lab}(e_{v,i}) = i$ for $v \in V$, $1 \leq i \leq \mathsf{ar}(\mathsf{lb}(v))$, $\mathsf{ar}(\mathsf{lb}(v)) \geq 2$.

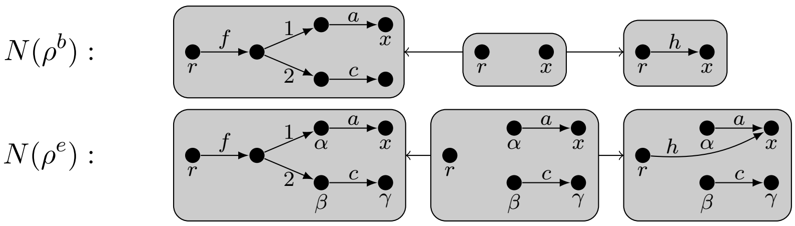
If $\varphi : (V, \mathsf{lb}, \mathsf{succ}) \to (V', \mathsf{lb}', \mathsf{succ}')$ is a term graph morphism, then the graph morphism $N(\varphi) : N(V, \mathsf{lb}, \mathsf{succ}) \to N(V', \mathsf{lb}', \mathsf{succ}')$ is defined by

- $N(\varphi)(v, i) = (\varphi(v), i)$ for $(v, i) \in V \times \{0, 1\}$,
- $N(\varphi)(e_{v,i}) = e_{\varphi(v),i}$ for every edge $e_{v,i}$ in $N(V, \mathsf{lb}, \mathsf{succ})$.

If $\rho = L \leftarrow_\ell I \rightarrow_r R$ is a term graph production, then we define the graph transformation rule by $N(\rho) = N(L) \xleftarrow{N(\ell)} N(I) \xrightarrow{N(r)} N(R)$.

Again, it is straightforward to check that above definition turns N into a functor. Note that the nodes $(v, 1)$ and the edges $e_{v,1}$ do not occur if $\mathsf{ar}(\mathsf{lb}(v)) = 1$; these nodes and edges may be ignored respectively removed.

Example 6.23. *Let the term rewriting rule $\rho = f(a(x), c) \to h(x)$ and the term graph productions ρ^b and ρ^e from Example 6.18 be given. Using the number encoding, the corresponding graph transformation rules $N(\rho^b)$ and $N(\rho^e)$ look as follows, where the interface morphisms are denoted by the node labels:*



Our main result now states, that we can preserve the termination property between term graph rewriting and graph transformation by using the number encoding on productions in a basic version. The rules of the basic number encoding automatically conform to the requirements of the following theorem.

Theorem 6.24 (Termination argument via basic number encoding)**.** Let P be a set of term graph productions, obtained by encoding term rewrite rules, all having an interface which consists only of the root and variable nodes. Then P is terminating on term graphs if and only if the graph transformation system $N(P) = \{N(\rho) \mid \rho \in P\}$ is terminating.

Therefore, we can conclude that we have soundness and completeness for the basic number encoding, since in this case the rules conform to the requirement of the theorem. For the extended number encoding the question of completeness is still open, but we can exploit soundness for our termination proofs.

The remainder of this chapter is devoted to automatically proving termination of term rewriting systems, interpreted as term graph rewriting systems in both versions (basic and extended) and converted into graph transformation systems using both encodings (function and number).

6.4. Experiments

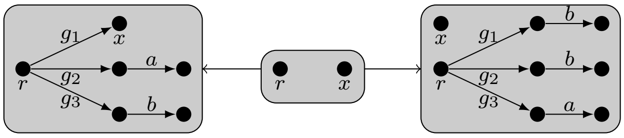
We use term rewriting systems from the *Termination Problems Database*[2] (short: TPDB). The TPDB is a collection of termination problems and is used for benchmarks in termination competitions. We encode some term rewriting systems to graph transformation systems using both, the interpretations of the basic version and the extended version together with the number and function encoding. Afterwards we try to prove termination of the resulting graph transformation systems by using the weighted type graph technique over ordered semirings from Chapter 5. The following termination proof examples were found by our prototype Java-based tool Grez (see also Section 5.5). An overview of the experiment statistics is given in Appendix B. Furthermore, a complete list of all analyzed term rewriting systems and their results can be found in Appendix B.1. The interface morphisms in the following examples will always be denoted by letters written next to the nodes.

Example 6.25. *As a first example we take the term rewriting system called* 27[3]*. This term rewriting system* $\mathcal{R} = \{\rho\}$ *where* ρ *is defined as*

$$\rho\colon\, g(x, a, b) \to g(b, b, a)$$

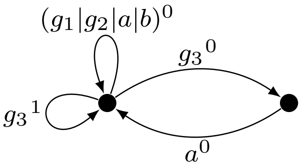
satisfies all three properties SN*,* SN^b *and* SN^e*. Termination of* $\mathcal{R}$ *follows since the number of* b*'s as third argument of the function* g *strictly decreases in each rewriting step. The transformation for the basic version, using the function encoding, results in the corresponding graph transformation system* $F(\mathcal{R}^b) = \{F(\rho^b)\}$*, where* $F(\rho^b)$ *is depicted below.*

$F(\rho^b)$:



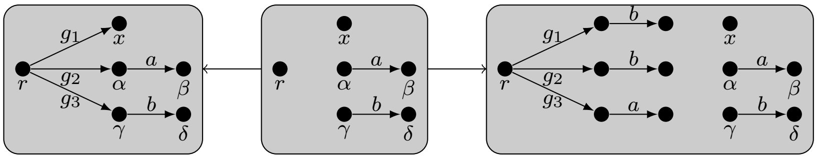
[2] http://cl2-informatik.uibk.ac.at/mercurial.cgi/TPDB/file/d43b82fe816c/TRS_Standard
[3] TPDB/TRS_Standard/Various04/27.xml

We evaluate the rule of $F(\mathcal{R}^b)$ with respect to the following weighted type graph T_{trop} over the tropical semiring. The weights are written as superscripts. There is only one possible morphism for the left-hand side graph L into T_{trop} which maps every node to the left node of T_{trop}.

$T_{trop} =$ 

For the right-hand side graph R, every node can be mapped to the left node of T_{trop}, but there exists another morphism. The target of the g_3-labeled edge can be mapped to the right node of T_{trop}. Therefore in this situation we get $w_t(\varphi_L) = min(1) = 1 > 0 = min(1,0) = w_t(\varphi_R)$ where $t\colon I \to T_{trop}$ maps all interface nodes to the left node of T_{trop}. Therefore the graph transformation system $F(\mathcal{R}^b)$ is terminating, proving $\mathsf{SN}^b(\mathcal{R})$ by Theorem 6.21.

We found a termination proof for the corresponding graph transformation system $F(\mathcal{R}^b)$ using the function encoding of the basic version. We now use the same termination proof technique for the function encoding of the extended version. In the basic version, only the root node and the nodes representing variables are part of the interface graph. The graph transformation system in the extended version preserves almost everything from the left-hand side graph. The result of applying the transformation for the extended version, using the function encoding, is the following graph transformation system $F(\mathcal{R}^e) = \{F(\rho^e)\}$:

$F(\rho^e)$: 

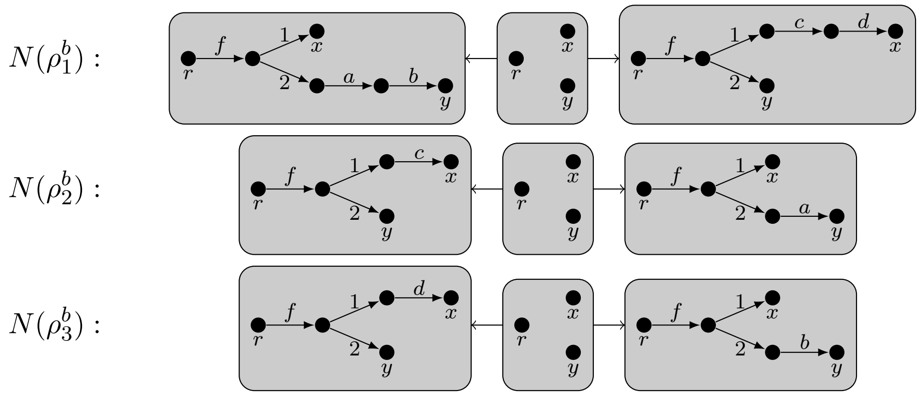
To prove termination of the graph transformation system $F(\mathcal{R}^e)$ we simply use the weighted type graph T_{trop} over the tropical semiring again. There is still only one possible morphism for the left-hand side graph L which maps all nodes to the left node of T_{trop}. The weight for the right-hand side graph R remains 0 as the weight of the additional a-, and b-labeled edges do not increase the weight since only the g_3-labeled edges have a weight of 1. The graph transformation system $F(\mathcal{R}^e)$ is terminating, proving $\mathsf{SN}^e(\mathcal{R})$ by Theorem 6.21.

Example 6.26. *As a next example we take the term rewriting system called* z17[4]. *This term rewriting system $\mathcal{R} = \{\rho_1, \rho_2, \rho_3\}$ for the three rules defined to the right satisfies* SN *and* SN^b, *but not* SN^e. *The term rewriting system $\mathcal{R}$ is terminating, as follows from proving* SN^b *as we will do now.*

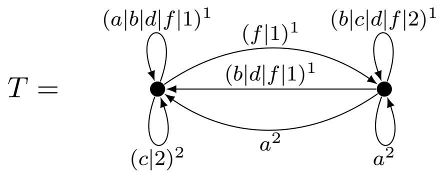
$$\begin{aligned} \rho_1 :&\quad f(x, a(b(y))) \to f(c(d(x)), y) \\ \rho_2 :&\quad f(c(x), y) \to f(x, a(y)) \\ \rho_3 :&\quad f(d(x), y) \to f(x, b(y)) \end{aligned}$$

The transformation for the basic version, using the number encoding, results in the corresponding graph transformation system $N(\mathcal{R}^b) = \{N(\rho_1^b), N(\rho_2^b), N(\rho_3^b)\}$, where $N(\mathcal{R}^b)$ is depicted below.

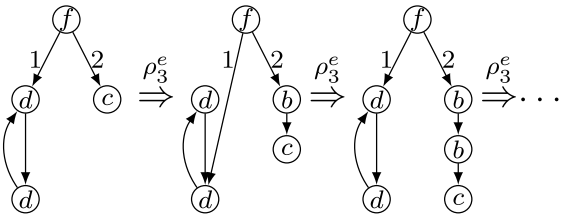
[4] TPDB/TRS_Standard/Zantema_05/z17.xml



We evaluate the rules of $N(\mathcal{R}^b)$ with respect to the following weighted type graph T over the arithmetic semiring. Again the weights are written as superscripts. We can prove that $N(\rho_1^b)$ and $N(\rho_2^b)$ are non-increasing and $N(\rho_3^b)$ is decreasing, which means that $N(\rho_3^b)$ can be removed using a relative termination argument. Due to the decreasing number of b-labeled edges in rule $N(\rho_1^b)$, which remain constant in $N(\rho_2^b)$, we can remove $N(\rho_1^b)$. Afterward we can remove $N(\rho_2^b)$ since it decreases the number of c-labeled edges. The graph transformation system $N(\mathcal{R}^b)$ is terminating, since there are no rules left. This proves $\mathsf{SN}^b(\mathcal{R})$ by Theorem 6.24.



Last we show that $\mathsf{SN}^e(\mathcal{R})$ does not hold by giving the infinite reduction below, only using ρ_3^e at every step.



Other remarkable examples, which are listed in the table of Appendix B.1, are the term rewriting system called *Ex261Composition*[5], *019*[6], *enger-nloop-toOne*[7] and *2.30*[8]. In the basic version all can be shown terminating for both the function and number encoding. The picture becomes more interesting if we restrict to the arithmetic semiring. Here, we can easily show termination with the function encoding, but not with the number encoding, for which the SMT solver does not return a solution. Hence, even though it is not complete, the function encoding has its right to exist: since it is simpler, termination proofs for the function encoding are often found in substantially less time than for the number encoding.

[5] TPDB/TRS_Standard/Applicative_05/Ex2_6_1Composition.xml
[6] TPDB/TRS_Standard/AotoYamada_05/019.xml
[7] TPDB/TRS_Standard/EEG_IJCAR_12/enger-nonloop-toOne.xml
[8] TPDB/TRS_Standard/SK90/2.30.xml

Conclusion of Part II

Over the years, automatically proving termination has been a central topic in rewriting research. Although it is widely agreed that term graph rewriting is the way to go for efficient implementation of term rewriting, quite surprisingly hardly any effort has been done in automatically proving termination of term graph rewriting. One reason could be that for term graph rewriting it is substantially harder since techniques strongly exploiting the term structure do not apply, like path orders and dependency pairs. Being close to graph transformation systems, in the previous chapters we studied how to transform term graph rewriting systems to graph transformation systems, in order to apply automated techniques for proving termination of graph transformation systems. In its turn, these techniques were inspired by and closely related to matrix interpretations of string and cycle rewriting. The weighted type graph approach does not subsume previous introduced termination analysis methods, but rather complements them. In practice one should always try several methods in parallel threads, as it is done in our termination tool Grez. As a side effect, by applying our transformations to a selection of term rewriting systems from the TPDB, we provided a substantial set of test cases for automatically proving termination of graph transformation systems.

Related Work

There is some work on termination analysis for graph transformation systems, often using rather straightforward counting arguments. Some work is specifically geared to the analysis of model transformations, taking for instance layers into account.

The weighted type graphs over general semirings technique and their application to the termination analysis of graph transformation systems is a generalization of the tropical and arctic weighted type graph based approach of the paper [BKZ14].

The paper [BH+05] considers high-level replacement units (hlru), which are transformation systems with external control expressions. The paper introduces a general framework for proving termination of such hlrus, but the only concrete termination criteria considered are node and edge counting, which are subsumed by the weighted type graph method (for more details see [BKZ14]).

In [EE+05] layered graph transformation systems are considered, which are graph transformation systems where interleaving creation and deletion of edges with the same label is prohibited and creation of nodes is bounded. The paper shows such graph transformation systems are terminating.

Another interesting approach encodes graph transformation systems into Petri nets [VV+06] by introducing one place for every edge label and transforming rules into transitions. Whenever the Petri net terminates on all markings, we can conclude uniform termination of the original graph transformation rules. Note that Example 5.22 can not be handled in this way by Petri nets. Starting with three edges labelled $0, 1, \mathit{count}$, rule ρ'_2 transforms them into three labels $0, c, \mathit{count}$, which, via rule ρ'_3, are again transformed into $0, 1, \mathit{count}$. On the other hand [VV+06] can handle negative application conditions in a limited way, a feature we did not consider here.

Another termination technique via forward closures is presented in [Plu95]. Note that the example discussed in this paper (termination of a graph transformation system based on the string rewriting rules $ab \to ac, cd \to db$) can be handled by our tool via tropical type graphs.

Furthermore in [Plu18] a modular termination analysis technique based on sequential critical pairs for hypergraph transformation systems is introduced, which resembles to a relative termination argumentation.

Not only termination techniques but also the interpretations of term rewriting systems as term graph rewriting systems has been studied in other publications as well. Traditionally, term graph rewriting is considered modulo unraveling, by which the extended version is natural. But when splitting up steps in this extended version into its building blocks of unraveling steps and basic steps, the basic version is more natural.

When restricting to unary symbols the basic version exactly corresponds to cycle rewriting as investigated in [ZBK14; SZ15].

Termination of term graph rewriting in the extended version is closer to infinitary termination of rewriting than to termination. Infinitary rewriting and its infinitary termination have been studied in [KV03; KS+05; KV05; Zan08].

Open Questions

Naturally, integration of (negative) application conditions to the graph transformation rules is an interesting direction for future work. Furthermore techniques for pattern counting, i.e. whether a given graph transformation rule always decreases the number of occurrences of a given subgraph, seems like a promising approach which could be investigated in more detail.

Another area of future research that might be of great interest is non-uniform termination analysis, i.e., to analyse whether the rules terminate only on a restricted set of graphs. In applications it is often the case that rules do not always terminate, but they terminate on all input graphs of interest (lists, cycles, trees, etc.). For this, it will be necessary to find a suitable way to characterize graph languages that is useful for the application areas and integrates well with termination analysis.

With respect to the encoding between term graph rewriting systems and graph transformation systems the question of completeness for Theorem 6.24 in case of the extended number encoding is still open.

Part III.

Specifying Graph Languages

Motivation of Part III

Formal languages in general and regular languages in particular play an important role in computer science. They can be used for pattern matching, parsing, verification and in many other domains. For instance, verification approaches such as reachability checking, counterexample-guided abstraction refinement [CG+03] and non-termination analysis [EZ15] could be directly adapted to graph transformation systems if one had a graph specification formalism with suitable closure properties, computable pre- and postconditions and inclusion checks. Inclusion checks are also important for checking when a fixpoint iteration sequence stabilizes.

While regular languages for words and trees are well-understood and can be used efficiently and successfully in applications, the situation is less satisfactory when it comes to graphs. Although the work of Courcelle [CE12] presents an accepted notion of recognizable graph languages, corresponding to regular languages, this is often not useful in practice, due to the sheer size of the resulting graph automata. Many specification formalisms that are usually used in abstract graph transformation [SWW11] and verification, are based on type graphs. For instance, shape graphs [Ren04a] can be seen as type graphs with additional annotations. Other formalisms, such as application conditions [Ren04b; HP05] and first-order or second-order logics, feature more compact descriptions, but there are problems with expressiveness, undecidability issues or unsatisfactory closure properties.[1] There are many approaches for specifying graph languages. One cannot say that one is superior to the other, usually there is a tradeoff between expressiveness and decidability properties, furthermore they differ in terms of closure properties.

Hence, it seems that there exists no one-fits-all solution. However, it is important to study and compare specification formalisms (i.e., automata, grammars and logics) that allow to specify potentially infinite sets of graphs. Therefore, our goal in this Part III is to study some selected graph specification languages and classify them according to their properties.

In fact, we investigate three different formalisms based on type graphs: first, restriction graphs R, where the language consists of all graphs that do not admit a homomorphic image of R. We also discuss the connection between type graph and restriction graph languages. Then, in order to obtain a language with better boolean closure properties, we study type graph logic, which consists of type graphs enriched with boolean connectives (negation, conjunction, disjunction). Finally, we consider annotated type graphs, where the annotations constrain the number of items mapped to a specific node or edge, somewhat similar to the proposals from abstract graph rewriting mentioned above.

In all three cases we are interested in closure properties (such as closure under union, intersection, complementation and rule application) and in decidability issues (such as decidability of the membership, emptiness and inclusion problems) and in expressiveness.

[1] A more detailed overview over related formalisms is given in the conclusion of Part III on page 101.

Outline:
After introducing additional preliminaries in Section 7.1, in Section 7.2 we consider an alternative specification formalism aside from type graph languages, namely the *restriction graph language* of a graph R, consisting of all graphs to which there is no homomorphism from R. In Section 7.3, in order to obtain languages with better boolean closure properties, we study *type graph logic*, which consists of type graphs enriched with boolean connectives. Finally, in Section 8.1 we introduce ordered monoids which are used to define annotations and multiplicities for graphs in Section 8.2. In Section 8.3 we consider *multiply annotated type graphs* which are annotated with sets of pairs of multiplicities. We investigate closure and decidability properties for all three frameworks in their corresponding subsections. All proofs can be found in Appendix A.3 and A.4.

“High achievement always takes place in the framework of high expectation.”

Charles Kettering (1876-1958)

7

Pure Type Graphs, Restriction Graphs and Type Graph Logic

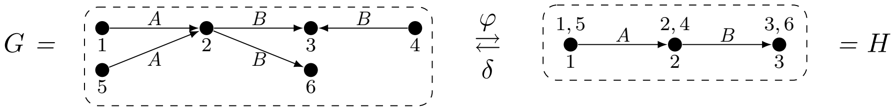
In this chapter we introduce two frameworks of graph languages. One that is characterized by a somewhat dual property of the *type graph languages* (introduced in Chapter 4) and one which is enriched by boolean operators. After a short additional preliminary section which introduces the notion of *cores*, we define *restriction graph languages* which include all graphs that *do not contain* a homomorphic image of a given *restriction graph*. Next, we discuss for type graph languages and restriction graph languages some properties such as closure under set operators, decidability of emptiness and inclusion, decidability of closure under rewriting via double-pushout rules, and discuss the relationship between these two classes of graph languages. Finally, we introduce a logic based on type graphs and analyse the closure and decidability properties of this framework as well.

7.1. Additional Preliminaries - Retracts and Cores

We revisit the concept of *retracts* and *cores* from [NT00]. They are a convenient way to minimize type graphs, and can be obtained from any graph of the class by considering so-called retracts which further reduce the graph. In [CM77], retracts were also studied under the name of *folding* to minimize conjunctive queries in relational database systems. The intuition behind a retract is, to find a proper subgraph structure of a host graph, which admits a total homomorphism from the host while mapping the subgraph structure to the host via an identity morphism.

Definition 7.1 (Retract and core)**.** A graph H is called a *retract* of a graph G if H is a subgraph of G and in addition there exists a morphism $\varphi\colon G \to H$, which is the identity when restricted to H, i.e., $\varphi_{|H} = id_H$. A graph H is called a *core* of G, written $H = core(G)$, if it is a retract of G and has itself no proper retracts.

Example 7.2. *The graph H is a retract of G, where the inclusion δ is indicated by the numbers under the nodes, while morphism φ is indicated by the numbers over the nodes:*



Since the graph H does not have a proper retract it is also a core of G.

Cores are the minimal representatives of homomorphism equivalence classes, as all graphs G, H with $G \sim H$ have a unique core up to isomorphism [NT00]. Therefore, for all type graphs describing the same type graph language, it makes sense to speak of *the* core. Please note, that the computation of the core is an NP-hard problem. This can be easily seen from a reduction from 3-colourability: Let T be a “triangle” graph with three nodes and edges connecting all pairs of distinct nodes (cf. Example 4.6). A graph G is 3-colourable if and only if the core of $G \uplus T$ (the disjoint union of G and T) is T. Therefore, in [KNN18] we employed SAT- and SMT-solvers to efficiently compute core graphs.

7.2. Type Graph and Restriction Graph Languages

As explained in Chapter 4 specifying graph languages using type graphs gives us the possibility to forbid certain graph structures by not including them into the type graph. Another way (possibly more explicit) to specify languages of graphs not including certain structures, is the following one.

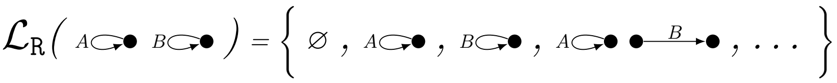
Definition 7.3 (Restriction graph language)**.** Let R be a Λ-labelled graph. Its *restriction graph language* $\mathcal{L}_{\mathtt{R}}(R)$ is defined as:

$$\mathcal{L}_{\mathtt{R}}(R) = \{G \mid R \not\to G\}.$$

Hence, $\mathcal{L}_{\mathtt{R}}(R)$ includes all graphs G that do not contain a homomorphic image of R. Restriction graph languages provide another way of representing graph languages that is in several respects dual to type graphs.

Even if the graphs used for specifying restriction graph languages are just ordinary graphs, we will in the following call them *restriction graphs* in order to emphasize their role.

Example 7.4. *The following restriction graph R over the edge label set $\Lambda = \{A, B\}$ specifies the language $\mathcal{L}_{\mathtt{R}}(R)$ consisting of those graphs which do not contain an A-labelled loop and a B-labelled loop at the same time:*



We will consider the relationship between the class of languages introduced in Definitions 4.1 and 7.3 in Section 7.2.3. Next, we investigate closure and decidability properties for both classes of languages.

7.2.1. Closure and Decidability Properties

Graph languages specified by either a type graph or a restriction graph enjoy similar decidability properties. We compare the two classes of languages with respect to decidability of the *membership*, *emptiness* and *language inclusion problem*.

Proposition 7.5 (Decidability results for type/restriction graph languages)**.** For a graph language $\mathcal{L}$ characterized by a type graph T (i.e. $\mathcal{L} = \mathcal{L}(T)$) or by a restriction graph R (i.e. $\mathcal{L} = \mathcal{L}_{\mathtt{R}}(R)$) the following problems are decidable:

1. Membership, i.e. for each graph G it is decidable if $G \in \mathcal{L}$ holds.

2. Emptiness, i.e. it is decidable if $\mathcal{L} = \emptyset$ holds.

Furthermore, language inclusion is decidable for both classes of languages:

3. Given type graphs T_1 and T_2, $\mathcal{L}(T_1) \subseteq \mathcal{L}(T_2)$ holds iff $T_1 \to T_2$.

4. Given restriction graphs R_1 and R_2, $\mathcal{L}_{\mathtt{R}}(R_1) \subseteq \mathcal{L}_{\mathtt{R}}(R_2)$ holds iff $R_1 \to\ R_2$.

Type graph and restriction graph languages enjoy the following complementary closure properties with respect to set operators. Note that the product in the category of graphs is obtained by taking separately the (cartesian) products of nodes and edges, which induce the other components in a unique way. Similarly, coproducts of graphs are built by taking coproducts (disjoint unions) of nodes and edges in the category of sets.

Proposition 7.6 (Closure properties of type/restriction graph languages)**.** Type graph languages are closed under intersection (by taking the product of type graphs) but not under union or complement.

Restriction graph languages are closed under union (by taking the coproduct of restriction graphs) but not under intersection or complement.

7.2.2. Closure under Double-Pushout Rewriting

Next, we discuss how we can show that a graph language $\mathcal{L}$ is closed under a given graph transformation rule $\rho = (L \xleftarrow{\varphi_L} I \xrightarrow{\varphi_R} R)$. Given a graph language $\mathcal{L}$, we say that it is *closed under a rule* ρ (also called *invariant*) if membership in $\mathcal{L}$ is preserved by the application of ρ, that is, for all graphs G and H such that $G \Rightarrow_\rho H$, it holds that $G \in \mathcal{L}$ implies $H \in \mathcal{L}$. For both type graph languages and restriction graph languages, separately, we characterize sufficient and necessary conditions which show that closure under rule application is decidable.

The condition for restriction graph languages is related to a condition already discussed in [HW95]. It is sufficient to check if for each overlap H of the right-hand side and the restriction graph S that S is already present in the predecessor of H, that is, the graph obtained by applying the rule backwards. If this is the case, then the absence of the homomorphic image of S is an invariant which is maintained by rule application.

Proposition 7.7 (Closure under DPO rewriting for restriction graphs)**.** A restriction graph language $\mathcal{L}_{\mathtt{R}}(S)$ is closed under a rule $\rho = (L \leftarrow\varphi_L - I - \varphi_R \rightarrow R)$ if and only if the following condition holds: for every pair of morphisms $\alpha\colon R \to F$, $\beta\colon S \to F$ which are jointly surjective, all graphs E that we obtain by applying the rule ρ with (co-)match α backwards to F, satisfy $S \to E$.

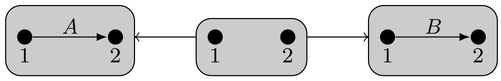
$$\begin{tikzcd}
L \arrow[d] & I \arrow[l, "\varphi_L"'] \arrow[r, "\varphi_R"] \arrow[d] & R \arrow[d, "\alpha"'] \arrow[r, dash, dashed] & S \arrow[dl, "\beta"] \arrow[dlll, dashed] \\
E & C' \arrow[l] \arrow[r] & F &
\end{tikzcd}$$

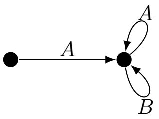
Closure under rewriting for a graph language, specified by a type graph, is decidable as long as the type graph is a core.

Proposition 7.8 (Closure under DPO rewriting for type graphs)**.** A type graph language $\mathcal{L}(T)$ is closed under a rule $\rho = (L \leftarrow\varphi_L - I - \varphi_R \rightarrow R)$ if and only if for each morphism $t_L\colon L \to \mathit{core}(T)$ there exists a morphism $t_R\colon R \to \mathit{core}(T)$ such that $t_L \circ \varphi_L = t_R \circ \varphi_R$, that is:

$$\mathcal{L}(T) \text{ is closed under application of } \rho \Leftrightarrow \begin{tikzcd}
 & \rho & \\
L \arrow[dr, "\forall t_L"'] & I \arrow[l, "\varphi_L"'] \arrow[r, "\varphi_R"] & R \arrow[dl, "\exists t_R"] \\
 & \mathit{core}(T) &
\end{tikzcd}$$

To conclude this subsection, we show that the *only if part* ($\Rightarrow$) of Proposition 7.8 cannot be weakened by considering morphisms to the type graph T, instead of to $\mathit{core}(T)$. We give a counterexample which shows that there exist type graphs, which are closed under a set of rules $\mathcal{R}$ but the diagram to the above right does not commute for all possible morphisms $t_L\colon L \to T$. In fact, consider the following type graph T and the rule ρ:

$\rho =$ 

$T =$ 

The type graph T contains the flower node, i.e., it has $T_{\ast}^{\{A,B\}}$ as subgraph. This ensures that each graph G, edge-labeled over $\Lambda = \{A, B\}$, is in the language $\mathcal{L}(T)$, and thus by rewriting any graph $G \in \mathcal{L}(T)$ into a graph H using ρ it is guaranteed that $H \in \mathcal{L}(T)$. However there is a morphism $t_L\colon L \to T$, the one mapping the A-labeled edge of L to the left A-labeled edge of T, such that there exists no morphism $t_R\colon R \to T$ satisfying $t_L \circ \varphi_L = t_R \circ \varphi_R$. Namely, by mapping the binary A-labelled edge in L to the non-loop edge in T. Due to the absence of a parallel B-labelled edge the diagram can not commute.

7.2.3. Relating Type Graph and Restriction Graph Languages

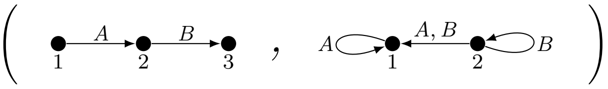
Both type graph and restriction graph languages specify collections of graphs by forbidding the presence of certain structures. This is more explicit with the use of restriction graphs, though. A natural question is how the two classes of languages are related. A partial answer to this is provided by the notion of *duality* *pairs* and by an important result concerning their existence, presented in [NT00].[1] Intuitively a duality pair consists of a type graph and a restriction graph whose graph languages are equivalent.

Definition 7.9 (Duality pair)**.** Given two graphs R and T, we call T the *dual* of R if for every graph G it holds that $G \rightarrow T$ if and only if $R \nrightarrow G$. In this case the pair (R, T) is called a *duality pair.*

Clearly, we have that (R, T) is a duality pair if and only if the restriction graph language $\mathcal{L}_{\mathtt{R}}(R)$ coincides with the type graph language $\mathcal{L}(T)$.

Example 7.10. *Let $\Lambda = \{A, B\}$ be given. The following is a duality pair:*

$(R, T) =$



Since node 1 of T is not the source of a B-labeled edge and node 2 is not the target of an A-labeled edge, for every graph G we have $G \rightarrow T$ iff it does not contain a node which is both the target of an A-labeled edge and the source of a B-labeled edge. But it contains such a node if and only if $R \rightarrow G$.

One can identify the class of restriction graphs for which a corresponding type graph exists which defines the same graph language.

Fact 7.11 (Duality pair construction [NT00])**.** Results from [NT00] state[2]that given a core graph R, a graph T can be constructed such that (R, T) is a duality pair if and only if R is a tree.

Thus we have a precise characterisation of the intersection of the classes of type and restriction graph languages: $\mathcal{L}$ belongs to the intersection if and only if it is of the form $\mathcal{L} = \mathcal{L}_{\mathtt{R}}(R)$ and *core*(R) is a tree. The results from [NT00] can be used to construct a type graph T from a given restriction graph R. It is worth mentioning that the construction of T from R contains two exponential blow-ups. However, the results stated in [NT00] can not be used to derive a construction of R from T.

All this can be interpreted by saying that type graphs have limited expressiveness if used to forbid the presence of certain structures.

7.3. Type Graph Logic

In this section we investigate the possibility to define a language of graphs using a logical formula over type graphs. Such a logic could alternatively also be defined based on restriction graphs. A related logic, for injective occurrences of restriction graphs, is studied in [OEP08], where the authors also give a decidability result via inference rules.

[1] Note that in [NT00] graphs are simple, but it can be easily seen that for our purposes the results can be transferred straightforwardly.

[2] This refers to Lemma 2.3, Lemma 2.5 and Theorem 3.1 in [NT00].

We start by defining the syntax and semantics of a *type graph logic* (*TGL*).

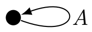
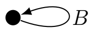
Definition 7.12 (Syntax and semantics of *TGL*)**.** A *TGL* formula F over a fixed set of edge labels Λ is formed according to the following grammar:

$$F := T \mid F_1 \vee F_2 \mid F_1 \wedge F_2 \mid \neg F, \qquad \text{where } T \text{ is a type graph.}$$

Each *TGL* formula F denotes a graph language $\mathcal{L}(F) \subseteq |\mathbf{Graph}_\Lambda|$ defined by structural induction as follows:

$$\mathcal{L}(T) = \{G \in |\mathbf{Graph}_\Lambda| \mid G \to T\} \qquad \mathcal{L}(\neg F) = |\mathbf{Graph}_\Lambda| \setminus \mathcal{L}(F)$$
$$\mathcal{L}(F_1 \wedge F_2) = \mathcal{L}(F_1) \cap \mathcal{L}(F_2) \qquad \mathcal{L}(F_1 \vee F_2) = \mathcal{L}(F_1) \cup \mathcal{L}(F_2)$$

Example 7.13. *Let the following TGL formula F over $\Lambda = \{A, B\}$ be given:*

$F = \neg$  $\bigwedge \neg$ 

The graph language $\mathcal{L}(F)$ consists of all graphs which do not consist exclusively of A-edges or of B-edges, i.e., which contain at least one A-labeled edge and at least one B-labeled edge, something that can not be expressed by pure type graphs.

With respect to the closure properties we get more positive results when using the *TGL* formula instead of a pure type graph, which we will show next.

7.3.1. Closure and Decidability Properties for Type Graph Logic

Using pure type graphs to specify the type graph language we got negative results with respect to the closure under union and complement (see Proposition 7.6). In the case of the type graph logic we can use the semantics of the formulae to get positive results for all the desired closure properties.

Proposition 7.14 (Closure properties of *TGL*)**.** Graph languages $\mathcal{L}(F)$ characterized by a *TGL* formula F, are closed under union, intersection and complement.

We now present some positive results for graph languages over *TGL* formulas with respect to decidability problems. Due to the conjunction and negation operator, the emptiness (or unsatisfiability) check is not as trivial as it is for pure type graphs. Note also that thanks to the presence of boolean connectives, inclusion can be reduced to emptiness.

Proposition 7.15 (Decidability properties of *TGL*)**.** For a graph language $\mathcal{L}(F)$ characterized by a *TGL* formula F, the following problems are decidable:

- Membership, i.e. for all graphs G it is decidable if $G \in \mathcal{L}(F)$ holds.
- Emptiness, i.e. it is decidable if $\mathcal{L}(F) = \emptyset$ holds.
- Language inclusion, i.e. given two *TGL* formulas F_1 and F_2 it is decidable if $\mathcal{L}(F_1) \subseteq \mathcal{L}(F_2)$ holds.

As already mentioned in the introduction of Part III, the different specification frameworks have the purpose to be used for verification. Therefore, it is natural to ask for a framework which is able to compute weakest preconditions and strongest postconditions, i.e. given a graph language and a set of rules, we want to specify the language of all successors (or the language of all predecessors) in our formalism. However, neither type graphs nor the type graph logic can count and hence can not express that all items of the newly created right-hand occur exactly once. The computation of postconditions is impossible in these frameworks.

Therefore, in the next chapter we will improve the expressiveness of the type graphs themselves, rather than using an additional logic to do so.

"Truth is ever to be found in simplicity, and not in the multiplicity and confusion of things."

Isaac Newton (1642-1727)

8 Annotated Type Graphs

In this chapter we will equip graphs with additional annotations, thus making our graph languages more expressive. As explained in the introduction of Part III, this idea was already used similarly in abstract graph rewriting [Ren04a]. In contrast to most other approaches, we will investigate the problem from a categorical point of view. Another reason for considering annotated type graphs is that they are a suitable formalism for computing post-conditions, a task which is studied in more detail in Chapter 10.

8.1. Additional Preliminaries - Ordered Monoids

We will annotate each type graph with pairs of annotations, denoting upper and lower bounds for nodes and edges. A graph will belong to the corresponding language only if it has a morphism to the type graph satisfying such bounds.

The bounds of the type graph are elements from an ordered monoid, i.e. a monoid which comes equipped with a partial order for its elements.

Definition 8.1 (Ordered monoid)**.** An *ordered monoid* $(\mathcal{M}, +, \leq)$ consists of a set $\mathcal{M}$, a partial order $\leq$ and a binary operation $+$ such that

- $(\mathcal{M}, +, 0)$ is a monoid.
- The partial order is compatible with the monoid operation, in particular $a \leq b$ implies $a + c \leq b + c$ and $c + a \leq c + b$ for all $a, b, c \in \mathcal{M}$.

An ordered monoid is *commutative* if $+$ is commutative.

We denote by **Mon** the category having ordered monoids as objects and monoid homomorphisms which are monotone as arrows. In the following we will sometimes denote an ordered monoid by its underlying set.

Definition 8.2 (Homomorphisms of ordered monoids)**.** Let $\mathcal{M}_1$, $\mathcal{M}_2$ be two ordered monoids. A map $h\colon \mathcal{M}_1 \to \mathcal{M}_2$ is called *monotone* if $a \leq b$ implies $h(a) \leq h(b)$ for all $a, b \in \mathcal{M}_1$. It is called a *homomorphism* if in addition $h(0) = 0$ and $h(a + b) = h(a) + h(b)$.

Example 8.3. *Let $n \in \mathbb{N}$ and take $\mathcal{M}_n = \{0, 1, \dots, n, *\}$ (zero, one, ..., n, many) with $0 \leq 1 \leq \dots \leq n \leq *$ and addition as monoid operation with the proviso that for $x, y \in \mathcal{M}_n$, we have $x + y = *$ if the sum is larger than n. In addition $x + * = *$ for all $x \in \mathcal{M}_n$.*

Furthermore, given a set S and an ordered monoid $(\mathcal{M}, +, \leq)$, it is easy to check that $(\mathcal{M}^S, +, \leq)$ is an ordered monoid, where the elements of the *exponentiation* $\mathcal{M}^S = \{a \colon S \to \mathcal{M}\}$ are functions from S to $\mathcal{M}$ and both, the partial order and the monoidal operation, are taken pointwise. An ordered monoid like $\mathcal{M}_n$ (the monoid given in the previous Example 8.3) can be used to count the number of nodes or edges contained in a graph. However, there are other monoids, which can for instance be used to specify whether there exists a path between two nodes.

The following *path monoid* $\mathcal{P}_G$ is useful if we want to annotate a graph with information over which paths are present. Note that due to the folding caused by the abstraction, a path in the type graph does not necessarily imply the existence of a corresponding path in a graph of the language. Hence annotations based on such a monoid can yield useful additional information.

Example 8.4. *Given a graph G and the transitive closure $E_G^+ \subseteq V_G \times V_G$ of the edge relation $E_G = \{(src_G(e), tgt_G(e)) \mid e \in E_G\}$. The* path monoid *$\mathcal{P}_G$ of G has the carrier set $\mathcal{P}(E_G^+)$, i.e. the powerset of the transitive closure E_G^+. The partial order is simply set inclusion and the monoid operation is defined as follows: given $P_0, P_1 \in \mathcal{P}_G$, we have*

$$\begin{aligned} P_0 + P_1 \quad = \quad & \{(v_0, v_n) \mid \exists v_1, \dots, v_{n-1} \colon (v_i, v_{i+1}) \in P_{j_i}, \\ & \qquad i \in \{0, \dots, n-1\}, j_0 \in \{0, 1\}, j_{i+1} = 1 - j_i\} \end{aligned}$$

That is, new paths can be formed by concatenating alternating path fragments from P_0, P_1. It is obvious to see that $+$ is commutative and one can also show associativity. $P = \emptyset$ is the unit.

Using ordered monoids we are now ready to define annotations for our graphs and how they are used to make graph languages more expressive.

8.2. Annotations and Multiplicities

The annotations are defined, in general, as elements of an ordered monoid, and in the concrete case that we use as running example they are functions mapping nodes and edges of the type graph to corresponding multiplicities. This idea has earlier been proposed in [Kön99]. Please note that the following formal definition of our annotations can easily be extended to a functor for objects from an arbitrary given category to **Mon**.

Definition 8.5 (Annotations)**.** Given a functor $\mathcal{A}\colon \mathbf{Graph}_\Lambda \to \mathbf{Mon}$, an *annotation based on* $\mathcal{A}$ for a graph G is an element $a \in \mathcal{A}(G)$. We assume that for each graph G there is a *standard annotation* based on $\mathcal{A}$ that we denote by s_G, thus $s_G \in \mathcal{A}(G)$.

An annotation functor assigns an ordered monoid to every graph. We write $\mathcal{A}_\varphi$, instead of $\mathcal{A}(\varphi)$, for the action of functor $\mathcal{A}$ on a graph morphism φ. To make this notion of an annotation functor easier to understand we introduce a concrete instance of the functor $\mathcal{A}$ in form of the functor $\mathcal{B}^n$.

Definition 8.6 (Multiplicities)**.** Given an ordered monoid $\mathcal{M}_n = \{0, \ldots, n, *\}$ with $n \in \mathbb{N}_0$ we define the functor $\mathcal{B}^n : \mathbf{Graph}_\Lambda \to \mathbf{Mon}$ as follows:

- for every graph G, $\mathcal{B}^n(G) = \{a\colon (V_G \cup E_G) \to \mathcal{M}_n\} = \mathcal{M}_n^{V_G \cup E_G}$;
- for every graph morphism $\varphi\colon G \to G'$ and $a \in \mathcal{B}^n(G)$, we have $\mathcal{B}^n_\varphi(a) \in \mathcal{B}^n(G')$ with:

$$\mathcal{B}^n_\varphi(a)(y) = \sum_{\varphi(x)=y} a(x), \quad \textit{where } x \in (V_G \cup E_G) \textit{ and } y \in (V_{G'} \cup E_{G'})$$

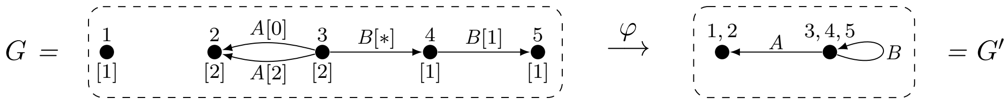
  Notice that, if y is not in the image of φ, $\mathcal{B}^n_\varphi(a)(y) = 0$.

Annotations based on the functor $\mathcal{B}^n$ are called *multiplicities.*

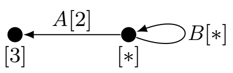
For a graph G, its *standard multiplicity* $s_G \in \mathcal{B}^n(G)$ is defined as the function which maps every node and edge of G to 1.

The action of the functor $\mathcal{B}^n$ on a morphism transforms a multiplicity by summing up (in $\mathcal{M}_n$) the values of all items of the source graph that are mapped to the same item of the target graph. Please note, that in the specific case of the functor $\mathcal{B}^n$ (our running example in this thesis) the ordered monoid is not $\mathcal{M}_n$ but the set of possible annotation functions from the graph items to $\mathcal{M}_n$. Therefore an annotation based on a functor $\mathcal{B}^n$ associates every node or edge of a graph with a number (or the top value $*$).

Example 8.7. *Let the following graphs G and G' with multiplicities $a \in \mathcal{B}^3(G)$ be given. The multiplicity of each graph element is indicated by the element of $\mathcal{M}_3 = \{0, 1, 2, 3, *\}$ shown in the brackets. Furthermore, let $\varphi\colon G \to G'$ be a graph morphism indicated by the numbers above the nodes.*



Let $a' = \mathcal{B}^3_\varphi(a)$. We compute $a'(y)$ for each element $y \in G'$ by summing up the multiplicities of all preimages, i.e. they build the sum of all $a(x) \in \mathcal{B}^3(G)$ for which $\varphi(x) = y$. The annotation a' is depicted on the right.



Some of the results that we will present in the rest of the chapter will hold for annotations based on a generic functor $\mathcal{A}$, some only for annotations based on functors $\mathcal{B}^n$, i.e. for multiplicities.

Note that the multiplicity functor $\mathcal{B}^n$ is only one of many possible instances of our generic annotation functor $\mathcal{A}$. For instance, we can consider an annotation functor which records the out-degree of a node or we can consider an annotation functor based on the path monoid from Example 8.4. Such a *path annotation* functor $\mathcal{T}$ is defined below.

Definition 8.8 (Path annotation)**.** The functor $\mathcal{T} : \mathbf{Graph}_\Lambda \to \mathbf{Mon}$ is defined as follows:

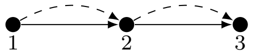
- for every graph G, $\mathcal{T}(G) = \mathcal{P}_G$;
- for every graph morphism $\varphi\colon G \to G'$ and $P \in \mathcal{T}(G)$, we have $\mathcal{T}_\varphi(P) \in \mathcal{P}_{G'}$ with:
$$\mathcal{T}_\varphi(P) = \{(\varphi(v), \varphi(w)) \mid (v, w) \in P\}.$$

For a graph G, its *standard annotation* $s_G \in \mathcal{T}(G)$ is the transitive closure of the edge relation, i.e., $s_G = E_G^+$.

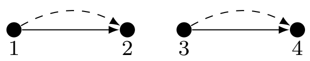
Equipping a type graph with an annotation makes our specified graph languages more expressive since it gives rise to additional properties which need to be satisfied by all graphs contained in the language.

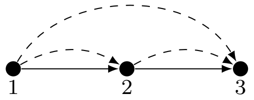
Definition 8.9 (Graph languages of annotated type graphs)**.** We say that a graph G with standard annotation $s_G \in \mathcal{A}(G)$ is represented by an annotated type graph $T[a]$ with $a \in \mathcal{A}(T)$ whenever there exists a morphism $\varphi\colon G \to T$ such that $\mathcal{A}_\varphi(s_G) \le a$. We will write $G \in \mathcal{L}(T[a])$ in this case.

Example 8.10. *Consider the annotated type graph $T[P_T]$ with the path annotation $P_T = \{(1,2),(2,3)\} \in \mathcal{T}(T)$ shown to the right.*

$T[P_T] =$ 

As evident from the picture we will denote the possible existence of a path between nodes by additional dashed parallel edges. Therefore, the annotated type graph $T[P_T]$ specifies a graph language consisting of standard annotated graphs $G[s_G]$ for which there exists a graph morphism $\varphi\colon G \to T$ such that there may exist a path between nodes that are being mapped to the type graph nodes 1 *and* 2 *and similar to the nodes* 2 *and* 3*, but there must not be a path between the nodes in G which are mapped to the type graph nodes* 1 *and* 3*. For instance, the graph $G[s_G]$ (shown below to the left) with $s_G = \{(1,2),(3,4)\} \in \mathcal{T}(G)$ satisfies the condition in T, i.e. $G \in \mathcal{L}(T[P_T])$, due to the fact that there exists a graph morphism $\varphi\colon G \to T$ for which we get $\mathcal{T}_\varphi(s_G) \subseteq P_T$.*

$G[s_G] =$ 

$G'[s_{G'}] =$ 

However, the standard annotated graph $G'[s'_G]$ (shown above to the right) is not included in the graph language specified by $T[P_T]$, i.e. $G' \notin \mathcal{L}(T[P_T])$, since the standard annotation $s'_G = \{(1,2),(2,3)(1,3)\} \in \mathcal{T}(G')$ describes a path from node 1 *to node* 3 *due to the transitive closure of the edge relation.*

Please note that compared to the pure type graph languages $\mathcal{L}(T)$ from Chapter 4 we do not have the property that a type graph T is always contained in its own specified language (see also Example 4.3). For instance, in Example 8.10 above we have $G' \cong T$ but since $G' \notin \mathcal{L}(T[P_T])$ we can conclude $T \notin \mathcal{L}(T[P_T])$.

Furthermore, a type graph may not only have a single annotation but it can be equipped with a set of annotations, as we will see in the upcoming sections.

8.3. Multiply Annotated Graphs

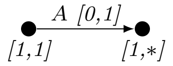
The type graphs which we are going to consider are enriched with a set of pairs of annotations. Each pair can be interpreted as establishing a lower and an upper bound to what a graph morphism can map to the graph. The motivation for considering multiple (pairs of) annotations rather than a single one is mainly to ensure closure under union.

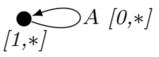
Definition 8.11 (Multiply annotated graphs)**.** Let $\mathcal{A}\colon \mathbf{Graph}_\Lambda \to \mathbf{Mon}$ be a functor. A *multiply annotated graph* $G[M]$ *(over* $\mathcal{A}$*)* is a graph G equipped with a finite set of pairs of annotations $M \subseteq \mathcal{A}(G) \times \mathcal{A}(G)$, such that $\ell \leq u$ for all $(\ell, u) \in M$. We will write $G[\ell, u]$ as an abbreviation of $G[\{(\ell, u)\}]$.

An arrow $\varphi\colon G[M] \to G'[M']$, also called a *legal morphism*, is a graph morphism $\varphi\colon G \to G'$ such that for all $(\ell, u) \in M$ there exists $(\ell', u') \in M'$ with $\mathcal{A}_\varphi(\ell) \geq \ell'$ and $\mathcal{A}_\varphi(u) \leq u'$.

In case of annotations based on $\mathcal{B}^n$, we will often call a pair (ℓ, u) a *double multiplicity*.

Example 8.12. *Consider the following multiply annotated graphs (over* $\mathcal{B}^2$*)* $G[\ell, u]$ *and* $H[\ell', u']$*, both having one double multiplicity.*

$G[\ell, u] =$ 

$H[\ell', u'] =$ 

Referring to the picture, for example $G[\ell, u]$ *is defined as follows: If* e *is the single edge of* G*, labelled by* A*, then* $\ell, u\colon \{e, src_G(e), tgt_G(e)\} \to \mathcal{M}_2$*, and in particular* $\ell(e) = 0, \ell(src_G(e)) = \ell(tgt_G(e)) = 1, u(e) = u(src_G(e)) = 1$*, and* $u(tgt_G(e)) = *$*. Thus multiplicities are represented by writing the lower and upper bounds next to the corresponding graph elements. Note that there is a unique, obvious graph morphism* $\varphi\colon G \to H$*, mapping both nodes of* G *to the only node of* H*. Concerning multiplicities, by adding the lower and upper bounds of the two nodes of* G*, one obtains the interval* $[2, *]$ *which is included in the interval of the node of* H*,* $[1, *]$*. Similarly, the double multiplicity* $[0, 1]$ *of the edge of* G *is included in* $[0, *]$*. Therefore, since both* $\mathcal{B}^2_\varphi(\ell) \geq \ell'$ *and* $\mathcal{B}^2_\varphi(u) \leq u'$ *hold, we can conclude that* $\varphi\colon G[\ell, u] \to H[\ell', u']$ *is a legal morphism.*

Multiply annotated graphs and legal morphisms form a category.

Lemma 8.13. The composition of two legal morphisms is a legal morphism.

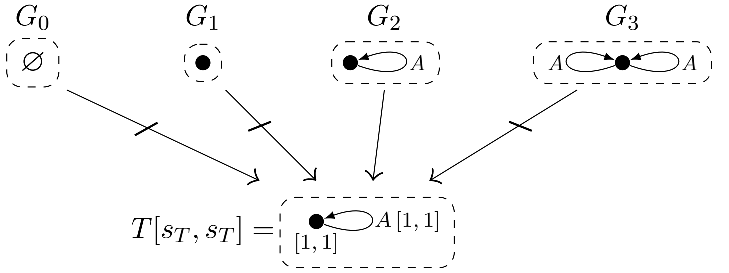
We are now ready to define how a graph language $\mathcal{L}(T[M])$ looks like.

Definition 8.14 (Graph languages of multiply annotated type graphs)**.** We say that a graph G is represented by a multiply annotated type graph $T[M]$ whenever there exists a legal morphism $\varphi\colon G[s_G, s_G] \to T[M]$, i.e., there exists $(\ell, u) \in M$ such that $\ell \leq \mathcal{A}_\varphi(s_G) \leq u$. We will write $G \in \mathcal{L}(T[M])$ in this case.

It follows from the definition that whenever $M = \emptyset$, then we get $\mathcal{L}(T[M]) = \emptyset$. In the special case of the functor $\mathcal{B}^n$, the graph language $\mathcal{L}(T[M])$ coincides with $\mathcal{L}(T)$ from Definition 4.1, if there exists a double multiplicity $(\ell, u) \in M$ which assigns $[0, *]$ to every node and edge in T.

Note that pure type graph languages $\mathcal{L}(T)$ are always infinite if $T \neq \emptyset$. However, if we use annotations based on $\mathcal{B}^n$, then a graph language $\mathcal{L}(T[M])$ is finite up to isomorphism if there does not exist a double multiplicity $(\ell, u) \in M$ with u assigning the upper bound $*$ to at least one node or edge in T.

Example 8.15. *Let the standard annotated type graph $T[s_T, s_T]$ in the figure below be given. As evident from the figure, the resulting graph language $\mathcal{L}(T[s_T, s_T])$ only contains one graph, that is T, up to isomorphism.*



Both, G_0 and G_1 do not satisfy the lower bound of the A-labelled loop of $T[s_T, s_T]$. Furthermore, for G_3 one would need to map both of its A-labelled loops to the single loop of $T[s_T, s_T]$. Since annotations (in this case the standard annotation 1) are summed up, we would violate the upper bound of the targeted loop. Only for the graph G_2 (which represents $T[s_G, s_G]$) there exists a legal morphism into the annotated type graph, which respects all lower and upper bounds.

Multiply annotated type graphs have a higher expressiveness compared to pure type graphs and restriction graphs, since one can enforce certain structures to appear in the language, by increasing the lower bounds as thresholds. The aim of this thesis is to find specification mechanisms, which are useful for the verification of systems. Before investigating decidability results and closure properties for the graph languages specified by multiply annotated type graphs, we show some example properties, that can be expressed by annotated type graphs.

Example 8.16. *In order to illustrate the use of annotated type graphs in applications, we model a client-server scenario with the following specification:*

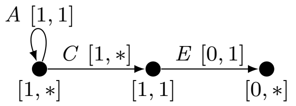
- *There exists exactly one server.*
- *An arbitrary number of users can connect to the server, even using multiple connection sessions at the same time.*
- *There exists one user with special administrative rights.*
- *At least one user is always connected to the server.*
- *The server can host an arbitrary number of files from which at most one can be edited at the same time.*

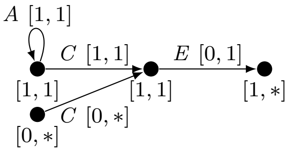
The scenario above can be modelled using an annotated type graph $T_1[\ell, u]$ (see below). We will use the following edge labels: A-labelled loops for administrative rights, C-labelled edges for connections between users and the server and E-labelled edges which are pointing to the file that is currently edited. The left-hand node represents users, the middle node the server and the right-hand node files.

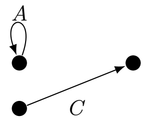
Now we extend the requirements of our specification:

- *The user with the administrative rights is always connected to the server.*
- *There has to be at least one file on the server.*

We use the annotated type graph $T_2[\ell', u']$ (depicted below to the right), to model the extended scenario.

$T_1[\ell, u] =$ 

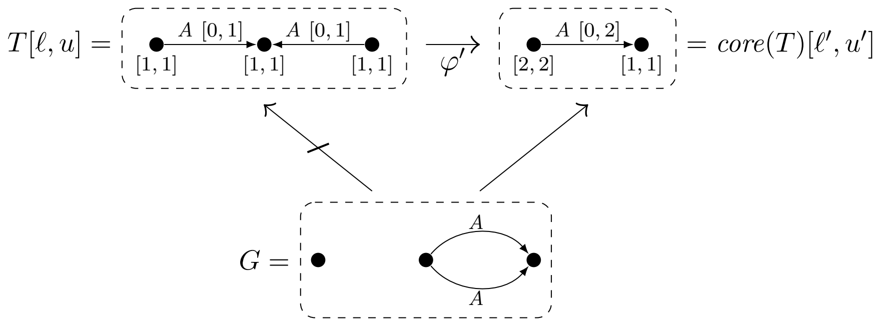
$T_2[\ell', u'] =$ 



Since the second scenario is more restrictive than the first, there exist graphs in $\mathcal{L}(T_1[\ell, u])$, which do not fulfill the additional requirements of the extended specification. For instance the graph G shown to the right is such a model, which describes that there exists a user with administrative rights but he is not connected to the server.

In contrast to type graph languages, core graphs can not be used to minimize annotated type graphs since they do not preserve graph languages specified by multiply annotated type graphs.

Example 8.17. *Consider the following annotated type graph $T[\ell, u]$ together with the corresponding core graph core(T) alongside the unique graph morphism $\varphi\colon T \to core(T)$. We annotate the core with the double multiplicity $(\ell', u') = (\mathcal{B}^2_\varphi(\ell), \mathcal{B}^2_\varphi(u))$ such that $\varphi'\colon T[\ell, u] \to core(T)[\ell', u']$ becomes a legal morphism in the framework of annotated type graphs. Then graph G is a witness for the fact that $\mathcal{L}(T[\ell, u]) \neq \mathcal{L}(core(T)[\ell', u'])$ since $G \in \mathcal{L}(core(T)[\ell', u'])$ but $G \notin \mathcal{L}(T[\ell, u])$.*



The double multiplicity $(\ell', u') = (\mathcal{B}^2_\varphi(\ell), \mathcal{B}^2_\varphi(u))$ is not the only possibility to annotate core(T). However, by either increasing ℓ' or decreasing u', the morphism φ' will no longer be legal and one could easily find a graph contained in $\mathcal{L}(T[\ell, u])$ but not in $\mathcal{L}(core(T)[\ell', u'])$. On the other hand, by either decreasing ℓ' or increasing u' the graph G remains as a witness for the fact that the language is not preserved. Therefore, we can conclude that there can not exist an annotated core graph $core(T)[\ell', u']$ of $T[\ell, u]$ specifying the same language.

8.3.1. Local vs. Global Annotations

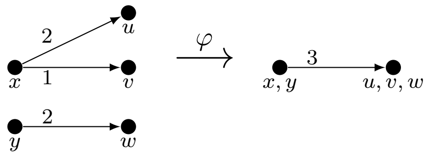
Note that our multiplicities are global, i.e., we count *all* items that are mapped to a specific item in the type graph. This holds also for edges, as opposed to UML multiplicities, which are local wrt. the classes which are related by an edge (i.e., an association). Consider for instance two nodes A, B which might also represent classes in an UML class diagram. Assume that these nodes are connected by an edge with UML multiplicity 1 at both endpoints. This means that each instance of A has exactly one outgoing edge to an instance of B. (In other words: if we restrict to edges going to B-nodes, each A-node has out-degree 1.) A symmetric condition holds for instances of B. However, there could be many instances of both A and B and hence many edges between such instances. Similar local multiplicity constraints are considered in [Ren04a; BB+08].

We will show that it is not straightforward to integrate such annotations into our framework, since they are not always functorial in the sense of Definition 8.5. We will demonstrate this with an example.

Example 8.18. *For the sake of this example we assume that we have local annotations restricting the out-degree of a node. For instance in the graph T below on the left there can be at most two edges going from a fixed instance of x to any instance of u. Formally, an annotation for a graph T is a function $a\colon V_T \times E_T \to \mathbb{N}_0$, where $a(v, e)$ is an upper bound for the number of instances of $e \in E_T$ attached to a fixed instance of $v \in V_T$.*

Now consider the morphism $\varphi\colon T \to T'$ below, which is denoted by the node labels. The question is – given the annotation for T – how to determine the annotation for T'. Naturally, this should be done in such a way that the existence of a legal morphism implies language inclusion.

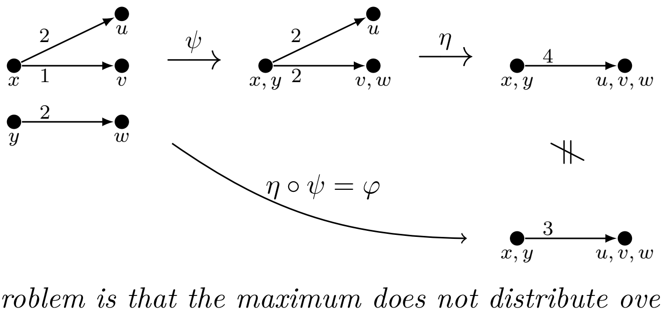
Hence, the standard procedure would be to first add up the local multiplicities (for instance $2 + 1 = 3$, since u, v are mapped to the same node) and then take the maximum over all multiplicities ($\max\{3, 2\} = 3$).



Formally this can be written as

$$\mathcal{A}_\varphi(a)(v_2, e_2) = \bigvee_{\varphi(v_1)=v_2} \sum_{\varphi(e_1)=e_2} a(v_1, e_1).$$

However, if we try to do this in our framework, the functor does not preserve composition, i.e. $\mathcal{A}_{\eta\circ\psi}(a) \neq \mathcal{A}_\eta(\mathcal{A}_\psi(a))$. We consider the following counterexample where $\eta \circ \psi = \varphi$ and φ is the morphism which we just considered:



The main problem is that the maximum does not distribute over the sum and we get $\max\{2 + 1, 2\} = 3 \neq 4 = \max\{2, 2\} + \max\{1, 2\}$.

This counterexample of course calls for an extension of our abstract annotation framework. One solution might be to allow lax functors, but a generalization of our results is not straightforward. (For instance, the proof of the closure under intersection, Proposition 8.33, requires functoriality.) An alternative solution is to define the annotations in a way such that they yield a functor. We will introduce a similar local annotation, which fits into our annotation framework, in Chapter 10.

8.3.2. Decidability Properties for Multiply Annotated Graphs

We now address some decidability problems for languages defined by multiply annotated graphs. We obtain positive results, under mild assumptions, with respect to the membership and emptiness problems. However, for decidability of language inclusion we only obtain partial results.

For the membership problem we can enumerate all graph morphisms $\varphi\colon G \to T$ and check if there exists a legal morphism $\varphi\colon G[s_G, s_G] \to T[M]$, i.e. if there exists a pair of annotations $(\ell, u) \in M$ with $\ell \leq \mathcal{A}_\varphi(s_G) \leq u$. Clearly, this is decidable if the functor is computable and the partial order is decidable, which is certainly true if $\mathcal{A} = \mathcal{B}^n$. The emptiness check is somewhat more involved, since we have to take care of "illegal" annotations.

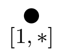
Proposition 8.19 (Emptiness check for languages specified by multiply annotated graphs)**.** For a graph language $\mathcal{L}(T[M])$ characterized by a multiply annotated type graph $T[M]$ over $\mathcal{B}^n$ the emptiness problem is decidable: $\mathcal{L}(T[M]) = \emptyset$ iff $M = \emptyset$ or for each $(\ell, u) \in M$ there exists an edge $e \in E_T$ such that $\ell(e) \geq 1$ and $(u(\mathit{src}(e)) = 0$ or $u(\mathit{tgt}(e)) = 0)$.

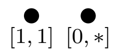
Language inclusion can be deduced from the existence of a legal morphism between the two multiply annotated type graphs.

Proposition 8.20 (Language inclusion and legal morphisms)**.** The existence of a legal morphism $\varphi\colon T_1[M] \to T_2[N]$ implies $\mathcal{L}(T_1[M]) \subseteq \mathcal{L}(T_2[N])$.

Example 8.21. *Consider the annotated type graphs $T_1[\ell, u]$ and $T_2[\ell', u']$ from Example 8.16. Since the more restrictive annotated type graph $T_2[\ell', u']$ models the same scenario as $T_1[\ell, u]$, the two languages should be included into each other. Indeed, by Proposition 8.20 it holds that $\mathcal{L}(T_2[\ell', u']) \subseteq \mathcal{L}(T_1[\ell, u])$, since we can easily find a legal graph morphism $\varphi\colon T_2[\ell', u'] \to T_1[\ell, u]$.*

Please note, that this condition is sufficient but not necessary, as shown by the following counterexample. Let the following two multiply annotated type graphs $T_1[M_1]$ and $T_2[M_2]$ over $\mathcal{B}^1$ be given where $|M_1| = |M_2| = 1$:

$T_1[M_1] =$ 

$T_2[M_2] =$ 

Clearly we have that the languages $\mathcal{L}(T_1[M_1])$ and $\mathcal{L}(T_2[M_2])$ are equal as both contain all discrete non-empty graphs. Thus $\mathcal{L}(T_1[M_1]) \subseteq \mathcal{L}(T_2[M_2])$, but there exists no legal morphism $\varphi\colon T_1[M_1] \to T_2[M_2]$. In fact, the upper bound of the first node of T_2 would be violated if the node of T_1 is mapped by φ to it, while the lower bound would be violated if the node of T_1 is mapped to the other node.

8.3.3. Deciding Language Inclusion for Annotated Type Graphs

In this section we show that if we allow only bounded graph languages consisting of graphs up to a fixed *pathwidth*, the language inclusion problem becomes decidable for annotations based on $\mathcal{B}^n$. Pathwidth is a well-known concept from graph theory that intuitively measures how closely a graph resembles a path. A more formal definition is given below. We start by giving the notion of *tree decompositions* and *path decompositions* which were introduced in [RS83; RS86].

Definition 8.22 (Tree and path decomposition)**.** Let $G = (V, E, src, tgt, lab)$ be a graph. A *tree decomposition* of G is a pair $\mathsf{T}_G = \langle T, X \rangle$, where T is a tree and $X = \{X_{t_1}, \ldots, X_{t_n}\}$ is a family of subsets of V, called *bags*, indexed by the nodes of T, such that:

- for each node $v \in V$, there exists a node t of T such that $v \in X_t$;
- for each edge $e \in E$, there is a node t of T such that all nodes v attached to e are in X_t;
- for each node $v \in V$, the graph induced by the nodes $\{t \mid v \in X_t\}$ is a subtree of T.

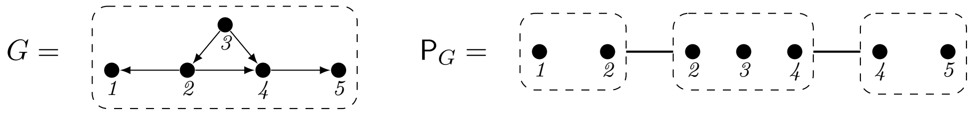
A *path decomposition* $\mathsf{P}_G = \langle T, X \rangle$ of a graph G is a tree decomposition where T is a path.

The treewidth and the pathwidth of a graph can be used to measure how similar the graph is to a tree or to a path. Please note that in the following definition we always decrement the *width* of a tree decomposition by 1, to ensure that trees have treewidth 1, paths have pathwidth 1 and discrete graphs have treewidth and pathwidth 0.

Definition 8.23 (Width, treewidth and pathwidth)**.** The *width* of a tree decomposition $\mathsf{T}_G = \langle T, X \rangle$ is $w(\mathsf{T}_G) = (\max_{t \in T} |X_t|) - 1$. The *pathwidth* $pw(G)$ and the *treewidth* $tw(G)$ of a graph G are defined as follows:

- $pw(G) = \min\{w(\mathsf{P}_G) \mid \mathsf{P}_G \text{ is a path decomposition of } G\}$,
- $tw(G) = \min\{w(\mathsf{T}_G) \mid \mathsf{T}_G \text{ is a tree decomposition of } G\}$.

Example 8.24. *Consider the graph G shown below to the left. Please note that the graph G contains a 3-clique. One possible decomposition of G is the path decomposition P_G shown below to the right which has width $w(\mathsf{P}_G) = 2$, due to the fact that the largest bag consists of three nodes.*



For all possible path decompositions this width is the smallest possible since the three nodes of the 3-clique always have to be in the same bag. Therefore, the graph G has the pathwidth $pw(G) = 2$.

The following proof for the decidability of the language inclusion problem is based on the notion of recognizability, which will be described via automaton functors that were introduced in [BK08]. We start with the main result and explain step by step the arguments that will lead to decidability.

Proposition 8.25 (Language inclusion and bounded pathwidth)**.** The language inclusion problem is decidable for graph languages of bounded pathwidth characterized by multiply annotated type graphs over $\mathcal{B}^n$. That is, given $k \in \mathbb{N}$ and multiply annotated type graphs $T_1[M_1]$ and $T_2[M_2]$, it is decidable whether $\mathcal{L}(T_1[M_1])^{\leq k} \subseteq \mathcal{L}(T_2[M_2])^{\leq k}$, with $\mathcal{L}(T[M])^{\leq k} = \{G \in \mathcal{L}(T[M]) \mid pw(G) \leq\ k\}$.

Our automaton model, given by automaton functors, reads cospans (i.e., graphs with interfaces) instead of single graphs. Therefore in the following, the category under consideration will be $Cospan_m(\mathbf{Graph}_\Lambda)$, i.e. the category of cospans of graphs where the objects are discrete graphs J, K and the arrows are cospans $c\colon J \to G \leftarrow K$ where both graph morphisms are injective [BK08]. We will refer to the graph J as the *inner interface* and to the graph K as the *outer interface* of the cospan c. Although in general interfaces could also be non-discrete, discrete graphs suffice in order to glue graphs and hence we restrict to discrete interfaces. We will sometimes abbreviate the cospan $c\colon J \to G \leftarrow K$ to the short representation $c\colon J \mathrel{\circ\!\!\rightarrow} K$. Composition of two cospans c_1, c_2 where the outer interface of c_1 matches the inner interface of c_2 is done via pushouts and denoted by $c_1; c_2$.

Definition 8.26 (Composition of cospans)**.** Let $c_1\colon J \xrightarrow{\psi_L} G \xleftarrow{\psi_R} I$ and $c_2\colon I \xrightarrow{\psi'_L} G' \xleftarrow{\psi'_R} K$ be two cospans in the category $Cospan_m(\mathbf{Graph}_\Lambda)$. The cospan composition $c_1; c_2\colon J \xrightarrow{f\circ\psi_L} H \xleftarrow{g\circ\psi'_R} K$ is defined by the commuting diagram shown to the right, where the middle square is a pushout.

$$\begin{tikzcd}
 & & I \arrow[dl, "\psi_R"'] \arrow[dr, "\psi'_L"] & & \\
J \arrow[r, "\psi_L"] \arrow[drr, dashed] & G \arrow[dr, "f"] & (PO) & G' \arrow[dl, "g"'] & K \arrow[l, "\psi'_R"'] \arrow[dll, dashed] \\
 & & H & &
\end{tikzcd}$$

According to [BB+13] a graph has pathwidth k iff it can be decomposed into cospans where each middle graph of a cospan has at most $k + 1$ nodes.

Our main goal is to build an automaton which can read all graphs of our language step by step, similar to the idea of finite automata reading words in formal languages. Since graphs are per se monolithic, we use cospan (de)composition as a means to split graphs into smaller entities. In particular, graph automata read cospans in place of alphabet symbols. Such an automaton can be constructed for an unbounded language, where the pathwidth is not restricted. However, we obtain a *finite* automaton only if we restrict the pathwidth.

Then we can use well-known algorithms for finite automata to solve the language inclusion problem. Note that, if we would use tree automata instead of finite automata, our result could be generalized to graphs of bounded *treewidth*.

We will first introduce the notion of automaton functor (which is a categorical automaton model for so-called recognizable arrow languages) and which is inspired by Courcelle's theory of recognizable graph languages [CE12].

Definition 8.27 (Automaton functor [BK08])**.** An automaton functor $\mathcal{C}$ is a functor $\mathcal{C}\colon \mathit{Cospan}_m(\mathbf{Graph}_\Lambda) \to \mathbf{Rel}$ that maps every object J (i.e., every discrete graph) to a finite set $\mathcal{C}(J)$ (the set of states of J) and every cospan $c\colon J \nrightarrow K$ to a relation $\mathcal{C}(c) \subseteq \mathcal{C}(J) \times \mathcal{C}(K)$ (the transition relation of c). In addition there is a distinguished set of initial states $I \subseteq \mathcal{C}(\varnothing)$ and a distinguished set of final states $F \subseteq \mathcal{C}(\varnothing)$. The *language* $\mathcal{L}_\mathcal{C}$ of $\mathcal{C}$ is defined as follows:

> A graph G is contained in $\mathcal{L}_\mathcal{C}$ if and only if there exist states $q \in I$ and $q' \in F$ which are related by $\mathcal{C}(c)$, i.e. $(q, q') \in \mathcal{C}(c)$, where $c\colon \varnothing \to G \leftarrow \varnothing$ is the unique cospan with empty interfaces and middle graph G.

Languages accepted by automaton functors are called *recognizable.*

As a warm-up let us show an example of an automaton functor, which accepts the pure type graph language determined by a graph T. This graph language has been considered before in [BK08].

Example 8.28. *Let T be a fixed type graph and let $\mathcal{L}(T)$ be its type graph language, i.e. the set of all graphs G for which a morphism $f\colon G \to T$ exists. We show that $\mathcal{L}(T)$ is recognizable, as it is accepted by the automaton functor $\mathcal{C}_T$ defined as follows. The automaton functor $\mathcal{C}_T$ maps every discrete graph J to the set of states $\mathcal{C}_T(J)$ consisting of all morphisms $f_J\colon J \to T$. For a cospan $c\colon J \to G \leftarrow K$, $\mathcal{C}_T(c)$ is the relation on $\mathcal{C}(J) \times \mathcal{C}(K)$ that relates a morphism $f_J\colon J \to\ T$ to a morphism $f_K\colon K \to T$ (i.e. $(f_J, f_K) \in\ \mathcal{C}_T(c)$) whenever there exists a morphism $f\colon G \to T$ such that $f \circ \psi_\mathrm{L} = f_J$ and $f \circ \psi_\mathrm{R} = f_K$ (see the diagram to the right).*

$$\begin{tikzcd}
J \arrow[r, "\psi_\mathrm{L}"] \arrow[dr, "f_J"'] & G \arrow[d, dashed, "f"] & K \arrow[l, "\psi_\mathrm{R}"'] \arrow[dl, "f_K"] \\
& T &
\end{tikzcd}$$

Intuitively, a graph is read step-by-step and we always record the current state via a morphism $J \to T$. Then, whenever we see a new slice G' of our graph (with inner interface J and outer interface K) we check locally whether we can extend the map into T, resulting in $K \to T$ as the new state.

The initial and final states are defined as $I = F = \mathcal{C}_T(\varnothing) = \{f_\varnothing\}$, where $f_\varnothing\colon \varnothing \to T$ is the unique morphism from the empty graph to T.

It follows immediately from the definition that $\mathcal{C}_T$ accepts $\mathcal{L}(T)$. In fact, a graph G is accepted by $\mathcal{C}_T$ iff only the states $f_\varnothing \in I$ and $f_\varnothing \in F$ are related by $\mathcal{C}_T(c)$, where $c\colon \varnothing \to G \leftarrow \varnothing$. But this holds if and only if there is a morphism $f\colon G \to T$, as the two additional conditions required above ($f \circ \psi_\mathrm{L} = f_\varnothing$ and $f \circ \psi_\mathrm{R} = f_\varnothing$) trivially hold.

As a proof obligation, it would remain to show that $\mathcal{C}_T$ is well defined as a functor from $\mathrm{Cospan}_m(\mathbf{Graph}_\Lambda)$ *to* **Rel***, i.e. it preserves identities and arrow composition. This follows easily from the proof of Proposition 8.31 presented below, where a more complex functor is considered.*

We will now define an automaton functor for a type graph $T[M]$ over $\mathcal{B}^n$. The following construction is an extension of Example 8.28, where we count nodes and edges in addition.

Definition 8.29 (Counting cospan automaton)**.** Let $T[M]$ be a multiply annotated type graph over $\mathcal{B}^n$. We define an automaton functor $\mathcal{C}_{T[M]}$ (for $T[M]$) with $\mathcal{C}_{T[M]}\colon \mathit{Cospan}_m(\mathbf{Graph}_\Lambda) \to \mathbf{Rel}$ as follows:

- For each object J of $\mathit{Cospan}_m(\mathbf{Graph}_\Lambda)$ (thus J is a finite discrete graph), $\mathcal{C}_{T[M]}(J) = \{(f, b) \mid f\colon J \to T, b \in \mathcal{B}^n(T)\}$ is its finite set of states
- $I \subseteq \mathcal{C}_{T[M]}(\varnothing)$ is the set of *initial* states with $I = \{(f\colon \varnothing \to T, 0)\}$, where 0 is the constant 0-function
- $F \subseteq \mathcal{C}_{T[M]}(\varnothing)$ is the set of *final* states with $F = \{(f\colon \varnothing \to T, b) \mid \exists(\ell, u) \in M : \ell \leq b \leq u\}$

Let $c\colon J \xrightarrow{\psi_L} G \xleftarrow{\psi_R} K$ be an arrow in the category $\mathit{Cospan}_m(\mathbf{Graph}_\Lambda)$ with discrete interface graphs J and K where both graph morphisms $\psi_L\colon J \to G$ and $\psi_R\colon K \to G$ are injective. Two states $(f\colon J \to T, b)$ and $(f'\colon K \to T, b')$ are in the relation $\mathcal{C}_{T[M]}(c)$ if and only if there exists a morphism $h\colon G \to T$ such that the diagram to the right commutes and for all $x \in V_T \cup E_T$ the following equation holds:

$$\overbrace{\hphantom{J \xrightarrow{\psi_L} G \xleftarrow{\psi_R} K}}^{c\colon J \nrightarrow K}$$

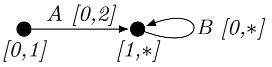
$$\begin{tikzcd}
J \arrow[r, "\psi_L"] \arrow[dr, "f"'] & G \arrow[d, dashed, "\exists h"] & K \arrow[l, "\psi_R"'] \arrow[dl, "f'"] \\
 & T &
\end{tikzcd}$$

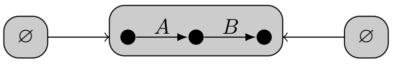
$$b'(x) = b(x) + |\{y \in (G \setminus \psi_R(K)) \mid h(y) = x\}|$$

The set $G \setminus \psi_R(K)$ consists of all elements of G which are not targeted by the morphism ψ_R, i.e. $G \setminus \psi_R(K) = (V_G \setminus \psi_R(V_K)) \cup (E_G \setminus \psi_R(E_K))$.

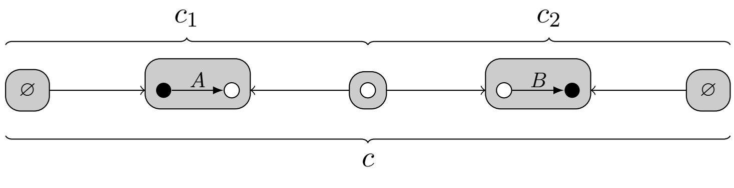
Instead of $\mathcal{L}_{\mathcal{C}_{T[M]}}$ and $\mathcal{C}_{T[M]}$ we just write $\mathcal{L}_\mathcal{C}$ and $\mathcal{C}$ if $T[M]$ is clear from the context. The intuition behind the construction is to count for each item x of T, step by step, the number of elements that are being mapped from a graph G (which is in the form of a cospan decomposition) to x, and then check if the bounds of a pair of annotations $(\ell, u) \in M$ of the multiply annotated type graph $T[M]$ are satisfied. We give a short example before moving on to the results.

Example 8.30. *Let the following multiply annotated type graph (over $\mathcal{B}^2$) $T[\ell, u]$ and the cospan $(c\colon \varnothing \to G \leftarrow \varnothing)$ with $G \in \mathcal{L}(T[\ell, u])$ be given:*

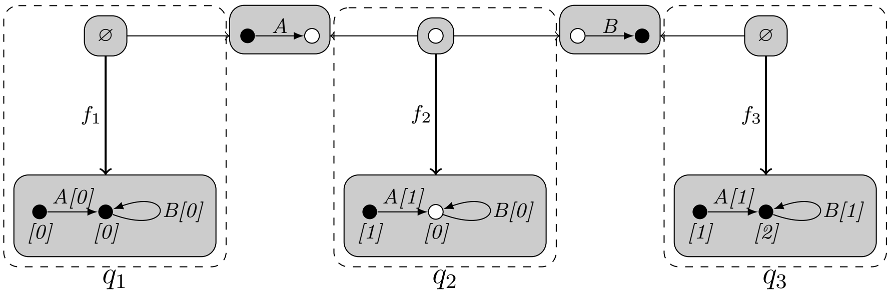
$T[\ell, u] =$ 

$c\colon$ 

We will now decompose the cospan c into two cospans c_1, c_2 with $c = c_1; c_2$ in the following way:



We let our counting cospan automaton parse the cospan decomposition $c_1; c_2$ step by step now to show how the annotations for the type graph T evolve during the *process. According to our construction, every element in T has multiplicity 0 in the initial state of the automaton. We then sum up the number of elements within the middle graphs of the cospans which are* not *part of the right interface. Therefore we get the following parsing process:*



We visited three states q_1, q_2 and q_3 in the automaton with $(q_1, q_2) \in \mathcal{C}(c_1)$ and $(q_2, q_3) \in \mathcal{C}(c_2)$. Since $\mathcal{C}$ is supposed to be a functor we get that $\mathcal{C}(c_1);\mathcal{C}(c_2) = \mathcal{C}(c)$ and therefore $(q_1, q_3) \in \mathcal{C}(c)$ also holds. In addition we have $q_1 \in I$ and since the annotation function $b \in \mathcal{B}^2(T)$ in $q_3 = (f_3, b)$ satisfies $\ell \leq b \leq u$ we can infer that $q_3 \in F$. Therefore we can conclude that $G \in \mathcal{L}_\mathcal{C}$ holds as well.

We still need to prove that $\mathcal{C}$ is indeed a functor. Intuitively this shows that acceptance of a graph by the automaton is not dependent on its specific decomposition.

Proposition 8.31 (Functoriality of $\mathcal{C}$)**.** Let two cospans $c_1 \colon J \to G \leftarrow K$ and $c_2 \colon K \to H \leftarrow L$ be given and let $id_G \colon G \to G \leftarrow G$ be the identity cospan.

The mapping $\mathcal{C}_{T[M]} \colon \mathit{Cospan}_m(\mathbf{Graph}_\Lambda) \to \mathbf{Rel}$ is a functor, i.e.

1. $\mathcal{C}_{T[M]}(id_G) = id_{\mathcal{C}_{T[M]}(G)}$
2. $\mathcal{C}_{T[M]}(c_1; c_2) = \mathcal{C}_{T[M]}(c_1);\mathcal{C}_{T[M]}(c_2)$

The language accepted by the automaton $\mathcal{L}_\mathcal{C}$ is exactly the graph language $\mathcal{L}(T[M])$.

Proposition 8.32 (Language of $\mathcal{C}$)**.** Let the multiply annotated type graph $T[M]$ (over $\mathcal{B}^n$) and the automaton functor $\mathcal{C} \colon \mathit{Cospan}_m(\mathbf{Graph}_\Lambda) \to \mathbf{Rel}$ for $T[M]$ be given. Then $\mathcal{L}_\mathcal{C} = \mathcal{L}(T[M])$ holds, i.e. for a graph G we have $G \in \mathcal{L}(T[M])$ if and only if there exist states $i \in I \subseteq \mathcal{C}(\varnothing)$ and $f \in F \subseteq \mathcal{C}(\varnothing)$ such that $(i, f) \in \mathcal{C}(c)$, where $c \colon \varnothing \to G \leftarrow \varnothing$.

Therefore we can construct an automaton for each graph language specified by a multiply annotated type graph $T[M]$, which accepts exactly the same language. In case of a bounded graph language this automaton will have only finitely many states. Furthermore we can restrict the label alphabet, i.e., the cospans by using only atomic cospans, adding a single node or edge (see [BB+12; Blu14]). Once these steps are performed, we obtain conventional non-deterministic finite automata over a finite alphabet and we can use standard techniques from automata theory to solve the language inclusion problem directly on the finite automata.

8.3.4. Closure Properties for Multiply Annotated Graphs

Extending the expressiveness of the type graphs by adding multiplicities gives us positive results in case of closure under union and intersection. Here we use constructions that rely on products and coproducts in the category of graphs. Closure under intersection holds for the most general form of annotations. From $T_1[M_1]$, $T_2[M_2]$ we can construct an annotated type graph $(T_1 \times T_2)[N]$, where N contains all annotations which make both projections $\pi_i \colon T_1 \times T_2 \to T_i$ legal.

Proposition 8.33 (Closure under intersection)**.** Graph languages specified by annotated graphs are closed under intersection.

We can prove closure under union for the case of annotations based on the functor $\mathcal{B}^n$. Here we take the coproduct $(T_1 \oplus T_2)[N]$, where N contains all annotations of M_1, M_2, transferred to $T_1 \oplus T_2$ via the injections $i_j \colon T_j \to T_1 \oplus T_2$. Intuitively, graph items not in the original domain of the annotations receive annotation $[0, 0]$.

In order to show closure under union for graph languages specified by annotated type graphs over $\mathcal{B}^n$, we depend on two lemmata given in Appendix A.4. Please note that this result can be generalized for annotations over a functor $\mathcal{A}$ whenever they satisfy the mild assumptions stated in Lemma L.3 and Lemma L.5.

Proposition 8.34 (Closure under union)**.** Graph languages specified by multiply annotated graphs over functor $\mathcal{B}^n$ are closed under union.

Finally, we can prove that graph languages specified by annotated type graphs are not closed under complement for the case of annotations based on the functor $\mathcal{B}^n$.

Proposition 8.35 (Non-closure under complement)**.** Graph languages specified by multiply annotated graphs over functor $\mathcal{B}^n$ are not closed under complement.

The main reason for this is, that there exist languages specified by multiply annotated type graphs based on $\mathcal{B}^n$ for which there exists no finite type graph which specifies the complement language. A counterexample is given in the proof found in Appendix A.4.

Conclusion of Part III

Our results from Part III on decidability and closure properties for the investigated specification frameworks are summarized in the following table. In the case where the results hold only for bounded pathwidth, the checkmark is in brackets. The results for the general case are still open.

		Pure TG	Restr. Gr.	TG Logic	Annot. TG
Decidability	$G \in \mathcal{L}$?	✓	✓	✓	✓
	$\mathcal{L} = \emptyset$?	✓	✓	✓	✓
	$\mathcal{L}_1 \subseteq \mathcal{L}_2$?	✓	✓	✓	(✓)
Closure Prop.	$\mathcal{L}_1 \cup \mathcal{L}_2$	✗	✓	✓	✓
	$\mathcal{L}_1 \cap \mathcal{L}_2$	✓	✗	✓	✓
	$\lvert\mathbf{Graph}_\Lambda\rvert \setminus \mathcal{L}$	✗	✗	✓	✗
Invariant Checking (DPO)		✓	✓		

In order to be able to use these formalisms extensively in applications, it is necessary to provide a mechanism to compute weakest preconditions and strongest postconditions. That is, given a graph language and a set of rules, we want to specify the language of all successors (or the language of all predecessors) in our formalism. This is not feasible for pure type graphs or the type graph logic, since neither formalism can count and hence can not express that all items of the newly created right-hand occur exactly once. Hence, in Chapter 10 we characterize strongest postconditions in the setting of annotated type graphs. This requires a materialization construction (discussed in the upcoming Chapter 9), similar to [SRW02], which is characterized abstractly, exploiting universal properties in category theory.

After characterizing the computation of postconditions for annotated type graphs, we can settle the invariant checking (at least for the case of bounded pathwidth) in the framework of multiply annotated type graphs. In Chapter 10, given a graph transformation rule we compute the postcondition of a graph language $\mathcal{L}$ specified by multiply annotated type graphs and afterwards check whether it is included in $\mathcal{L}$ for bounded pathwidth. This gives us a procedure for invariant checking.

Related Work

We review some of the more well-known approaches for specifying graph languages. While it is impossible to elaborate the advantages and disadvantages of each specification formalism in detail (some of these are current research questions) we will highlight some interesting points.

Note that our approach has a limited expressiveness compared to many other approaches, which is also witnessed by the fact that annotated type graphs can be encoded into graph automata. On the other hand, we have several closure properties and positive decidability results, which makes the formalisms interesting for verification.

Recognizable graph languages [Cou90; CE12], which are the counterpart to regular word languages, are closely related with monadic second-order graph logic.

If one restricts recognizable graph languages to bounded treewidth (or pathwidth as we did), one obtains satisfactory decidability properties. On the other hand, the size of the resulting graph automata is often quite intimidating [BB+12; Blu14] and hence they are difficult to work with in practical applications. The use of nested application conditions [HP05], equivalent to first-order logic [Ren04b], has a long tradition in graph rewriting and they can be used to compute pre- and postconditions for rules [Pen09]. However, satisfiability and implication are undecidable for first-order logic.

A notion of grammars corresponding to context-free (word) grammars are hyperedge replacement grammars [Hab92]. Many aspects of the theory of context-free languages can be transferred to the graph setting.

In heap analysis the representation of pointer structures to be analyzed requires methods to specify sets of graphs. Hence both the TVLA approach by Sagiv, Reps and Wilhelm [SRW02], as well as separation logic [OHe07; DOY06] provide formalisms for characterizing graph languages, both based on logic. In [SRW02] heaps are represented by graphs, annotated with predicates from a three-valued logics (with truth values *yes*, *no* and *maybe*).

A further interesting approach are forest automata [AH+13] that have many interesting properties, but are somewhat complex to handle.

In [RR+09] the authors study an approach called *Diagram Predicate Framework* (DPF), in which type graphs have annotations based on generalized sketches. This formalism is intended for modelling languages based on *Meta Object Facility* (MOF) and allows more complex annotations than our framework.

Open Questions

One open question that remains is whether language inclusion for annotated type graphs is decidable if we do not restrict to bounded pathwidth. Additionally, invariant checking for the type graph logic is still open. As discussed in Section 8.3.1, our edge annotations are global, as opposed to the local multiplicities that find use in UML. Local annotations only partially fit into our framework as we will see in Chapter 10, but they are of course quite relevant. Hence, to study the possibility to fully integrate such multiplicities and investigate the corresponding decidability and closure properties is another problem left open.

Part IV.

Abstract Object Rewriting

Motivation of Part IV

Abstract interpretation [Cou96] is a fundamental static analysis technique that applies not only to conventional programs but also to general infinite-state systems. Shape analysis [SRW02], a specific instance of abstract interpretation, pioneered an approach for analyzing pointer structures that keeps track of information about the "heap topology", e.g., out-degrees or existence of certain paths. One central idea of shape analysis is *materialization*, which arises as companion operation to summarizing distinct objects that share relevant properties. Materialization, also known as partial concretization, is also fundamental in verification approaches based on separation logic [CR08; CD+11; LRC15], where it is also known as rearrangement [OHe12], a special case of frame inference. Shape analysis—construed in a wide sense—has been adapted to graph transformation [Roz97]. Motivated by earlier work of shape analysis for graph transformation [SWW11; BW07; Bac15; BR15a; RZ10; Ren04a], we want to put the materialization operation on a new footing, widening the scope of shape analysis.

A natural abstraction mechanism for transition systems with graphs as states "summarizes" all graphs over a specific *shape graph*. Thus a single graph is used as abstraction for all graphs that can be mapped homomorphically into it, similar to the concept of graph languages specified via type graphs (see Chapter 4). Further annotations on shape graphs, such as cardinalities of preimages of its nodes and general first-order formulas, enable fine-tuning of the granularity of abstractions. While these natural abstraction principles have been successfully applied in previous work [SWW11; BW07; Bac15; BR15a; RZ10; Ren04a], their companion materialization constructions are notoriously difficult to develop, hard to understand, and are redrawn from scratch for every single setting. Thus, we set out to explain materializations based on mathematical principles, namely universal properties (in the sense of category theory). In particular, we will use partial map classifiers in the topos of graphs (and its slice categories) which cover the purely structural aspects of materializations; this is related to final pullback complements [DT87], a fundamental construction of graph rewriting [Löw10; CH+06]. Annotations of shape graphs are treated orthogonally via op-fibrations.

Our main goal in Part IV is to define a rewriting formalism for graph abstractions that lifts double-pushout rule-based rewriting from single graphs to multiply annotated type graphs (introduced in Chapter 8). In this context the type graphs specifying the language will be referred to as *abstract graphs.* Abstract graphs cannot be rewritten in the same way as ordinary graphs, since an occurrence of the left-hand side might represent additional structure apart from the match, which cannot simply be deleted. Hence it is necessary to materialize a copy of the left-hand side. It is, in slightly different forms, present in all the approaches cited above. To achieve the goal of defining a general rewriting framework, we do not restrict ourselves to the category $\mathbf{Graph}_\Lambda$, but will introduce both, the materialization construction and the abstract rewriting step, to work for (abstract) objects in an arbitrary topos $\mathbf{C}$.

Outline:

First, in Section 9.1, we introduce topoi and additional preliminaries in the form of categorical constructions such as partial map classifiers, final pullback complements and slice categories. Furthermore, in Section 9.2, we generalise our concept of type graph languages to languages of abstract objects in an arbitrary category. We define a materialization construction categorically in Section 9.3.1, that enables us to concretize an instance of a left-hand side of a production in a given abstract object. This construction is refined in Section 9.3.2 where we restrict to materializations that satisfy the gluing condition and can thus be rewritten via the production. In Section 9.3.3 we present the main result about materializations showing that we can fully characterize the co-matches obtained by rewriting. Afterwards, in Section 10.1 we extend abstract rewriting to abstract objects enriched with the annotations from Chapter 8. In Section 10.2 we define properties for annotations which need to be satisfied in order to achieve soundness and completeness of our abstract rewriting steps. Finally, in Section 10.3 we show that abstract rewriting with annotations is sound and, with additional assumptions regarding the properties, complete. We furthermore derive strongest post-conditions for the case of graph rewriting with annotations. All proofs can be found in Appendix A.5 and A.6.

> *“The inventor...looks upon the world and is not contented with things as they are. He wants to improve whatever he sees, he wants to benefit the world; he is haunted by an idea. The spirit of invention possesses him, seeking materialization.”*
>
> Alexander Graham Bell (1847-1922)

9 Materialization Category

In this chapter, we characterize a materialization operation for so-called *abstract objects* in a topos in terms of partial map classifiers, which gives us a sound and complete description of all occurrences of right-hand sides of rules obtained by rewriting an abstract object. The materialization yields an object which specifies a language consisting of all objects with a concrete instance of a left-hand side from a production.

9.1. Additional Preliminaries - More Categorical Concepts

Since this chapter presupposes familiarity with the *topos* structure of graphs and several additional concepts from category theory (in particular elementary topoi, subobject and partial map classifiers, final pullback complements and slice categories) we start with some definitions and results related to elementary topoi.

9.1.1. Topoi, Subobject Classifiers and Partial Map Classifiers

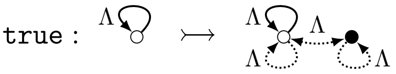
First, we define the categorical concept of *subobject classifiers*. Intuitively, a subobject classifier is a morphism which maps the subobjects of an object X to truth values, i.e. assigning “true“ to the elements of the subobject to be classified and “false“ to all other elements of X. Hence, a subobject classifier in a category $\mathbf{C}$ maps objects into a so-called *truth value object*. The structure of a truth value object depends on the category where the subobject classifier is defined.

Definition 9.1 (Subobject classifier)**.** Let $\mathbf{C}$ be a category where $\mathbf{1}$ is the terminal object and for each object $X \in \mathbf{C}$ let $!_X \colon X \to \mathbf{1}$ be the unique arrow from X into the terminal object. A mono $\mathtt{true}\colon \mathbf{1} \rightarrowtail \Omega$ is a *subobject classifier* if for every mono $i\colon X \rightarrowtail Y$ in $\mathbf{C}$ there exists a unique arrow $\chi_i\colon Y \to \Omega$ such that the diagram to the right is a pullback. In this case object Ω is called the *truth value object*.

$$\begin{tikzcd}
X \arrow[r, tail, "i"] \arrow[d, "!_X"'] \arrow[dr, phantom, "\text{(PB)}"] & Y \arrow[d, dashed, "\chi_i"] \\
\mathbf{1} \arrow[r, tail, "\mathtt{true}"'] & \Omega
\end{tikzcd}$$

Example 9.2. *In* **Set** *the subobject classifier* $\mathtt{true}\colon \mathbf{1} \rightarrowtail \Omega$ *is simply the embedding of* $\{1\}$ *into the two-element set* $\Omega = \{0,1\}$*. A subset* $X \subseteq Y$ *can be characterized via its characteristic function* $\chi_X\colon Y \to \{0,1\}$*.*

Example 9.3. *In the category* $\mathbf{Graph}_\Lambda$*, where the objects are labelled graphs over the label alphabet* Λ*, the subobject classifier* `true` *is shown to the right where every* Λ*-labelled edge represents several edges, one for each* $\lambda \in \Lambda$*. The subobject classifier* `true` *from the terminal object* $\mathbf{1}$ *to* Ω *allows us to single out a subgraph* X *of a graph* Y*, by mapping* Y *to* Ω *in such a way that all elements of* X *are mapped to the image of* `true`*.*



The category of graphs $\mathbf{Graph}_\Lambda$ is an *elementary topos*, which is an extremely rich categorical structure. The notion of elementary topoi [Joh02; Law70] is used in logic and it abstracts from the structure of the category of sets.

Definition 9.4 (Elementary topos)**.** An elementary topos is a category which has finite limits, is cartesian closed and has a subobject classifier.

We will often omit the qualifier "elementary" and simply talk about topoi.

In addition we will use the notion of *natural transformations*, which provides a way of transforming a functor into another functor while respecting the internal structure of the categories involved. Therefore, similar to functors being the abstract notion of morphisms between categories, natural transformations are the abstract notion of morphisms between functors.

Definition 9.5 (Natural transformation)**.** Let $\mathbf{C}, \mathbf{D}$ be two categories and let $\mathcal{F}$ and $\mathcal{G}$ be functors from $\mathbf{C}$ to $\mathbf{D}$. A natural transformation $\eta\colon \mathcal{F} \stackrel{\cdot}{\to} \mathcal{G}$ is a family of arrows from $\mathcal{F}$ to $\mathcal{G}$ that assigns to every $\mathbf{C}$-object X a $\mathbf{D}$-arrow $\eta_X\colon \mathcal{F}(X) \to \mathcal{G}(X)$ such that for any $\mathbf{C}$-arrow $f\colon X \to Y$ the diagram to the right commutes in $\mathbf{D}$. The morphism η_X is also called *component* of η on X.

$$\begin{tikzcd}
\mathcal{F}(X) \arrow[r, "\eta_X"] \arrow[d, "\mathcal{F}(f)"'] & \mathcal{G}(X) \arrow[d, "\mathcal{G}(f)"] \\
\mathcal{F}(Y) \arrow[r, "\eta_Y"] & \mathcal{G}(Y)
\end{tikzcd}$$

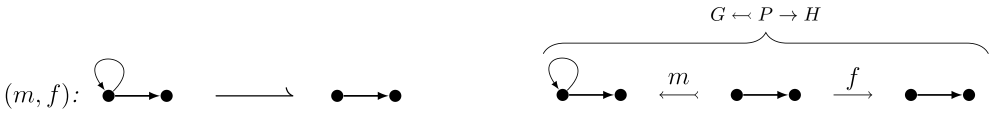
Every elementary topos has so-called *partial map classifiers* [CL03]. Informally, a partial map classifier for a partial map $(m, f)\colon X \rightharpoonup Y$ works similar to a subobject classifier, with the difference, that neither the truth value object nor the domain of the classifier morphism are fixed for the category. Instead, both need to be constructed from the codomain object Y of the partial map, by constructing an object $\mathcal{F}(Y)$ via a functor $\mathcal{F}$ and describing the morphism between Y and $\mathcal{F}(Y)$ via the monic component $\eta_Y\colon Y \rightarrowtail \mathcal{F}(Y)$ of a natural transformation η.

Definition 9.6 (Partial map classifier)**.** Let $\mathbf{C}$ be a category with pullbacks. A partial map $(m, f)\colon X \rightharpoonup Y$ in $\mathbf{C}$ is a span $X \overset{m}{\leftarrowtail} Z \overset{f}{\to} Y$ where $m\colon Z \rightarrowtail X$ is a mono. A *partial map classifier* $(\mathcal{F}, \eta)$ is a functor $\mathcal{F}\colon \mathbf{C} \to \mathbf{C}$ together with a natural transformation $\eta\colon Id_{\mathbf{C}} \stackrel{\cdot}{\to} \mathcal{F}$ such that for each object Y of $\mathbf{C}$ with the component $\eta_Y\colon Y \rightarrowtail \mathcal{F}(Y)$ the following holds: for each partial map $(m, f)\colon X \rightharpoonup Y$ there exists a unique arrow $\varphi(m, f)\colon X \to \mathcal{F}(Y)$ such that the diagram to the right is a pullback.

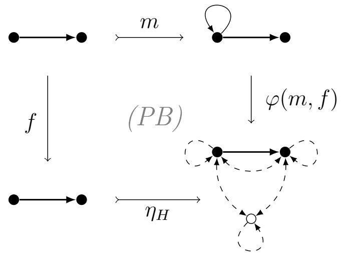
$$\begin{tikzcd}
Z \arrow[r, tail, "m"] \arrow[d, "f"'] \arrow[dr, phantom, "\text{(PB)}"] & X \arrow[d, dashed, "\varphi(m,f)"] \\
Y \arrow[r, tail, "\eta_Y"] & \mathcal{F}(Y)
\end{tikzcd}$$

Example 9.7. *In* **Set** *the functor $\mathcal{F}$ enriches each set Y with an additional element $\star$, i.e., $\mathcal{F}(Y) = Y + \{\star\}$. Then a partial map $p\colon X \rightharpoonup Y$ corresponds to a total map $p'\colon X \to \mathcal{F}(Y)$ such $p'(x) = p(x)$ if $p(x)$ is defined and $p'(x) = \star$ otherwise.*

Example 9.8. *We now consider a more involved example in the category* $\mathbf{Graph}_\Lambda$*. Let the partial map $(m, f)\colon G \rightharpoonup H$ (depicted below left) and a corresponding span $G \overset{m}{\leftarrowtail} P \overset{f}{\to} H$ (depicted below on the right) be given. We use a single edge label, which is omitted. All graph morphisms are indicated by black and white nodes and the thickness of the edges.*



The partial map classifier object $\mathcal{F}(H)$ alongside the component of the natural transformation $\eta_H\colon H \rightarrowtail \mathcal{F}(H)$ is depicted below right. Intuitively, the functor $\mathcal{F}$ enriches the codomain H of the partial map $(m, f)\colon G \rightharpoonup H$ by an additional structure which can be interpreted as the "false" classification of a subobject classifier. The natural transformation component η_H maps the graph H to the "true" part of $\mathcal{F}(H)$. As a result the resulting morphism $\varphi(m, f)$ of the pullback diagram classifies every element in G which is defined in the partial map to be "true" and "false" otherwise.



9.1.2. Slice Categories and Final Pullback Complements

The objects in the materialization category that we will introduce in Chapter 9.3, are materializations of the left-hand side of a production over a fixed abstract object A. We can use *slice categories* in connection with subobject classifiers to specify this behaviour. The slice category $\mathbf{C} \downarrow A$, of a category $\mathbf{C}$ over an object A, has morphisms with codomain A as objects and commutative triangles as morphisms.

Definition 9.9 (Slice category)**.** The slice category $\mathbf{C} \downarrow A$ of a category $\mathbf{C}$ over an object $A \in \mathbf{C}$ has the arrows $f \in \mathbf{C}$ such that $cod(f) = A$ as objects. An arrow $g\colon f \to f'$ in $\mathbf{C} \downarrow A$, with $f\colon X \to A$ and $f'\colon Y \to A$, is an arrow $g\colon X \to Y \in \mathbf{C}$ such that the diagram to the right commutes.

$$\begin{tikzcd}
X \arrow[rr, "g"] \arrow[dr, "f"'] & & Y \arrow[dl, "f'"] \\
& A &
\end{tikzcd}$$

Please note that the terminal object of a slice category $\mathbf{C} \downarrow A$ is the identity arrow $id_A\colon A \rightarrowtail A$. The existence of a subobject classifier in a slice category over a topos directly follows from the following theorem [MM94].

Theorem 9.10 (Slice category over a topos [MM94])**.** For any object A in a topos $\mathbf{C}$, the slice category $\mathbf{C} \downarrow A$ is also a topos.

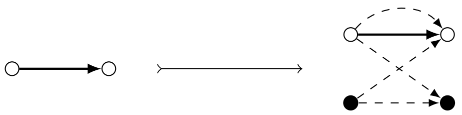
The subobject classifier in the slice category can be constructed as follows.

Fact 9.11 (Subobject classifier in slice category [MM94])**.** Let $\mathbf{C}$ be a topos with subobject classifier $\mathtt{true}\colon \mathbf{1} \rightarrowtail \Omega$ and truth value object Ω. For any object $A \in \mathbf{C}$ let $A \times \Omega$ be the product with projections $\pi_1\colon A \times \Omega \to A$ and $\pi_2\colon A \times \Omega \to \Omega$. Then a subobject classifier $\mathtt{true}_A$ of the slice category $\mathbf{C} \downarrow A$ is the unique mono $\mathtt{true}_A\colon A \rightarrowtail A \times \Omega$ such that the diagram to the right commutes.

$$\begin{tikzcd}
 & A \arrow[d, tail, "\mathtt{true}_A"] \arrow[ddl, bend right, "id_A"'] \arrow[ddr, bend left, "\mathtt{true}\circ !"] & \\
 & A \times \Omega \arrow[dl, "\pi_1"] \arrow[dr, "\pi_2"'] & \\
A & & \Omega
\end{tikzcd}$$

Example 9.12. *In order to provide an example for a subobject classifier in a slice category, we consider again the category* $\mathbf{Graph}_\Lambda$*, with a single edge label which is omitted. Let* $A =$ ○⟶○ *be the base graph for the slice category* $\mathbf{Graph}_\Lambda \downarrow A$ *of graph morphisms into* A*. The subobject classifier* $\mathtt{true}_A\colon A \rightarrowtail A \times \Omega$ *for this slice category is the following graph morphism:*

$\mathtt{true}_A$*:*



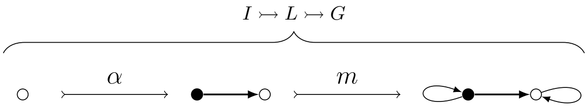
Given arrows α, m as in the diagram below, we can construct the most general pullback, called *final pullback complement* [DT87; CH+06].

Definition 9.13 (Final pullback complement)**.** A pair of arrows $I \xrightarrow{\gamma} F \xrightarrow{\beta} G$ is a *final pullback complement (FPBC)* of another pair $I \xrightarrow{\alpha} L \xrightarrow{m} G$ if

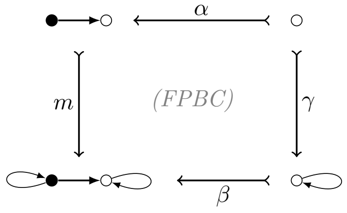
- they form a pullback square
- for each pullback $G \xleftarrow{m} L \xleftarrow{\alpha'} I' \xrightarrow{\gamma'} F' \xrightarrow{\beta'} G$ and arrow $f\colon I' \to I$ such that $\alpha \circ f = \alpha'$, there exists a unique arrow $f'\colon F' \to F$ such that $\beta \circ f' = \beta'$ and $\gamma \circ f = f' \circ \gamma'$ both hold (see the diagram to the right).

$$\begin{tikzcd}
L \arrow[d, "m"'] \arrow[dr, phantom, "\text{(FPBC)}"] & I \arrow[l, "\alpha"] \arrow[d, "\gamma"] & I' \arrow[l, "f"] \arrow[ll, bend right, "\alpha'"'] \arrow[d, "\gamma'"] \\
G & F \arrow[l, "\beta"'] & F' \arrow[l, dashed, "f'"'] \arrow[ll, bend left, "\beta'"]
\end{tikzcd}$$

Example 9.14. *Consider the following pair of graph morphisms* $I \overset{\alpha}{\rightarrowtail} L \overset{m}{\rightarrowtail} G$ *in the category* $\mathbf{Graph}_\Lambda$*:*



The final pullback complement $I \xrightarrow{\gamma} F \xrightarrow{\beta} G$ *is depicted in the diagram to the right. Please note that for the pair of graph morphisms* $I \overset{\alpha}{\rightarrowtail} L \overset{m}{\rightarrowtail} G$ *there does not exist a pushout complement due to the dangling edge condition. However, for the final pullback complement construction, all edges attached to the left black node in* G *are removed.*



Final pullback complements and subobject classifiers are closely related to partial map classifiers (see Definition 9.6 and [DT87, Corollary 4.6]): a category has FPBCs (over monos) and a subobject classifier if and only if it has a partial map classifier. These exist in all elementary topoi.

Proposition 9.15 (Final pullback complements, subobject and partial map classifiers)**.** Let $\mathbf{C}$ be a category with finite limits. Then the following are equivalent:

(1) $\mathbf{C}$ has a subobject classifier $\mathtt{true}\colon \mathbf{1} \rightarrowtail \Omega$ and final pullback complements for each pair of arrows $I \overset{\alpha}{\to} L \overset{m}{\rightarrowtail} G$ where m is a mono;

(2) $\mathbf{C}$ has a partial map classifier $(\mathcal{F} : \mathbf{C} \to \mathbf{C}, \eta : Id_{\mathbf{C}} \stackrel{\cdot}{\to} \mathcal{F})$.

In the next section, we will generalize the concepts of type graph languages to languages of arbitrary objects in a category $\mathbf{C}$.

9.2. Object Languages

The main theme of Part IV is "simultaneous" rewriting of entire sets of objects of a category by means of rewriting a single *abstract* object that represents a collection of structures—the *language* of the abstract object. The simplest example of an abstract structure is a plain object of a category to which we associate the language of objects that can be mapped to it.

Definition 9.16 (Language of an object)**.** Let A be an object of a category $\mathbf{C}$. Given another object X, we write $X \dashrightarrow A$ whenever there exists an arrow from X to A. We define the *language*[1]of A, denoted by $\mathcal{L}(A)$, as the set $\mathcal{L}(A) = \{X \in \mathbf{C} \mid X \dashrightarrow A\}$.

Whenever $X \in \mathcal{L}(A)$ holds we will say that X is *abstracted by* A, and A is called the *abstract object*. Type graphs are an instance of abstract objects alongside their corresponding language (cf. Chapter 4). Therefore, we will sometimes refer to type graphs as *abstract graphs*. We will also need to characterize a class of (co-)matches which are represented by a given (mono) (co-)match.

Definition 9.17 (Language of a mono)**.** Let $\varphi\colon L \rightarrowtail A$ be a mono in $\mathbf{C}$. The *language* of φ is the set of monos m with domain L such that φ factors through m and the square on the right is a pullback:

$$\begin{tikzcd}
L \arrow[r, tail, "m"] \arrow[d, tail, "id_L"'] \arrow[dr, phantom, "\text{(PB)}"] & X \arrow[d, dashed, "\exists\psi"] \\
L \arrow[r, tail, "\varphi"'] & A
\end{tikzcd}$$

$$\mathcal{L}(\varphi) \quad = \quad \{m\colon L \rightarrowtail X \mid X \dashrightarrow A, \\ \text{such that the square to the above right is a pullback}\}.$$

Intuitively, for any arrow $(L \overset{m}{\rightarrowtail} X) \in \mathcal{L}(\varphi)$ we have $X \in \mathcal{L}(A)$ and X has a distinguished subobject L which corresponds precisely to the subobject $L \rightarrowtail A$. In fact ψ restricts and co-restricts to an isomorphism between the images of L in

[1] Here we assume that $\mathbf{C}$ is essentially small, so that a language can be seen as a set instead of a class of objects.

X and A. Intuitively, for graphs, no nodes or edges in X outside of L are mapped by ψ into the image of L in A.

9.3. Materialization

Given a production $p : L \leftarrowtail I \rightarrowtail R$, an abstract object A and an arrow from L to A ($\varphi\colon L \to A$), we want to rewrite the so-called *materialization* of φ in order to characterize all successors of objects in $\mathcal{L}(A)$, obtained by rewriting via p at a match compatible with φ. In a sense, we want to lift DPO rewriting to the level of abstract objects. For this purpose, we will now categorically define a materialization construction, that enables us to concretize an instance of a left-hand side L of a production p in a given abstract object A. Afterwards, we will refine the construction in a way such that we restrict to materializations that satisfy the gluing condition and can thus be rewritten via p. Last, we present the main result of this chapter, which allows us to fully characterize the co-matches obtained by rewriting. Please note that in the following we only consider productions where both morphisms are monos.

9.3.1. Materialization Category and Existence of Materialization

From now on we assume $\mathbf{C}$ to be an elementary topos. Let $A \in \mathbf{C}$ be an abstract object specifying a language of objects $\mathcal{L}(A)$ and let L be the left-hand side of a production with $L \dashrightarrow A$. We want to find a way to materialize objects with a concrete image of L from A. To this end we consider the following category.

Definition 9.18 (Materialization)**.** Let $\varphi\colon L \to A$ be an arrow in $\mathbf{C}$. The *materialization category* for φ, denoted $\mathbf{Mat}_\varphi$, has as

objects all *factorizations* $L \rightarrowtail X \to A$ of φ whose first factor $L \rightarrowtail X$ is a mono, and as

arrows from a factorization $L \rightarrowtail X \to A$ to another one $L \rightarrowtail Y \to A$, all arrows $f\colon X \to Y$ in $\mathbf{C}$ such that the diagram to the right comprises a commutative triangle and a pullback square.

$$\begin{tikzcd}
L \arrow[r, tail] \arrow[d, tail, "id_L"'] \arrow[rr, dashed, bend left, "\varphi"] \arrow[dr, phantom, "\text{(PB)}"] & X \arrow[d, "f"] \arrow[dr] & \\
L \arrow[r, tail] \arrow[rr, dashed, bend right, "\bar{\varphi}"'] & Y \arrow[r] & A
\end{tikzcd}$$

If $\mathbf{Mat}_\varphi$ has a terminal object it is denoted by $L \rightarrowtail \langle\varphi\rangle \to A$ and is called the *materialization* of φ.

Sometimes we will also call $\langle\varphi\rangle$ the materialization of φ, omitting the arrows.

Since we are working in a topos by assumption, the slice category over A provides us with a convenient setting to construct materializations. Note in particular that in the diagram in Definition 9.18 above, the span $X \leftarrowtail L \rightarrowtail L$ is a partial map from X to L in the slice category over A. Hence the materialization $\langle\varphi\rangle$ corresponds to the partial map classifier for L in this slice category.

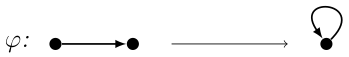
Proposition 9.19 (Existence of materialization)**.** Let $\varphi\colon L \to A$ be an arrow in $\mathbf{C}$, and let $\eta_\varphi\colon \varphi \to \mathcal{F}(\varphi)$, with $\mathcal{F}(\varphi)\colon \bar{A} \to A$, be the partial map classifier of φ in the slice category $\mathbf{C}{\downarrow}A$ (which also is a topos)^2. Then $L \overset{\eta_\varphi}{\to} \bar{A} \overset{\mathcal{F}(\varphi)}{\to} A$ is the materialization of φ, hence $\langle\varphi\rangle = \bar{A}$.

As a direct consequence of Proposition 9.15 and Proposition 9.19 (and the fact that final pullback complements in the slice category correspond to those in the base category [Löw10]), the terminal object of the materialization category can be constructed for each arrow of a topos by taking final pullback complements.

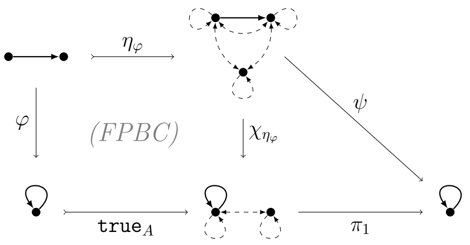
Corollary 9.20 (Construction of the materialization)**.** Let $\varphi\colon L \to A$ be an arrow of $\mathbf{C}$ and let $\mathtt{true}_A\colon A \rightarrowtail A \times \Omega$ be the subobject classifier in the slice category $\mathbf{C} \downarrow A$ mapping from $id_A\colon A \to A$ to the projection $\pi_1\colon A \times \Omega \to A$ (see also Fact 9.11). Then the terminal object $L \overset{\eta_\varphi}{\rightarrowtail} \langle\varphi\rangle \overset{\psi}{\to} A$ in the materialization category consists of the arrows $\eta_\varphi, \psi = \pi_1 \circ \chi_{\eta_\varphi}$, where $L \overset{\eta_\varphi}{\rightarrowtail} \langle\varphi\rangle \overset{\chi_{\eta_\varphi}}{\to} A \times \Omega$ is the final pullback complement of $L \overset{\varphi}{\to} A \overset{\mathtt{true}_A}{\rightarrowtail} A \times \Omega$.

$$\begin{tikzcd}
L \arrow[r, tail, dashed, "\eta_\varphi"] \arrow[d, "\varphi"'] \arrow[dr, phantom, "\text{(FPBC)}"] & \langle\varphi\rangle \arrow[d, dashed, "\chi_{\eta_\varphi}"] \arrow[dr, dashed, "\psi"] & \\
A \arrow[r, tail, dashed, "\mathtt{true}_A"'] & A \times \Omega \arrow[r, "\pi_1"'] & A
\end{tikzcd}$$

Example 9.21. *We construct the materialization* $L \overset{\eta_\varphi}{\rightarrowtail} \langle\varphi\rangle \overset{\psi}{\to} A$ *for the morphism* $\varphi\colon L \to A$ *of graphs with a single (omitted) egde label:*



In particular, the materialization is obtained as a final pullback complement as depicted to the right (compare with the corresponding diagram in Corollary 9.20). Note that edges which are not in the image of η_φ *resp.* $\mathtt{true}_A$ *are dashed.*



This construction corresponds to the usual intuition behind materialization: the left-hand side and the edges that are attached to it are "pulled out" of the given abstract graph. The concrete construction in the category $\mathbf{Graph}_\Lambda$ is spelled out in Chapter 11, so that we can implement the construction in a tool.

We can summarize the result of our constructions in the following proposition, which states that the materialization characterizes all objects X, abstracted over A, which contain a (mono) occurrence of the left-hand side compatible with φ.

Proposition 9.22 (Language of the materialization)**.** Let $\varphi\colon L \to A$ be an arrow in $\mathbf{C}$ and let $L \overset{\eta_\varphi}{\rightarrowtail} \langle\varphi\rangle \to A$ be the corresponding materialization. Then

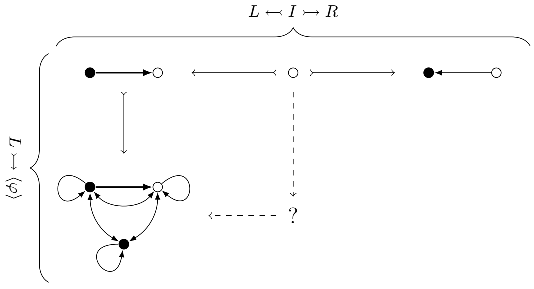
$$\mathcal{L}(L \overset{\eta_\varphi}{\rightarrowtail} \langle\varphi\rangle) = \{L \overset{m_L}{\rightarrowtail} X \mid \exists\psi\colon (X \to A).\ (\varphi = \psi \circ m_L)\}.$$

9.3.2. Characterizing the Language of Rewritable Objects

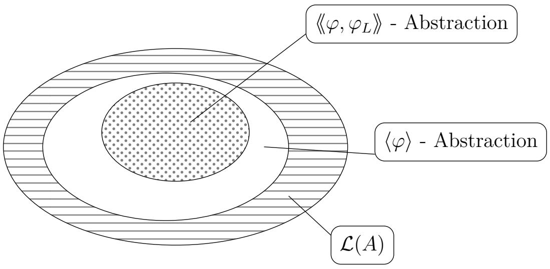
A match obtained through the materialization of the left-hand side of a production from a given object may not allow a DPO rewriting step because of the gluing condition. However, there may exist objects abstracted by the materialization for which there exist successors by the application of the production. We illustrate this problem with an example.

[2] [Fre72, Theorem 2.31].

Example 9.23. *Consider the materialization $L \rightarrowtail \langle\varphi\rangle \to A$ from Example 9.21 and the production $p\colon L \leftarrowtail I \rightarrowtail R$ shown in the diagram to the right. It is easy to see that the pushout complement of morphisms $I \rightarrowtail L \rightarrowtail \langle\varphi\rangle$ does not exist.*
Nevertheless there exist factorizations $L \rightarrowtail X \to A$ abstracted by $\langle\varphi\rangle$ that could be rewritten by p, for instance the factorization $L \rightarrowtail L \to A$.


$L \leftarrowtail I \rightarrowtail R$
$\langle\varphi\rangle \leftarrowtail L$
?

Therefore, we want to find the largest subobject (subsequently called $\langle\!\langle\varphi, \varphi_L\rangle\!\rangle$) of $\langle\varphi\rangle$, abstracting all objects X that can be rewritten via a production $L \leftarrowtail I \rightarrowtail R$ and a match $\varphi_L\colon L \to X$. The logical relation between the languages is depicted in the Venn digram shown to the right. Consequently, we consider the following subcategory of the materialization category.


$\langle\!\langle\varphi, \varphi_L\rangle\!\rangle$ - Abstraction
$\langle\varphi\rangle$ - Abstraction
$\mathcal{L}(A)$

Definition 9.24 (Materialization subcategory of rewritable objects)**.** Let $\varphi\colon L \to A$ be an arrow of $\mathbf{C}$ and let $\varphi_L\colon I \rightarrowtail L$ be a mono (corresponding to the left leg of a production). The *materialization subcategory of rewritable objects* for φ and φ_L, denoted $\mathbf{Mat}_\varphi^{\varphi_L}$, is the full subcategory of $\mathbf{Mat}_\varphi$ containing as objects all factorizations $L \overset{m}{\rightarrowtail} X \to A$ of φ, where m is a mono and $I \overset{\varphi_L}{\rightarrowtail} L \overset{m}{\rightarrowtail} X$ has a pushout complement.

Its terminal element, if it exists, is denoted by $L \overset{n_L}{\rightarrowtail} \langle\!\langle\varphi, \varphi_L\rangle\!\rangle \to A$ and is called the *rewritable materialization*.

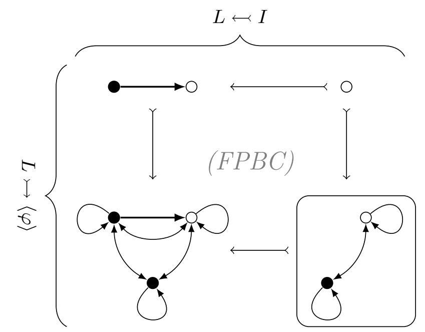
We next show that the subcategory $\mathbf{Mat}_\varphi^{\varphi_L}$ has a terminal object.

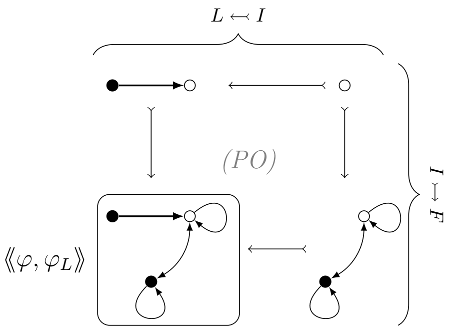
Proposition 9.25 (Construction of the rewritable materialization)**.** Let $\varphi\colon L \to A$ be an arrow and let $\varphi_L\colon I \rightarrowtail L$ be a mono of $\mathbf{C}$. Then the *rewritable materialization of φ w.r.t. φ_L* exists and can be constructed as the following factorization $L \overset{n_L}{\rightarrowtail} \langle\!\langle\varphi, \varphi_L\rangle\!\rangle \xrightarrow{\psi\circ\alpha} A$ of φ. In the diagram shown below to the left, F is obtained as the final pullback complement of $I \overset{\varphi_L}{\rightarrowtail} L \rightarrowtail \langle\varphi\rangle$, where $L \rightarrowtail \langle\varphi\rangle \overset{\psi}{\to} A$ is the materialization of φ (Definition 9.18). Next the diagram shown below to the right $L \overset{n_L}{\rightarrowtail} \langle\!\langle\varphi, \varphi_L\rangle\!\rangle \overset{\beta}{\leftarrowtail} F$ is the pushout of the span $L \overset{\varphi_L}{\leftarrowtail} I \rightarrowtail F$ and α is the resulting mediating arrow.

$$\begin{tikzcd}
 & L \arrow[dl, "\varphi"'] \arrow[d, tail] \arrow[dr, phantom, "\text{(FPBC)}"] & I \arrow[l, tail, "\varphi_L"'] \arrow[d, tail] \\
A & \langle\varphi\rangle \arrow[l, "\psi"] & F \arrow[l, tail]
\end{tikzcd} \tag{9.1}$$

$$\begin{tikzcd}
 & L \arrow[dl, "\varphi"'] \arrow[d, tail] & L \arrow[l, tail, "id_L"'] \arrow[d, tail, "n_L"'] \arrow[dr, phantom, "\text{(PO)}"] & I \arrow[l, tail, "\varphi_L"'] \arrow[d, tail] \\
A & \langle\varphi\rangle \arrow[l, "\psi"] & \langle\!\langle\varphi, \varphi_L\rangle\!\rangle \arrow[l, tail, dashed, "\alpha"'] & F \arrow[l, tail, "\beta"'] \arrow[ll, tail, bend left]
\end{tikzcd} \tag{9.2}$$

Example 9.26. *We come back to the running example (see Example 9.23) and, as in Proposition 9.25, we determine the final pullback complement $I \rightarrowtail F \rightarrowtail \langle\varphi\rangle$ of $I \overset{\varphi_L}{\rightarrowtail} L \rightarrowtail \langle\varphi\rangle$ (see diagram below left) and obtain $\langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ by taking the pushout over $L \leftarrowtail I \rightarrowtail F$ (see diagram below right).*





It remains to be shown that $L \rightarrowtail \langle\!\langle \varphi, \varphi_L \rangle\!\rangle \to A$ represents every factorization which can be rewritten. As before we obtain a characterization of the rewritable objects, including the match of the left-hand side L into the rewritable materialization $\langle\!\langle \varphi, \varphi_L \rangle\!\rangle$, as the language of an arrow.

Proposition 9.27 (Language of the rewritable materialization)**.** Assume there is a production $p\colon L \overset{\varphi_L}{\leftarrowtail} I \overset{\varphi_R}{\rightarrowtail} R$ and let $L \overset{n_L}{\rightarrowtail} \langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ be the match for the rewritable materialization for φ and φ_L. Then we have

$$\mathcal{L}(L \overset{n_L}{\rightarrowtail} \langle\!\langle \varphi, \varphi_L \rangle\!\rangle) = \{L \overset{m_L}{\rightarrowtail} X \mid \exists \psi\colon (X \to A).\ (\varphi = \psi \circ m_L \wedge X \overset{p,m_L}{\Longrightarrow})\}$$

where $X \overset{p,m_L}{\Longrightarrow}$ denotes that the object X can be rewritten.

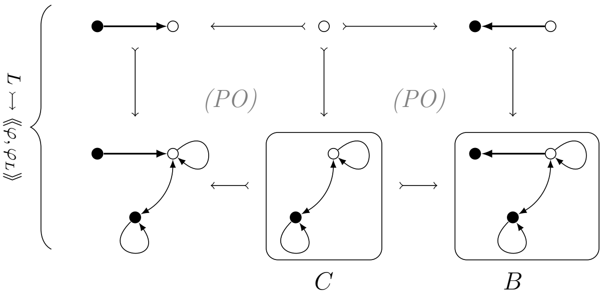
9.3.3. Rewriting Materializations

In the next step we will now rewrite the rewritable materialization $\langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ with the match $L \overset{n_L}{\rightarrowtail} \langle\!\langle \varphi, \varphi_L \rangle\!\rangle$, resulting in a co-match $R \rightarrowtail B$. In particular, we will show that this co-match represents all co-matches that can be obtained by rewriting an object X of $\mathcal{L}(A)$ at a match compatible with φ.

Proposition 9.28 (Rewriting abstract matches)**.** Let a match $n_L\colon L \rightarrowtail \tilde{A}$ and a production $p\colon L \leftarrowtail I \rightarrowtail R$ be given. Assume that $\tilde{A}$ is rewritten along the match n_L, i.e., $(L \overset{n_L}{\rightarrowtail} \tilde{A}) \overset{p}{\Rrightarrow} (R \overset{n_R}{\rightarrowtail} B)$. Then

$$\mathcal{L}(R \overset{n_R}{\rightarrowtail} B) = \{R \overset{m_R}{\rightarrowtail} Y \mid \exists (L \overset{m_L}{\rightarrowtail} X) \in \mathcal{L}(L \overset{n_L}{\rightarrowtail} \tilde{A})\colon\ ((L \overset{m_L}{\rightarrowtail} X) \overset{p}{\Rrightarrow} (R \overset{m_R}{\rightarrowtail} Y))\}$$

Example 9.29. *We can rewrite the materialization $L \rightarrowtail \langle\!\langle \varphi, \varphi_L \rangle\!\rangle \to A$ as follows:*


$L \rightarrowtail \langle\!\langle \varphi, \varphi_L \rangle\!\rangle$
(PO)
(PO)
C
B

If we combine Prop. 9.27 and Prop. 9.28, we immediately obtain the following corollary that characterizes the co-matches obtained from rewriting a match compatible with $\varphi \colon L \to A$.

Corollary 9.30 (Co-match language of the rewritable materialization)**.** Let $\varphi \colon L \to A$ and a production $p \colon L \overset{\varphi_L}{\leftarrowtail} I \overset{\varphi_R}{\rightarrowtail} R$ be given. Assume that $\langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ is obtained as the rewritable materialization of φ and φ_L with the match $L \overset{n_L}{\rightarrowtail} \langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ (see Proposition 9.25) and let $(L \overset{n_L}{\rightarrowtail} \langle\!\langle \varphi, \varphi_L \rangle\!\rangle) \overset{p}{\Rightarrow} (R \overset{n_R}{\rightarrowtail} B)$. Then

$$\begin{aligned}\mathcal{L}(R \overset{n_R}{\rightarrowtail} B) = \{R \overset{m_R}{\rightarrowtail} Y \mid \exists (L \overset{m_L}{\rightarrowtail} X), (X \overset{\psi}{\to} A) \colon \\ (\varphi = \psi \circ m_L \wedge (L \overset{m_L}{\rightarrowtail} X) \overset{p}{\Rightarrow} (R \overset{m_R}{\rightarrowtail} Y))\}\end{aligned}$$

This result does not yet enable us to construct post-conditions for languages of objects. The set of co-matches can be fully characterized as the language of a mono, which can only be achieved by fixing the right-hand side R and thus ensuring that exactly one occurrence of R is represented. However, as soon as we forget about the co-match, this effect is gone and can only be retrieved by adding annotations, which we will do in the next chapter.

"The only kind of writing is rewriting."

Ernest Hemingway (1899-1961)

10
Rewriting Annotated Objects

In this chapter we endow abstract objects with annotations, with the overall goal to be able to specify post-conditions for abstract rewriting steps. In particular, we will use ordered monoids enhanced with a subtraction operation to annotate objects. Furthermore, we will instantiate the materialization construction from the previous chapter in this framework to rewrite abstract objects with annotations. For this purpose, we specify how annotations can be computed within pushout constructions. We start by introducing the annotations for the abstract objects.

10.1. Additional Preliminaries - Annotated Objects

We will annotate each abstract object with pairs of annotations, denoting upper and lower bounds (see also Chapter 8). An object will belong to the corresponding language only if it has a legal arrow to the abstract object satisfying such bounds. We enrich ordered monoids (introduced in Chapter 8.1) with a *subtraction operation*, which we will need to define annotations for pushout objects. Similar annotations have already been studied in [Kön00] in the context of type systems.

Definition 10.1 (Ordered monoid with subtraction)**.** A tuple $(\mathcal{M}, +, -, \leq)$, where $(\mathcal{M}, +, \leq)$ is an ordered monoid and $-$ is a binary operation on $\mathcal{M}$, is called an *ordered monoid with subtraction*.
We say that subtraction is *well-behaved* whenever for all $a, b \in \mathcal{M}$ it holds that $a - a = 0$ and $(a - b) + b = a$ whenever $b \leq a$.

For now subtraction is just any operation, without specific requirements. Later we will concentrate on specific subtraction operations and demand that they are well-behaved. In the following we will consider only commutative monoids.

Definition 10.2 (Subtraction preserving maps)**.** Let $\mathcal{M}$, $\mathcal{M}'$ be two ordered monoids with subtraction. The category of ordered monoids with subtraction and monotone maps is called $\mathbf{Mon}^-$.
We say that a map $h \colon \mathcal{M} \to \mathcal{M}'$ *preserves subtraction* if $h(a-b) = h(a) - h(b)$.

We first have a look at some examples for ordered monoids with subtraction. We already introduced examples for ordered monoids in Chapter 8, which we now enrich with a subtraction operation.

Example 10.3. *Let $n \in \mathbb{N}$ and let $\mathcal{M}_n = \{0, 1, \ldots, n, *\}$ be the ordered monoid from Example 8.3. Then subtraction is truncated subtraction, i.e. $x - y = 0$ if $x \leq y$. Furthermore $* - x = *$ for all $x \in \{0, 1, \ldots, n\}$. It is easy to see that subtraction is well-behaved.*

Given a set S and an ordered monoid with subtraction $\mathcal{M}$, it is easy to check that also $\mathcal{M}^S$ is an ordered monoid with subtraction, where the elements are functions from S to $\mathcal{M}$ and the partial order, the monoidal operation and the subtraction are taken pointwise.

As another example we extend the path monoid to an ordered monoid with subtraction.

Example 10.4. *Given a graph G, let $\mathcal{P}_G$ be the* path monoid *from Example 8.4 and $P_0, P_1 \in \mathcal{P}_G$. Then subtraction simply returns the first parameter: $P_0 - P_1 = P_0$.*

We will now formally define annotations for objects via a functor from a given category to $\mathbf{Mon}^-$.

Definition 10.5 (Annotations for objects)**.** Given a category $\mathbf{C}$ and a functor $\mathcal{A}\colon \mathbf{C} \to \mathbf{Mon}^-$, an *annotation based on* $\mathcal{A}$ for an object $X \in \mathbf{C}$ is an element $a \in \mathcal{A}(X)$. We write $\mathcal{A}_\varphi$, instead of $\mathcal{A}(\varphi)$, for the action of functor $\mathcal{A}$ on a $\mathbf{C}$-arrow φ. Similar to Definition 8.5, we assume that for each object X there is a *standard annotation* based on $\mathcal{A}$ that we denote by s_X, thus $s_X \in \mathcal{A}(X)$.

It can be shown quite straightforwardly that the forgetful functor mapping an annotated object $X[a]$, with $a \in \mathcal{A}(X)$, to X is an op-fibration (or co-fibration [Jac99]), arising via the Grothendieck construction.

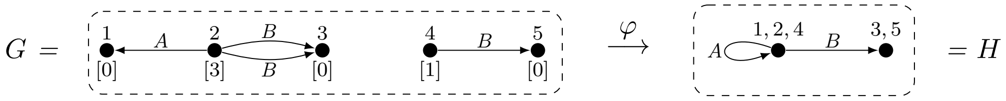
In Chapter 8 we already introduced two annotation functors for the category $\mathbf{Graph}_\Lambda$ based on the ordered monoid $\mathcal{M}_n$ and the path monoid $\mathcal{P}_G$ respectively, i.e. the multiplicity functor $\mathcal{B}^n$ (see Definition 8.6) and the path annotation functor $\mathcal{T}$ (see Definition 8.8). As another example we consider the following local annotation functor $\mathcal{S}^n$, based on the ordered monoid $\mathcal{M}_n$, which records the *out-degree of a node* and where the action of the functor is to take the supremum instead of the sum.

Definition 10.6 (Node out-degree annotation)**.** Given $n \in \mathbb{N}$, we define the functor $\mathcal{S}^n : \mathbf{Graph}_\Lambda \to \mathbf{Mon}^-$ as follows: For every graph G, $\mathcal{S}^n(G) = \mathcal{M}_n^{V_G}$. For every graph morphism $\varphi\colon G \to H$ and $a \in \mathcal{S}^n(G)$, we have $\mathcal{S}^n_\varphi(a) \in \mathcal{M}_n^{V_H}$ with:

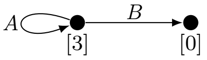
$$\mathcal{S}^n_\varphi(a)(w) = \bigvee_{\varphi(v)=w} a(v), \quad \textit{where } v \in V_G \textit{ and } w \in V_H$$

For a graph G, its *standard annotation* $s_G \in \mathcal{S}^n(G)$ is defined as the function which maps every node of G to its out-degree (or $*$ if the out-degree is larger than n).

Example 10.7. *Let the following graphs G and H with node out-degree annotations $a \in \mathcal{S}^3(G)$ be given. The out-degree of each node is indicated by the element of $\mathcal{M}_3 = \{0, 1, 2, 3, *\}$ shown in the brackets. Furthermore, let $\varphi\colon G \to H$ be a graph morphism indicated by the numbers above the nodes.*



Let $a' = \mathcal{S}^3_\varphi(a)$. We compute $a'(w)$ for each node $w \in V_H$ by taking the supremum of all $a(v) \in \mathcal{S}^3(G)$ for which $\varphi(v) = w$. The annotation a' is depicted on the right.



10.2. Annotation Properties

In the following we will consider only annotations satisfying certain properties in order to achieve soundness and completeness of our abstract rewriting steps.

Definition 10.8 (Properties of annotations)**.** Let $\mathcal{A} : \mathbf{C} \to \mathbf{Mon}^-$ be an annotation functor, together with standard annotations. We say that the

- *Homomorphism property* holds if whenever φ is a mono, then $\mathcal{A}_\varphi$ is a monoid homomorphism, preserving also subtraction.
- *Adjunction property* holds if whenever $\varphi\colon A \rightarrowtail B$ is a mono, then
  - $\mathcal{A}_\varphi\colon \mathcal{A}(A) \to \mathcal{A}(B)$ has a right adjoint $\mathit{red}_\varphi\colon \mathcal{A}(B) \to \mathcal{A}(A)$, i.e., red_φ is monotone and satisfies $a \le \mathit{red}_\varphi(\mathcal{A}_\varphi(a))$ for $a \in \mathcal{A}(A)$ and $\mathcal{A}_\varphi(\mathit{red}_\varphi(b)) \le b$ for $b \in \mathcal{A}(B)$.[1]
  - red_φ is a monoid homomorphism that preserves subtraction.
  - it holds that $\mathit{red}_\varphi(s_B) = s_A$, where s_A, s_B are standard annotations.

Furthermore, assuming that $\mathcal{A}_\varphi$ has a right adjoint red_φ, we say that the

- *Pushout property* holds, whenever for each pushout as shown in the diagram to the right, with all arrows monos where $\eta = \psi_1 \circ \varphi_1 = \psi_2 \circ \varphi_2$, it holds that for every $d \in \mathcal{A}(D)$:

$$\begin{tikzcd} A \arrow[r, tail, "\varphi_2"] \arrow[d, tail, "\varphi_1"'] \arrow[dr, tail, dashed, "\eta"] & C \arrow[d, tail, "\psi_2"] \\ B \arrow[r, tail, "\psi_1"'] & D \end{tikzcd}$$

$$d = \mathcal{A}_{\psi_1}(\mathit{red}_{\psi_1}(d)) + (\mathcal{A}_{\psi_2}(\mathit{red}_{\psi_2}(d)) - \mathcal{A}_\eta(\mathit{red}_\eta(d))).^2$$

  We say that the *pushout property for standard annotations* holds if we replace d by s_D, $\mathit{red}_{\psi_1}(d)$ by s_B, $\mathit{red}_{\psi_2}(d)$ by s_C and $\mathit{red}_\eta(d)$ by s_A.

- *Beck-Chevalley property* holds if whenever the square shown to the right is a pullback with φ_1, ψ_2 mono, then it holds for every $b \in \mathcal{A}(B)$ that

$$\begin{tikzcd} A \arrow[r, "\varphi_2"] \arrow[d, tail, "\varphi_1"'] \arrow[dr, phantom, "\text{(PB)}"] & C \arrow[d, tail, "\psi_2"] \\ B \arrow[r, "\psi_1"'] & D \end{tikzcd}$$

$$\mathcal{A}_{\varphi_2}(\mathit{red}_{\varphi_1}(b)) = \mathit{red}_{\psi_2}(\mathcal{A}_{\psi_1}(b)).$$

[1] This amounts to saying that the forgetful functor is a bifibration when we restrict to monos, see [Jac99, Lem. 9.1.2].

[2] Note that the brackets are essential, for instance in $\mathcal{M}_3$ we have $2+(2-1) = 3 \neq * = (2+2)-1$.

Note that the multiplicity functor from Definition 8.6 satisfies all properties above. In particular, for a given injective graph morphism $\varphi\colon G \rightarrowtail H$ the right adjoint $red_\varphi : \mathcal{M}_n^{V_H \cup E_H} \to \mathcal{M}_n^{V_G \cup E_G}$ to $\mathcal{B}_\varphi^n$ is defined as follows: given an annotation $b \in \mathcal{M}_n^{V_H \cup E_H}$, $red_\varphi(b)(x) = b(\varphi(x))$, i.e., red_φ simply provides a form of reindexing (see also Lemma L.5 in Appendix A.4).

The functors from Definition 8.8 and 10.6 satisfy both the homomorphism property and the pushout property for standard annotations, but do not satisfy all the remaining requirements (see Lemma L.6, L.7 and L.8 in Appendix A.6).

We will now introduce objects with two annotations, giving lower and upper bounds, which we will call *doubly annotated objects*. Therefore, doubly annotated objects are a generalization of annotated type graphs (see Definition 8.11) where the set of annotations consist of only one pair of annotations. Alternatively we could allow several pairs of annotations for objects, as in Chapter 8, yielding multiply annotated objects, however for the sake of simplicity we will only assume a single annotation.

Definition 10.9 (Doubly annotated object)**.** Given a topos $\mathbf{C}$ and a functor $\mathcal{A}\colon \mathbf{C} \to \mathbf{Mon}^-$, a *doubly annotated object* $A[a_1, a_2]$ is an object A of $\mathbf{C}$ with two annotations $a_1, a_2 \in \mathcal{A}(A)$. An arrow $\varphi\colon A[a_1, a_2] \to B[b_1, b_2]$, also called a *legal arrow*, is a $\mathbf{C}$-arrow $\varphi\colon A \to B$ such that $\mathcal{A}_\varphi(a_1) \geq b_1$ and $\mathcal{A}_\varphi(a_2) \leq b_2$.

The *language of a doubly annotated object* $A[a_1, a_2]$ (also called the language of objects which are abstracted by $A[a_1, a_2]$) is defined as follows:

$$\mathcal{L}(A[a_1, a_2]) = \{X \in \mathbf{C} \mid \text{there exists a legal arrow } \varphi\colon X[s_X, s_X] \to A[a_1, a_2]\}$$

Note that legal arrows are closed under composition (analogous to Lemma 8.13). Several examples of doubly annotated objects can be found in Section 8.3, where we considered an instance of the annotation functor in the category $\mathbf{Graph}_\Lambda$.

Another annotation property is the *isomorphism property* which – in contrast to the properties mentioned earlier – is defined specifically for standard annotations of doubly annotated objects.

Definition 10.10 (Isomorphism property)**.** Let $\mathcal{A}\colon \mathbf{C} \to \mathbf{Mon}^-$ be an annotation functor together with standard annotations. Then the functor $\mathcal{A}$ satisfies the *isomorphism property* if the following holds:

Whenever $\varphi\colon X[s_X, s_X] \to Y[s_Y, s_Y]$ is legal, then φ is an isomorphism, i.e. $\mathcal{L}(Y[s_Y, s_Y])$ contains only Y itself (and objects isomorphic to Y).

The multiplicity functor from Definition 8.6 satisfies the isomorphism property, however the path annotations (Definition 8.8) and the node out-degree annotations (Definition 10.6) violate this requirement: For instance, there exists a graph morphism $\varphi\colon G \to H$ mapping a discrete graph G which consists of two nodes into a discrete graph H consisting of a single node. Clearly, $\varphi\colon G[s_G, s_G] \to H[s_H, s_H]$ is legal if we consider standard annotations for both functors, however φ is not an isomorphism.

10.3. Abstract Rewriting of Annotated Objects

We will now show how to actually rewrite annotated objects. The challenge is both to find suitable annotations for the materialization and to "rewrite" the annotations.

10.3.1. Abstract Rewriting and Soundness

We first describe how the annotated rewritable materialization is constructed and then we investigate its properties.

Definition 10.11 (Construction of annotated rewritable materialization)**.** Let $p\colon L \overset{\varphi_L}{\leftarrowtail} I \overset{\varphi_R}{\rightarrowtail} R$ be a production and let $A[a_1, a_2]$ be a doubly annotated object. Furthermore let $\varphi\colon L \to A$ be an arrow.

We first construct the factorization $L \overset{n_L}{\rightarrowtail} \langle\!\langle \varphi, \varphi_L \rangle\!\rangle \overset{\psi}{\to} A$, obtaining the rewritable materialization $\langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ from Definition 9.24. Next let M contain all maximal[1]elements of the set

$$\{(a_1', a_2') \in \mathcal{A}(\langle\!\langle \varphi, \varphi_L \rangle\!\rangle)^2 \mid \mathcal{A}_{n_L}(s_L) \le a_2', \\ a_1 \le \mathcal{A}_\psi(a_1'), \mathcal{A}_\psi(a_2') \le a_2\}.$$

Then the doubly annotated objects $\langle\!\langle \varphi, \varphi_L \rangle\!\rangle[a_1', a_2']$ with $(a_1', a_2') \in M$ are the annotated rewritable materializations for $A[a_1, a_2]$, φ and φ_L.

Note that in general there are several materializations, differing for the annotations only, or possibly none. The definition of M ensures that the upper bound a_2' of the materialization covers the annotations arising from the left-hand side. We can not use a corresponding condition for the lower bound, since the materialization might contain additional structures, hence the arrow n_L is only *semi-legal*. A more symmetric condition will be studied in Section 10.3.2.

Proposition 10.12 (Annotated rewritable materialization is terminal)**.** Let $p\colon L \overset{\varphi_L}{\leftarrowtail} I \overset{\varphi_R}{\rightarrowtail} R$ be a production and let $L \overset{m_L}{\rightarrowtail} X$ be the match of L in an object X such that $X \overset{p,m_L}{\Longrightarrow}$, i.e., X can be rewritten. Assume that X is abstracted by $A[a_1, a_2]$, witnessed by ψ. Let $\varphi = \psi \circ m_L$ and let $L \overset{n_L}{\rightarrowtail} \langle\!\langle \varphi, \varphi_L \rangle\!\rangle \overset{\psi'}{\to} A$ be the corresponding rewritable materialization. Then there exists an arrow ζ_A and a pair of annotations $(a_1', a_2') \in M$ for $\langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ (as described in Definition 10.11) such that the diagram below commutes and the square is a pullback in the underlying category. Furthermore the triangle consists of legal arrows. This means in particular that ζ_A is legal.

$$\begin{tikzcd}
L[s_L, s_L] \arrow[r, tail, "m_L"] \arrow[d, tail, "id_L"'] \arrow[dr, phantom, "(\mathrm{PB})"] & X[s_X, s_X] \arrow[r, "\psi"] \arrow[d, "\zeta_A"] & A[a_1, a_2] \\
L[s_L, s_L] \arrow[r, tail, "n_L"'] & \langle\!\langle \varphi, \varphi_L \rangle\!\rangle[a_1', a_2'] \arrow[ur, "\psi'"'] &
\end{tikzcd}$$

[1] *Maximal* means maximality w.r.t. the interval order $(a_1, a_2) \sqsubseteq (a_1', a_2') \iff a_1' \le a_1, a_2 \le a_2'$.

Once we have performed the materialization, we will now show how to rewrite annotated objects. Note that we cannot simply take pushouts in the category of annotated objects and legal arrows, since this would result in taking the supremum of annotations, when instead we need the sum (subtracting the annotation of the interface I, analogous to the inclusion-exclusion principle).

Definition 10.13 (Abstract rewriting step $\rightsquigarrow$)**.** Let $p\colon L \overset{\varphi_L}{\leftarrowtail} I \overset{\varphi_R}{\rightarrowtail} R$ be a production and let $A[a_1, a_2]$ be an annotated abstract object. Furthermore let $\varphi\colon L \to A$ be a match of a left-hand side, let $n_L\colon L \rightarrowtail \langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ be the match obtained via materialization and let $(a'_1, a'_2) \in M$ (as in Definition 10.11).

Then $A[a_1, a_2]$ can be transformed to $B[b_1, b_2]$ via p whenever there are arrows such that the two squares below are pushouts in the base category and b_1, b_2 are defined as:

$$b_i = \mathcal{A}_{\varphi_B}(c_i) + (\mathcal{A}_{n_R}(s_R) - \mathcal{A}_{n_R \circ \varphi_R}(s_I)) \qquad \text{for } i \in \{1, 2\}$$

where c_1, c_2 are maximal annotations such that:

$$\begin{aligned} a'_1 &\leq \mathcal{A}_{\varphi_A}(c_1) + (\mathcal{A}_{n_L}(s_L) - \mathcal{A}_{n_L \circ \varphi_L}(s_I)) \\ a'_2 &\geq \mathcal{A}_{\varphi_A}(c_2) + (\mathcal{A}_{n_L}(s_L) - \mathcal{A}_{n_L \circ \varphi_L}(s_I)) \end{aligned}$$

$$\begin{tikzcd}
L[s_L, s_L] \arrow[d, tail, "n_L"'] & I[s_I, s_I] \arrow[l, tail, "\varphi_L"'] \arrow[r, tail, "\varphi_R"] \arrow[d, tail, "n_I"] & R[s_R, s_R] \arrow[d, tail, "n_R"] \\
\langle\!\langle \varphi, \varphi_L \rangle\!\rangle[a'_1, a'_2] & C[c_1, c_2] \arrow[l, tail, "\varphi_A"'] \arrow[r, tail, "\varphi_B"] & B[b_1, b_2]
\end{tikzcd}$$

In this case we write $A[a_1, a_2] \overset{p,\varphi}{\rightsquigarrow} B[b_1, b_2]$ and say that $A[a_1, a_2]$ makes an *abstract rewriting step* to $B[b_1, b_2]$.

We will now show soundness of abstract rewriting, i.e., whenever an object X is abstracted by $A[a_1, a_2]$ and X is rewritten to Y, then there exists an abstract rewriting step from $A[a_1, a_2]$ to $B[b_1, b_2]$ such that Y is abstracted by $B[b_1, b_2]$.

Assumption: In the following we will require that the homomorphism property as well as the pushout property for standard annotations hold (cf. Definition 10.8).

Proposition 10.14 (Soundness for $\rightsquigarrow$)**.** The relation $\rightsquigarrow$ is sound, i.e. if $X \in \mathcal{L}(A[a_1, a_2])$ (witnessed via a legal arrow $\psi\colon X[s_X, s_X] \to A[a_1, a_2]$) and $X \overset{p,m_L}{\Longrightarrow} Y$, then there exists an abstract rewriting step $A[a_1, a_2] \overset{p,\psi \circ m_L}{\rightsquigarrow} B[b_1, b_2]$ such that $Y \in \mathcal{L}(B[b_1, b_2])$.

10.3.2. Completeness

The conditions that we imposed so far are too weak to guarantee completeness, that is the fact that every object represented by $B[b_1, b_2]$ can be obtained by rewriting an object represented by $A[a_1, a_2]$. This can be clearly seen by the fact that the requirements hold also for the singleton monoid and, as discussed before, the graph structure of B is insufficient to characterize the successor objects or graphs.

Hence we will now strengthen our requirements in order to obtain completeness.

Assumption: In addition to the assumptions of Section 10.3.1, we will need that subtraction is well-behaved and that the adjunction property, the isomorphism property, the pushout property and the Beck-Chevalley property hold (cf. Definition 10.8).

The multiplicities from Definition 8.6 satisfy all these properties. We will now modify the abstract rewriting relation and allow only those abstract annotations for the materialization that reduce to the standard annotation of the left-hand side.

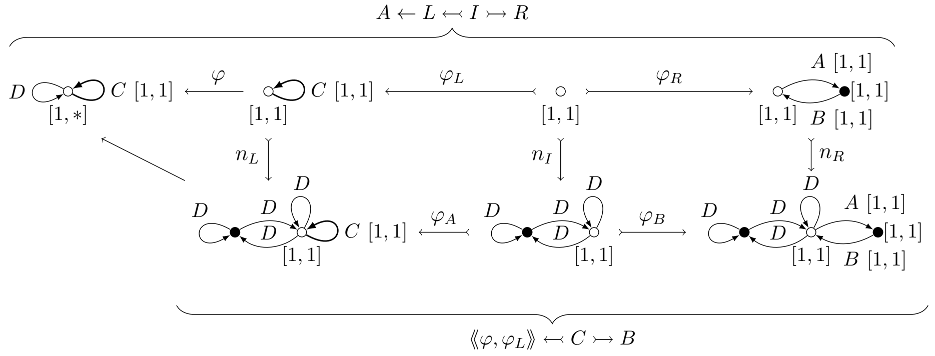
Definition 10.15 (Abstract rewriting step $\hookrightarrow$)**.** Given $\varphi\colon L \to A$, assume that $B[b_1, b_2]$ is constructed from $A[a_1, a_2]$ via the construction described in Definition 10.11 and 10.13, with the modification that the set of annotations from which the set of maximal annotations M of the materialization $\langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ are taken, is replaced by:

$$\begin{aligned}\{(a_1', a_2') \in \mathcal{A}(\langle\!\langle \varphi, \varphi_L \rangle\!\rangle)^2 \mid \mathit{red}_{n_L}(a_i') = s_L \text{ for } i \in \{1,2\},\\ a_1 \le \mathcal{A}_\psi(a_1'), \mathcal{A}_\psi(a_2') \le a_2\}.\end{aligned}$$

In this case we write $A[a_1, a_2] \overset{p,\varphi}{\hookrightarrow} B[b_1, b_2]$.

Due to the adjunction property we have $\mathcal{A}_{n_L}(s_L) = \mathcal{A}_{n_L}(\mathit{red}_{n_L}(a_2')) \le a_2'$ and hence the set M of annotations of Definition 10.15 is a subset of the corresponding set of Definition 10.13. We give a small example of an abstract rewriting step.

Example 10.16. *Consider the diagram shown below which depicts an abstract rewriting step. Elements without annotation are annotated by* $[0, *]$ *by default and those with annotation* $[0, 0]$ *are omitted. Furthermore elements in the image of the match and co-match are annotated by the standard annotation* $[1, 1]$ *to specify the concrete occurrence of the left-hand and right-hand side.*



First, the concrete instance of the left hand side L is materialized out of the abstract graph A (shown to the left), resulting in the rewritable materialization $\langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ *alongside the match* n_L*. Please note that the materialization only consists of one C-labelled loop since there is only a standard annotated C-labelled loop available in A and we omitted all additional edges which would have the annotation* $[0, 0]$*. The rewriting step then replaces the unique C-labelled loop by the concrete instance of the right-hand side R of the rule, resulting in the annotated type graph B. The annotation of the context C gets carried over during the rewriting process since all morphisms are monos.*

The variant of abstract rewriting introduced in Def. 10.15 can be proven to be sound and complete, assuming the extra requirements stated above.

Proposition 10.17 (Soundness for $\hookrightarrow$)**.** The relation $\hookrightarrow$ is sound, i.e. if $X \in \mathcal{L}(A[a_1, a_2])$ (witnessed via a legal arrow $\psi\colon X[s_X, s_X] \to A[a_1, a_2]$) and $X \overset{p,m_L}{\Longrightarrow} Y$, then there exists an abstract rewriting step $A[a_1, a_2] \overset{p,\psi\circ m_L}{\hookrightarrow} B[b_1, b_2]$ such that $Y \in \mathcal{L}(B[b_1, b_2])$.

Proposition 10.18 (Completeness for $\hookrightarrow$)**.** The relation $\hookrightarrow$ is complete, i.e. if $A[a_1, a_2] \overset{p,\varphi}{\hookrightarrow} B[b_1, b_2]$ and $Y \in \mathcal{L}(B[b_1, b_2])$, then there exists $X \in \mathcal{L}(A[a_1, a_2])$ (witnessed via a legal arrow $\psi\colon X[s_X, s_X] \to A[a_1, a_2]$) such that $X \overset{p,m_L}{\Longrightarrow} Y$ and $\varphi = \psi \circ m_L$.

Finally, we can show that annotated objects of this kind are expressive enough to construct a *strongest post-condition*. For the construction we now allow several annotations for objects, analogous to Definition 8.11, such that we can represent the strongest post-condition with a single (multiply) annotated object $B[N]$ where N is a set of annotations. The structure of the object B is always the same since the structure is dependent on the arrow φ, but not on the annotation of the abstract object A.

Corollary 10.19 (Strongest post-condition)**.** Let $A[a_1, a_2]$ be an annotated object and let $\varphi\colon L \to A$. We obtain (several) abstract rewriting steps $A[a_1, a_2] \overset{p,\varphi}{\hookrightarrow} B[b_1, b_2]$, where we always obtain the same object B. Now let $N = \{(b_1, b_2) \mid A[a_1, a_2] \overset{p,\varphi}{\hookrightarrow} B[b_1, b_2]\}$. Then the *strongest post-condition* is the language of the multiply annotated object $B[N]$, i.e.

$$\begin{aligned}\mathcal{L}(B[N]) \quad = \quad & \{Y \mid \exists (X \in \mathcal{L}(A[a_1, a_2]), \text{witnessed by } \psi), \\ & (L \overset{m_L}{\rightarrowtail} X)\colon (\varphi = \psi \circ m_L \wedge X \overset{p,m_L}{\Longrightarrow} Y)\}\end{aligned}$$

We conclude with a worked example which shows that, thanks to the result above, multiply annotated type graphs can be used for verification methods such as invariant checking.

Example 10.20. *In the following, we give an example for the computation of a postcondition. We specify an online-shop scenario using an annotated abstract graph with the following edge label semantics:*

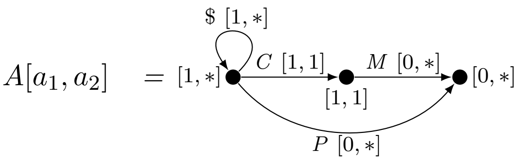
C: *The* **c***onnection of a customer node to the online-shop.*

M: *The* **m***arket relation describing which items are purchasable in the shop.*

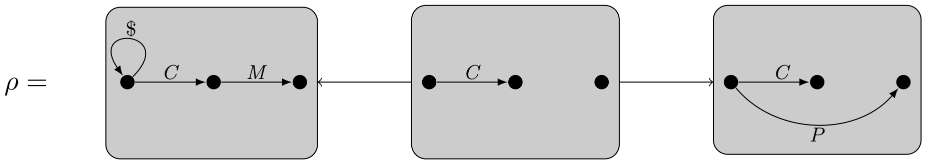
P: *The* **p***ossession relation describing which items are purchased by a customer.*

\$: *One* ***\$***-*coin of the currency used by customers to buy items in the shop.*

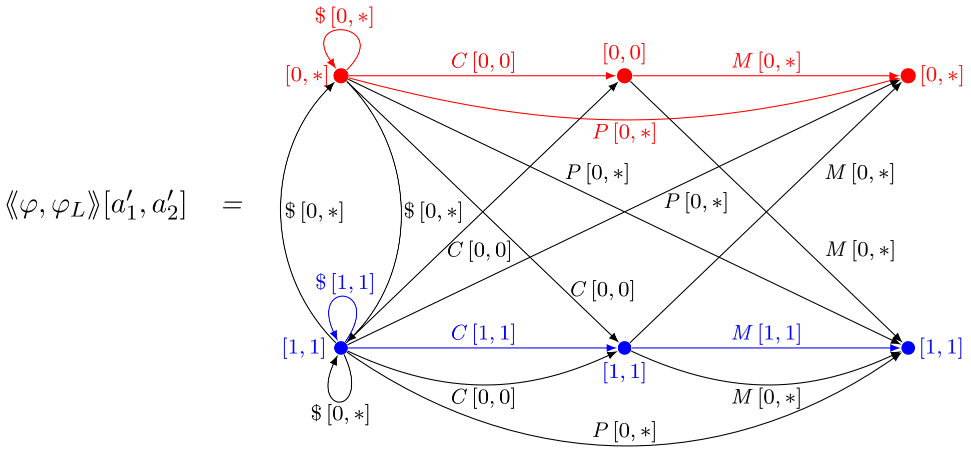
Now, we would like to model the following situation: Exactly one of many customers has established a connection to an online-shop. At least one of the customers has a \$-coin to purchase items and the online-shops have an arbitrary number of items *available. A customer can be in possession of an arbitrary number of items. Graphs modelling this specification can for instance be part of the language described by the following annotated abstract graph* $A[a_1, a_2]$*:*



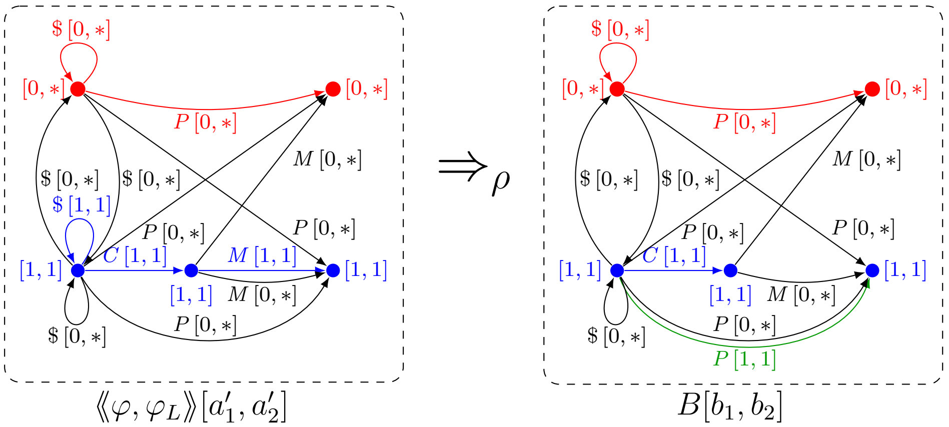
The following graph transformation production $\rho\colon L \leftarrowtail I \rightarrowtail R$ *specifies, that a customer, who is in possession of at least one $-coin and who is connected to the online shop, can purchase one of the items in stock in exchange for the currency. The production morphisms are indicated by the node positions:*



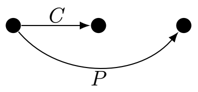
Please note, that there exists only one possibility to map the left-hand side graph L *of the production* ρ *into the abstract graph* A*. We now depict the rewritable abstract graph* $\langle\!\langle \varphi, \varphi_L \rangle\!\rangle[a'_1, a'_2]$ *consisting of the abstract graph* A *(upper part), the left-hand side graph* L *(lower part) and the additional edges introduced in the construction of Prop. 9.25 alongside a maximal pair of annotations* $(a'_1, a'_2) \in M$ *conforming to Definition 10.15:*



All elements in $\langle\!\langle \varphi, \varphi_L \rangle\!\rangle[a'_1, a'_2]$ *annotated with* $[0, 0]$ *cannot be the target of a legal morphism and therefore can be removed to simplify the graphical representation. If a node annotated with* $[0, 0]$ *is removed this way, all incident edges are removed as well independently of their annotation. We apply the production* ρ *to the simplified rewritable abstract graph* $\langle\!\langle \varphi, \varphi_L \rangle\!\rangle[a'_1, a'_2]$ *(shown below to the left) resulting in the abstract graph* $B[b_1, b_2]$ *(shown below, to the right):*



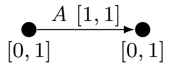
We can use the postcondition for an invariant check of the graph language $\mathcal{L}(A[a_1, a_2])$ with respect to the production ρ. In fact, the annotated abstract graph $B[b_1, b_2]$ specifies (a part of) the strongest postcondition and therefore the graph $G \in \mathcal{L}(B[b_1, b_2])$, shown to the right, is a witness for the fact that the graph language $\mathcal{L}(A[a_1, a_2])$ is not closed under application of ρ since $G \notin \mathcal{L}(A[a_1, a_2])$ due to a missing \$-coin edge in G, which is required by $A[a_1, a_2]$.



Please note that the simplification of the visual representation does not influence the computation of the strongest postcondition, however, it might hide information which is necessary to argue that the computed rewritable materialization is contained in the initial graph language. We illustrate this effect in a small example.

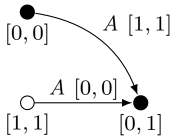
Example 10.21. *Let the graph transformation rule $\rho = L \leftarrow I \rightarrow R$ and the annotated type graph $T[a_1, a_2]$ shown below be given. Let $\varphi\colon L \to T$ match the single node of L with the left node of $T[a_1, a_2]$:*

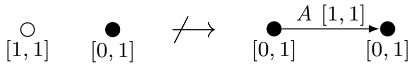
$\rho = \circ \leftarrowtail \circ \rightarrowtail \circ$ $T[a_1, a_2] =$



We construct the rewritable materialization $⟪\varphi, \varphi_L⟫[a'_1, a'_2]$, shown to the right, with a maximal pair of annotations $(a'_1, a'_2) \in M$ conforming to Definition 10.15. For the non-simplified version of the rewritable materialization $⟪\varphi, \varphi_L⟫[a'_1, a'_2]$ there clearly exists a non-monic legal morphism into the annotated type graph $T[a_1, a_2]$. However, simplifying the visual representation results in the annotated type graph shown below left from which we can not infer the existence of a legal morphism anymore, since the lower bound of the A-labeled edge in $T[a_1, a_2]$ can not be satisfied:

$⟪\varphi, \varphi_L⟫[a'_1, a'_2] =$





Conclusion of Part IV

We have described a rewriting framework for abstract graphs that also applies to objects in any topos, based on existing work for graphs [SWW11; BW07; Bac15; BR15a; RZ10; Ren04a]. In particular, we have given a blueprint for materialization in terms of the universal property of partial map classifiers. This is a first theoretical milestone towards shape analysis as a general static analysis method for rule-based systems with graph-like objects as states. Soundness and completeness results for rewriting of abstract objects with annotations in an ordered monoid provide an effective verification method for the special case of graphs (cf. Example 10.20). The results from Chapters 8,9 and 10, i.e. the materialization construction and the computation of rewriting steps of abstract graphs enriched with annotations have been implemented in the prototype tool DrAGoM which we will introduce in the upcoming chapters.

Related Work

The idea of shape graphs together with shape constraints was pioneered in [SRW02] where the constraints are specified in a three-valued logic. A similar approach was proposed in [SWW11], using first-order formulas as constraints. In partner abstraction [Bau06; BW07], cluster abstraction [Bac15; BR15a], and neighbourhood abstraction [RZ10] nodes are clustered according to local criteria, such as their neighbourhood and the resulting graph structures are enriched with counting constraints, similar to our constraints. The idea of counting multiplicities of nodes and edges is also found in canonical graph shapes [Ren04a].

Open Questions

The extension of annotations with logical formulas is the natural next step, which will lead to a more flexible and versatile specification language, as described in previous work [SRW02; SWW11]. The logic can possibly be developed in full generality using the framework of nested application conditions [HP05; LO14] that applies to objects in adhesive categories. This logical approach might even reduce the proof obligations for annotation functors.

Another open question is how to integrate widening or similar approximation techniques, which collapse abstract objects. Ideally, such techniques would lead to finite abstract transition systems that (over-)approximate the typically infinite transitions systems of graph transformation systems.

Part V.

Tools and Applications

"Machines take me by surprise with great frequency."
Alan Turing (1912-1954)

11 DrAGoM

In this chapter we describe the prototype tool called DrAGoM (abbreviation for: **Dir**ected **A**bstract **G**raphs **o**ver **M**ultiplicities) which is a software to handle and manipulate multiply annotated type graphs (introduced in Chapter 8). The main idea of DrAGoM is to be able to automatically check invariants of graph transformation systems, in abstract graph rewriting. For this purpose, we implemented the categorical notions introduced in Chapter 10 for the category of multiply annotated type graphs and legal morphisms.

First, in Section 11.1, we give an overview of the general functionalities of DrAGoM. A detailed explanation for the installation and usage of the tool can be found in Appendix C.1. Afterwards, in Section 11.2 we describe how the categorical notions of Part IV are implemented as a concrete instance for our annotated type graph framework. We close this chapter with a comparison to other graph verification tools in Section 11.3.

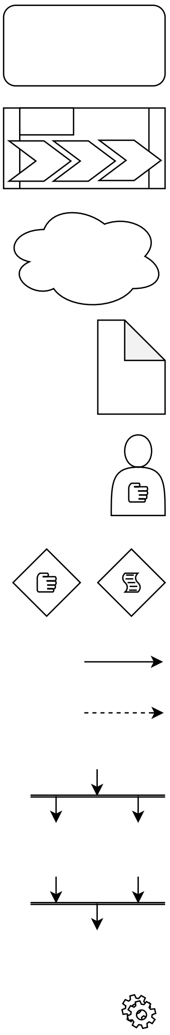
11.1. An Introduction to DrAGoM

The implementation of DrAGoM started in Spring 2016 as a prototype tool to visualize graphs contained in a graph language specified by an annotated type graph. In Fall 2018 the prototype tool was used as a base to build a new tool on top of it. Given a graph language specified by a multiply annotated type graph, we implemented techniques which are able to construct an abstract graph that specifies the strongest postcondition with respect to a graph transformation rule. To this end, we created DrAGoM.

To compute the strongest postcondition, DrAGoM uses the materialization construction introduced in Chapter 9 to extract a concrete instance of the left-hand side graph out of the abstract graph in every possible way. Afterwards, DrAGoM can perform a language inclusion check (see Proposition 8.20) to verify if the computed postcondition is already covered by the initial graph language. This way, the tool is able to check for invariants.

The DrAGoM-User Workflow

In this section we describe the typical workflow of a user interaction with DrAGoM. The workflow which we refer to is depicted in Figure 11.1. First, we explain the meaning of all elements in the shown flowchart:



Dialog Box A dialog that offers different functionalities which depend on the current user-interaction.

Algorithm/Construction An automatic process which performs several serial steps once started.

Visualization The visualization of a created or modified data structure is updated and displayed.

External File A file from which data structures can be loaded from or to save data structures into.

User The user who initializes the workflow. This is the flowcharts start point.

Choice A point where either the user (symbol shown to the left) or an algorithm (symbol shown to the right) makes a choice.

Flow The control flow (solid arrow) and object flow (dashed arrow) of the flow diagram.

Fork Splits the control flow into several parallel independent subflows illustrating different tasks.

Join Continues the main control flow once all subflows finish their tasks and reach the join node.

Algorithm Symbol Symbol which indicates the usage of an algorithm (see also Section 11.2).

In the following we will describe the flowchart (see Figure 11.1) in more detail.

The interaction starts with the user who initializes DrAGoM. Before the tool can start with the analysis, the user needs to create (or load) an annotated type graph (short: ATG) and a graph transformation system (short: GTS).

Annotated type graphs can be directly created via the user interface. After the creation, the user can add additional multiplicities to extend the annotated type graph to a multiply annotated type graph. For loading and storing multiply annotated type graphs, DrAGoM uses the XML based standard *Graph eXchange Language*[1] (GXL) [Win02; KRW02; HS+06]. An example for the file format, of an encoded multiply annotated type graph, is given in Appendix C.2.

[1] See also http://www.gupro.de/GXL/

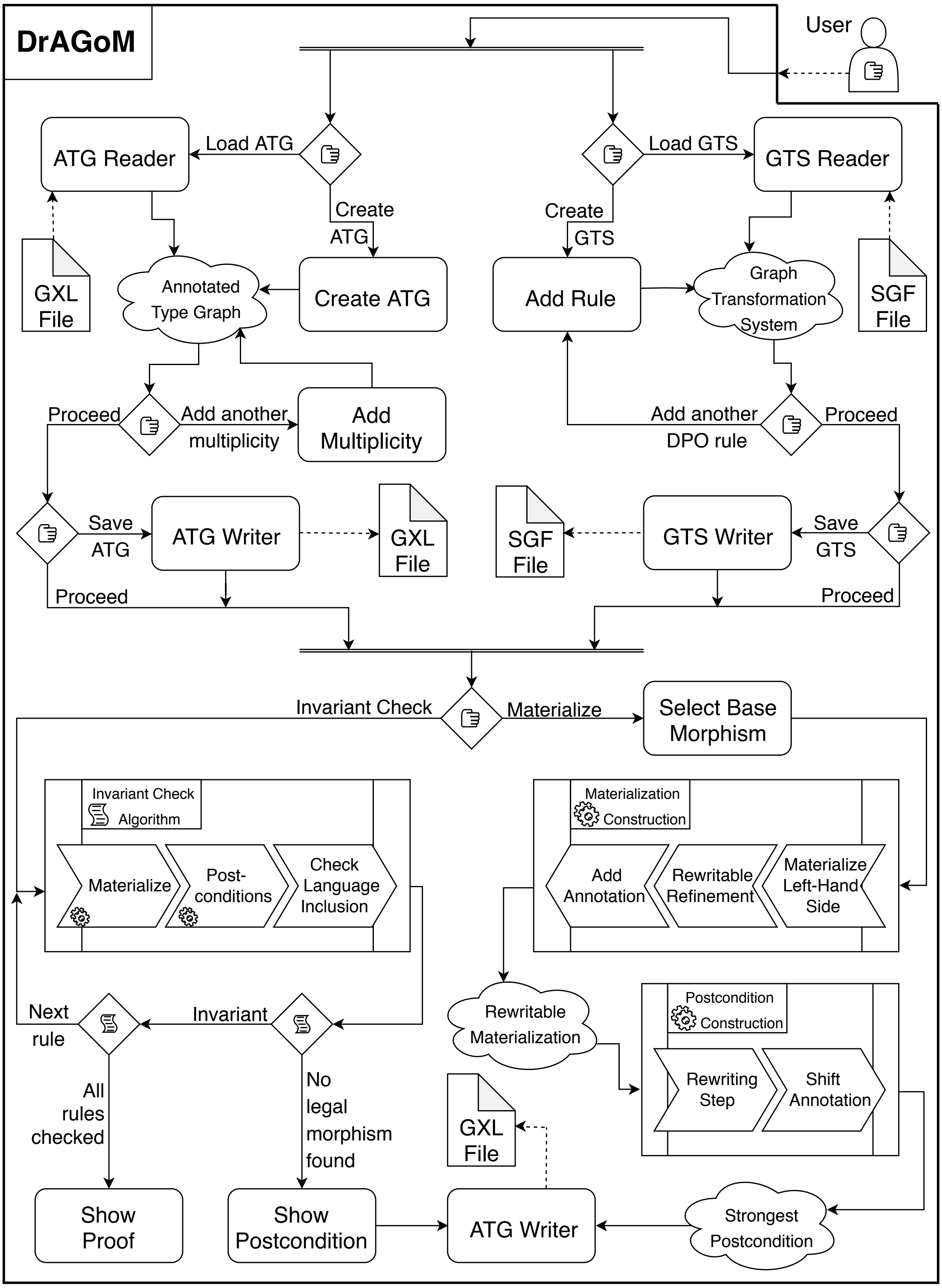


Figure 11.1.: UML flowchart of a typical user interaction with DrAGoM

Likewise, graph transformation systems can be directly created via the user interface. The user can create a graph transformation rule which is added to an empty graph transformation system. This process can be repeated until all rules have been added. For loading and storing graph transformation systems, DrAGoM uses the *Simple Graph Format* (SGF) [Bru15]. Unlike other text-based formats, such as XML-derived formats, which are mainly designed to be easily parsed by a computer program, the SGF format is designed to be easily written, read and maintained in source form by the user. An example for the file format of an encoded graph transformation system, can be found in Appendix C.3.

Once both data structures, the multiply annotated type graph and the graph transformation system, are created, the user can choose to either compute a rewritable materialization or to let DrAGoM perform an invariant check.

For the case that the user wants to compute the rewritable materialization, the currently displayed graph transformation rule is used for the construction. The user can select a semi-legal base morphism (if there are any) and afterwards DrAGoM automatically computes the rewritable materialization. This materialization can be used as an input for the computation of the strongest postcondition. The materialization algorithm alongside the postcondition algorithm are explained in more detail in Section 11.2. The user can choose to save the multiply annotated type graph which specifies the strongest postcondition in the GXL format. Furthermore, the user can choose to let DrAGoM perform a language inclusion check with respect to the computed postcondition and the initial annotated type graph.

If the user chooses to let DrAGoM perform an invariant check instead, both algorithms, the materialization construction (for all semi-legal base morphisms) and the postcondition construction, are performed in sequence. The gearwheel symbols in the algorithm steps indicate that these two tasks resemble the two constructions depicted on the right side of the diagram. Afterwards, a language inclusion check is used to possibly find a legal morphism from the graph specifying the postcondition to the graph that specifies the initial graph language. If such a legal morphism can not be found, the corresponding multiply annotated type graphs are displayed and DrAGoM is unable to prove that the initial specified graph language is an invariant for the given graph transformation system. Otherwise, the algorithm is restarted until every rule of the graph transformation system is checked. If for all rules and every semi-legal base morphism there exists a corresponding legal morphism for the postcondition graph, then all legal morphisms are displayed as a proof to the user.

11.2. Implementing Categorical Notions

The computation of the materialization and its annotations, which was described categorically for an arbitrary topos $\mathbf{C}$ in Chapter 9, needs to be implemented in DrAGoM as a concrete instance for our annotated type graph framework.

11.2.1. Concrete Construction of the Materialization

We define a concrete construction of the materialization $\langle\varphi\rangle$ in the category $\mathbf{Graph}_\Lambda$. Even though $\mathbf{Graph}_\Lambda$ is a topos, we do not use the notions of subobject classifiers or partial map classifiers for the following step-by-step construction:

Definition 11.1 (Concrete construction of the materialization)**.** Let two graphs $L = (V_L, E_L, src_L, tgt_L, lab_L)$ and $A = (V_A, E_A, src_A, tgt_A, lab_A)$ over a fixed edge label alphabet Λ be given and let $\varphi\colon L \to A$ be a fixed graph morphism.

First we define the function $\psi_V\colon (V_L \uplus V_A) \to V_A$ which maps the nodes of L and A to the nodes of A with respect to φ:

$$\psi_V(x) = \begin{cases} \varphi_V(x) & \text{if } x \in V_L \\ x & \text{otherwise} \end{cases}$$

We construct $\tilde{A} = (V, E, src, tgt, lab)$ in the following way:

$$\begin{aligned} V &= V_L \uplus V_A \\ E &= E_L \uplus \{(e, s, t, l) \in E_A \times V \times V \times \Lambda \mid \\ &\qquad\qquad src_A(e) = \psi_V(s) \wedge tgt_A(e) = \psi_V(t) \wedge lab_A(e) = l\} \end{aligned}$$

$$src\colon\ E \to V \qquad src(x) = \begin{cases} s & \text{if } x = (e, s, t, l) \\ src_L(x) & \text{otherwise} \end{cases}$$

$$tgt\colon\ E \to V \qquad tgt(x) = \begin{cases} t & \text{if } x = (e, s, t, l) \\ tgt_L(x) & \text{otherwise} \end{cases}$$

$$lab\colon\ E \to \Lambda \qquad lab(x) = \begin{cases} l & \text{if } x = (e, s, t, l) \\ lab_L(x) & \text{otherwise} \end{cases}$$

This concludes the construction of the graph $\tilde{A}$. We now define the embedding graph morphism $\alpha\colon L \to \tilde{A}$ where $\alpha(x) = x$ to get the diagram shown to the right.

$$\begin{tikzcd} L \arrow[r, tail, "\alpha"] \arrow[rr, bend right, "\varphi"'] & \tilde{A} & A \end{tikzcd}$$

To get a valid factorization $L \rightarrowtail \tilde{A} \to A$ of φ, we define the morphism $\psi\colon \tilde{A} \to A$ with $\psi = (\psi_V, \psi_E)$ where $\psi_E\colon E \to E_A$ is given by:

$$\psi_E(x) = \begin{cases} e & \text{if } x = (e, s, t, l) \\ \varphi_E(x) & \text{otherwise (i.e. } x \in E_L) \end{cases}$$

Obviously $\psi \circ \alpha = \varphi$ holds. The object $L \overset{\alpha}{\rightarrowtail} \tilde{A} \overset{\psi}{\to} A$ is a factorization of $L \overset{\varphi}{\to} A$ and the diagram shown to the right commutes.

$$\begin{tikzcd} L \arrow[r, tail, "\alpha"] \arrow[rr, bend right, "\varphi"'] & \tilde{A} \arrow[r, "\psi"] & A \end{tikzcd}$$

Next, we prove that the above constructed object $L \overset{\alpha}{\rightarrowtail} \tilde{A} \overset{\psi}{\to} A$ is the terminal object in the materialization category $\mathbf{Mat}_\varphi$, i.e. we have $\tilde{A} = \langle \varphi \rangle$.

Proposition 11.2 (Constructed materialization is terminal)**.** Let $L \overset{\varphi}{\to} A$ be a fixed graph morphism in $\mathbf{Graph}_\Lambda$. Then the factorization $L \overset{\alpha}{\rightarrowtail} \tilde{A} \overset{\psi}{\to} A$ from Definition 11.1 is the terminal object in the category $\mathbf{Mat}_\varphi$.

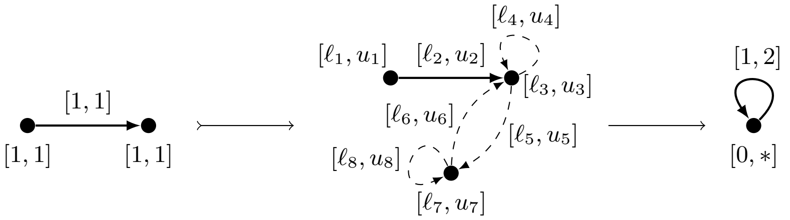
Therefore, the construction of Definition 11.1 is a guideline which tells us how to implement a materialization algorithm step-by-step. Since we are working with injective rule morphisms, it is straightforward to refine this construction into one for the rewritable materialization $\langle\!\langle \varphi, \varphi_L \rangle\!\rangle$. This can be achieved by removing all edges incident to nodes which are not contained in the codomain of the left-hand side morphism of the rule and by subsequently restricting the domain of the embedding morphism ψ, i.e. we obtain $\psi' \colon \langle\!\langle \varphi, \varphi_L \rangle\!\rangle \to A$ with $\psi' = \psi|_{\langle\!\langle \varphi, \varphi_L \rangle\!\rangle}$.

11.2.2. Computation of Annotations

Once the rewritable materialization $\langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ is created, we can annotate its elements. To enrich the rewritable materialization, with a set of maximal pairs of annotations $(a_1, a_2) \in M$ which conform Definition 10.15, DrAGoM employs the SMT solver Z3. The reason behind this approach is the fact that a brute-force search for valid multiplicities is too costly with respect to computation time. Instead, we encode the problem of annotation computation into an SMT formula.

Let $A[M']$ be a multiply annotated type graph and let $L \overset{\alpha}{\rightarrowtail} \langle\!\langle \varphi, \varphi_L \rangle\!\rangle \overset{\psi'}{\to} A$ be the factorization which contains the rewritable materialization $\langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ of a given fixed graph morphism $L \overset{\varphi}{\to} A$. We encode the problem into an SMT formula which yields a model in form of a maximal pair of annotations (a_1, a_2) whenever the two morphisms $\alpha \colon L[s_L, s_L] \rightarrowtail \langle\!\langle \varphi, \varphi_L \rangle\!\rangle[a_1, a_2]$ and $\psi' \colon \langle\!\langle \varphi, \varphi_L \rangle\!\rangle[a_1, a_2] \to A[M']$ are legal. We illustrate the encoding based on $\langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ from Example 9.29.

Example 11.3. *Let the factorization* $L[s_L, s_L] \overset{\alpha}{\rightarrowtail} \langle\!\langle \varphi, \varphi_L \rangle\!\rangle[\ell, u] \overset{\psi'}{\to} A[b_1, b_2]$ *from Example 9.29 over* $\mathcal{M}_3$ *be given (we only denote multiplicities):*



First, we encode the set of graph elements and our searched annotation functions. We use the SMT-LIB2 format [BST10] where operators are used in prefix notation.

(declare-datatypes () ((X x1 ... xN)))	$\mid X = V_{\langle\!\langle \varphi, \varphi_L \rangle\!\rangle} \cup E_{\langle\!\langle \varphi, \varphi_L \rangle\!\rangle}$
(declare-fun low (X) Int)	$\mid \ell \colon X \to \mathbb{N}_0$
(declare-fun up (X) Int)	$\mid u \colon X \to \mathbb{N}_0$

For our encoding, we do not distinguish between nodes and edges in the rewritable materialization. Instead, we simply refer to graph elements $x \in \langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ *for the two annotation functions* $\ell, u \colon \langle\!\langle \varphi, \varphi_L \rangle\!\rangle \to \mathcal{M}_3$*, where* $\ell(x) = \ell_x$ *and* $u(x) = u_x$ *in the visualized graph above.*
Next, we ensure that the codomain of the functions only consists of values in $\mathcal{M}_3$*, i.e.,* $\mathcal{M}_3 = \{0, 1, 2, *\}$ *where* $*$ *will be represented by the value* 3*. Furthermore, we encode that for the pair of annotations* $\ell \leq u$ *holds (see also Definition 8.11):*

(assert (forall ((x X)) (>= (low x) 0)))	$\mid \forall x \ (0 \leq \ell(x))$
(assert (forall ((x X)) (<= (up x) 3)))	$\mid \forall x \ (u(x) \leq *)$
(assert (forall ((x X)) (<= (low x) (up x))))	$\mid \forall x \ (\ell(x) \leq u(x))$

Instead of encoding the morphisms α and ψ', we use them to compute sets of nodes and edges with specific requirements. According to Definition 10.15 all elements in the image of α are annotated with the standard annotation $[1,1]$. In our example, $\alpha(L) = \{x_1, x_2, x_3\}$ and therefore we add the following six constraints:

(assert (= (low x1) 1)) $\mid \ell(x_1) = 1$ (assert (= (up x1) 1)) $\mid u(x_1) = 1$
(assert (= (low x2) 1)) $\mid \ell(x_2) = 1$ (assert (= (up x2) 1)) $\mid u(x_2) = 1$
(assert (= (low x3) 1)) $\mid \ell(x_3) = 1$ (assert (= (up x3) 1)) $\mid u(x_3) = 1$

Since we search for maximal pairs of annotations we set up the following constraints for all remaining elements $\bar{x} \in (\langle\!\langle \varphi, \varphi_L \rangle\!\rangle \setminus \alpha(L))$, i.e. elements which are not part of the left-hand side image: All remaining elements $\bar{x}$ for which $b_2(\psi'(\bar{x})) = *$, receive the upper bound $u(\bar{x}) = *$. In a similar way, the remaining elements $\bar{x}$ for which $b_1(\psi'(\bar{x})) = 0$, receive the lower bound $\ell(\bar{x}) = 0$. In our example this yields the constraint for x_7:

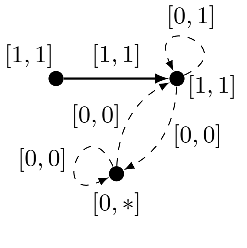
(assert (= (low x7) 0)) $\mid \ell(x_7) = 0$ (assert (= (up x7) 3)) $\mid u(x_7) = *$

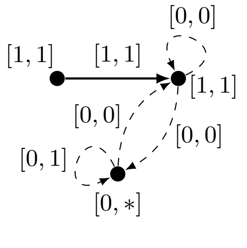
Last, we encode constraints for the remaining elements $\bar{x} \in (\langle\!\langle \varphi, \varphi_L \rangle\!\rangle \setminus \alpha(L))$ which are mapped onto type graph elements $t \in A$ with a multiplicity value b with $0 < b < *$: In case of $b_1(t) = b$, if $\bar{x} \in \psi'^{-1}(t)$ and for the number of left-hand side image elements $\alpha(L)$ in $\psi'^{-1}(t)$ we have $|\alpha(L) \cap \psi'^{-1}(t)| \geq b$, then $\ell(\bar{x}) = 0$. Otherwise, the sum of all corresponding lower bounds in $\psi'^{-1}(t)$ must be equal to b. Similarly the sum of all upper bounds in $\psi'^{-1}(t)$ must be equal to $b_2(t)$. For our running example we get the following constraints:

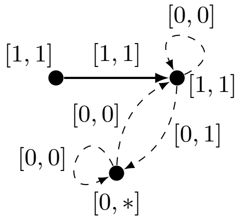
(assert (= (low x4) 0)) $\mid \ell(x_4) = 0$ (assert (= (low x6) 0)) $\mid \ell(x_6) = 0$
(assert (= (low x5) 0)) $\mid \ell(x_5) = 0$ (assert (= (low x8) 0)) $\mid \ell(x_8) = 0$

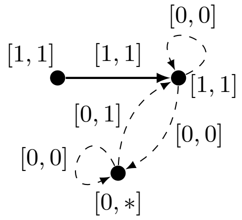
(assert (= (sum (up x2) (up x4) (up x5) (up x6) (up x8)) 2))
$\mid u(x_2) + u(x_4) + u(x_5) + u(x_6) + u(x_8) = 2$

This completes the SMT encoding. Whenever the solver finds a model, we get a pair of annotations (ℓ, u) which yields a legal morphism ψ' and the pair is maximal in the sense that we exactly hit all desired bounds, i.e. $(\ell, u) \in M$ conforming Definition 10.15. We then extend the SMT formula to exclude all previously found models and check for its satisfiability until the formula gets unsatisfiable. In each iteration step we collect another maximal pair. In our example we receive the following four pairs of annotations for the rewritable materialization $\langle\!\langle \varphi, \varphi_L \rangle\!\rangle[M]$:

$\langle\!\langle \varphi, \varphi_L \rangle\!\rangle[\ell, u] =$


$\langle\!\langle \varphi, \varphi_L \rangle\!\rangle[\ell'', u''] =$


$\langle\!\langle \varphi, \varphi_L \rangle\!\rangle[\ell', u'] =$ 

$\langle\!\langle \varphi, \varphi_L \rangle\!\rangle[\ell''', u'''] =$ 

In practice, DrAGoM uses an optimization where, in case of a universal quantification, we substitute the variable with every possible instance (e.g. with every element) and take the conjunction of the resulting formulas. This is possible since we always quantify over finite sets.

11.3. Other Verification Tools

Please note that DrAGoM is not supposed to be a universally applicable verification tool for abstract graph rewriting, but rather an evidence for the fact that the specification framework based on multiply annotated type graphs can be implemented. Of course, there exist implementations of various other frameworks and verification techniques, both for graph transformation systems and abstract graph rewriting. In this section, we spotlight some of these tools and sketch their functionalities and approaches.

The tool GROOVE[2] was originally created to support the use of graphs for modeling the design-time, compile-time, and run-time structure of object-oriented systems [Ren03], but since then has grown to be a full-fledged general-purpose graph transformation tool. The emphasis lies on the efficient exploration of the state space, given a particular graph transformation system. While doing so, GROOVE recognizes previously visited graphs modulo isomorphism, avoiding duplication during their exploration. For verification purposes, GROOVE has a built-in model checker that can run temporal logic queries over the state space.

For infinite state systems there exist implementations for verification purposes that do not depend on the exploration of the entire state space. Instead abstract interpretation can be used to over-approximate these systems. The tool AUGUR2[3] uses a technique called abstract unfolding and is based on so-called Petri graphs [BCK01] which are an over-approximation that consists of a hypergraph and an (attributed) Petri net. AUGUR2 verifies properties of graph transformation systems by using regular expressions, first order logic and coverability checking techniques for (attributed) Petri nets. Furthermore, the abstraction can be refined via counterexample-guided abstraction refinement [KK06; CG+03].

For recognizable graph languages, the tool suite RAVEN[4] provides verification techniques based on formal language theory for the notion of a so-called graph automaton [Blu14] which is a finite automaton to process graphs instead of words. RAVEN uses binary decision diagrams to symbolically encode the graph automaton and therefore the state space. Some functionalities of RAVEN are the computation of pre-defined classes of automata, the computations of their union

[2] GROOVE homepage: http://groove.cs.utwente.nl/
[3] AUGUR2 homepage: http://www.ti.inf.uni-due.de/research/tools/augur2/
[4] RAVEN homepage: https://www.uni-due.de/theoinf/research/tools_raven.php

and product automata, membership checks for the corresponding languages of cospan decomposed graphs, and invariant checks for graph transformation systems.

Then there exist other implementations which are designed for the analysis of pointer programs [Ren04a] and for shape analysis [SRW02]. In [SWW10; SW11] and [SWW11] the authors mention an implementation based on the source code of the shape analysis tool TVLA[5] [BL+07]. Given a shape graph, a set of shape productions and a set of forbidden patterns, the implementation constructs the set of reachable shape graphs where each graph is represented as a logical structure based on three-valued logic. If a reached shape graph contains the forbidden pattern, a counterexample in form of the respective derivation sequence is returned.

The tool JUGGRNAUT[6] [HJ+15] uses hyperedge replacement grammars to specify pointer program structures and their abstractions. The grammar rules are used in two directions: a backward applications of the rule abstracts a subgraph of the heap into a single nonterminal edge and a forward application can materialize parts of the heap.

Another abstraction mechanism, called partner abstraction, is represented in [Bau06] and implemented in the tool HIRALYSIS. The graph structures are over-approximated by the labels of adjacent nodes. This approach has been extended to so-called cluster abstractions in [BR15a], where the information about possible edges between partners is stored using three-valued logic. The tool ASTRA[7] [BR15b] uses the cluster abstraction approach and can be used for the static analysis and verification of topological structures in dynamic communication systems.

[5] TVLA homepage: http://www.cs.tau.ac.il/~tvla/

[6] JUGGRNAUT homepage: https://moves.rwth-aachen.de/research/projects/juggrnaut/

[7] ASTRA homepage: http://www.rw.cdl.uni-saarland.de/~rtc/astra/

“It doesn’t matter how beautiful your theory is, it doesn’t matter how smart you are. If it doesn’t agree with experiment, it’s wrong.”

Richard P. Feynman (1918-1988)

12 Evaluation

To evaluate the practicability of the framework based on multiply annotated type graphs, we conduct different case studies using DrAGoM. We focus on runtime results for invariant checking and test the limits of DrAGoM’s inclusion checks with respect to graph sizes. First, we evaluate the output based on examples of this thesis, to check if the materializations and strongest postconditions coincide with the theoretical results. Afterwards, we perform invariant checks for graph languages specified by annotated type graphs and discuss the correctness of the results.

All tests in this chapter were performed on a machine running a 64-bit version of Windows 10 on an Intel Xeon e3-1505m v6 processor and 16 GB RAM. All case studies are available in the source archive on the DrAGoM homepage.

12.1. Thesis Examples

To showcase correct results for the implemented base functionalities in DrAGoM, we use two examples from previous chapters, namely Examples 10.16 and 10.20, to compare the output with the formerly expected results.

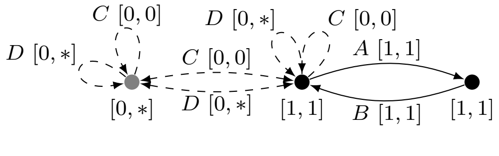
For all upcoming case studies in this chapter, we indicate rule morphisms by the numbers below the nodes. Furthermore, in some case studies we might omit edge labels to simplify the visual representation.

Abstract Rewriting Step for Example 10.16

Let the graph transformation rule ρ and the annotated type graph $T[\ell, u]$ over the label alphabet $\Lambda = \{A, B, C, D\}$ from Example 10.16 be given.

The computation of the rewritable materialization and the subsequent computation of the annotated type graph which specifies the strongest postcondition takes DrAGoM a few milliseconds. The output of the postcondition construction is the following annotated type graph $T'_\rho[a_1, a_2]$:

$T'[a_1, a_2] =$



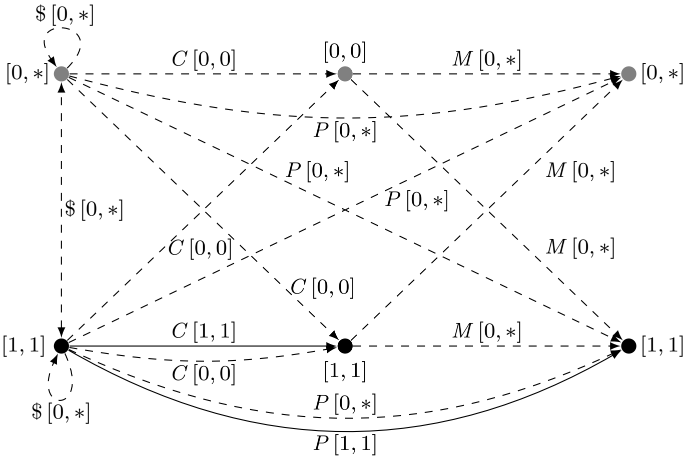
As evident from the picture above, we illustrate the concrete instance of the right-hand side graph R by using regular drawn edges and black nodes, whereas freshly introduced graph elements are drawn as gray colored nodes and bidirectional dashed edges. We will use these visual semantics for the remainder of this chapter.

The simplification of the computed annotated type graph $T'[a_1, a_2]$ coincides with the expected result from Example 10.16. Furthermore, the invariant check yields no positive answer and terminates within a few milliseconds, as the initial graph language does not contain graphs with A-labelled or B-labelled edges. The graph transformation rule and the annotated type graph of this case study are provided in the source archive on the DrAGoM homepage[1] as files named `SimpleExample.sgf` and `SimpleExample.gxl` respectively.

Invariant Check for Example 10.20

Let the graph transformation rule ρ and the annotated type graph $A[a_1, a_2]$ from Example 10.20 be given. The invariant check of DrAGoM yields no positive answer within a few milliseconds and shows the following strongest postcondition graph $T'_\rho[b_1, b_2]$ which only consists of one pair of multiplicities:

$T'_\rho[b_1, b_2] \quad =$



Please note that the simplification of $T'_\rho[b_1, b_2]$ coincides with the simplified result postulated in Example 10.20. Furthermore, thanks to DrAGoM, we can conclude that the pair of multiplicities (b_1, b_2) is the only maximal pair which conforms to Definition 10.15. The corresponding files of this case study are named `WorkedExample.sgf` and `WorkedExample.gxl`.

12.2. Invariant Check for Colorability

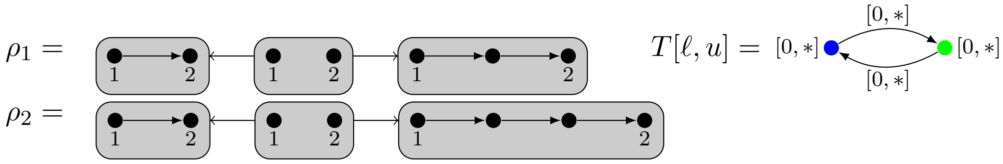
In this section we perform invariant checks for graph language examples inspired by [Blu14]. Type graphs are expressive enough to specify languages consisting of all k-colorable graphs, i.e., all graphs which can be colored using maximal k different colors, such that every pair of adjacent nodes never share the same color. Type graph languages can be simulated using annotated type graphs by annotating every element with the multiplicity pair $[0, *]$. Hence, we let DrAGoM perform an invariant check for 2-colorability and 3-colorability.

[1] DrAGoM homepage: https://www.uni-due.de/theoinf/research/tools_dragom.php

12.2.1. 2-Colorability with Path Extension

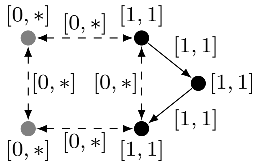
The 2-coloring problem can be solved in linear time. The main idea is to assign one of the two colors to a node v_1 and subsequently color all adjacent nodes with the opposite color. This process is repeated for the neighbours of v_1 until every node in the graph received a color. If no colored node has been assigned to both colors in this process, the graph is 2-colorable. It is easy to see that every 2-colorable graph is bipartite. Therefore, we can use the complete graph with two nodes as a type graph, to specify the set of 2-colorable graphs. Each node in the type graph resembles a partition of all nodes which share the same color.

Let the following graph transformation system $\mathcal{R} = \{\rho_1, \rho_2\}$ and the annotated type graph $T[\ell, u]$ over a label alphabet Λ with $|\Lambda| = 1$ be given. We omit the single edge label:

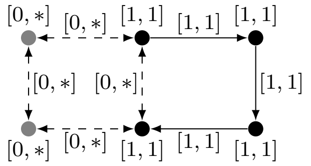


The invariant check takes a few milliseconds and yields no positive answer for the language of the annotated type graph $T[\ell, u]$ with respect to rule ρ_1. The annotated type graphs $T'_{\rho_1}[a_1, a_2]$ and $T'_{\rho_2}[b_1, b_2]$ (both computed by DrAGoM) specify the strongest postconditions for the respective rules. Both annotated type graphs contain only one pair of multiplicities since the annotation bounds $[0, *]$ are always the unique possible maximal pair of multiplicities:

$T'_{\rho_1}[a_1, a_2] =$



$T'_{\rho_2}[b_1, b_2] =$

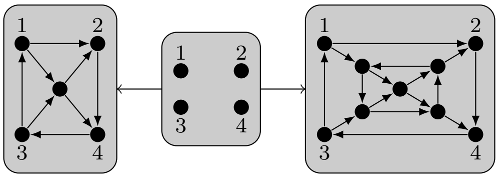


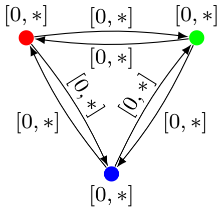
Clearly the annotated type graph $T'_{\rho_1}[a_1, a_2]$ is not bipartite (and hence not 2-colorable) due to the triangle subgraph on the right side. Performing an invariant check for rule ρ_2 only, yields a positive answer. DrAGoM is able to compute the legal morphism from $T'_{\rho_2}[b_1, b_2]$ to $T[\ell, u]$ within milliseconds. The graph transformation system and annotated type graph of this case study are provided as files named `2-Colorability.sgf` and `2-Colorability.gxl`.

12.2.2. 3-Colorability with Node Replacement

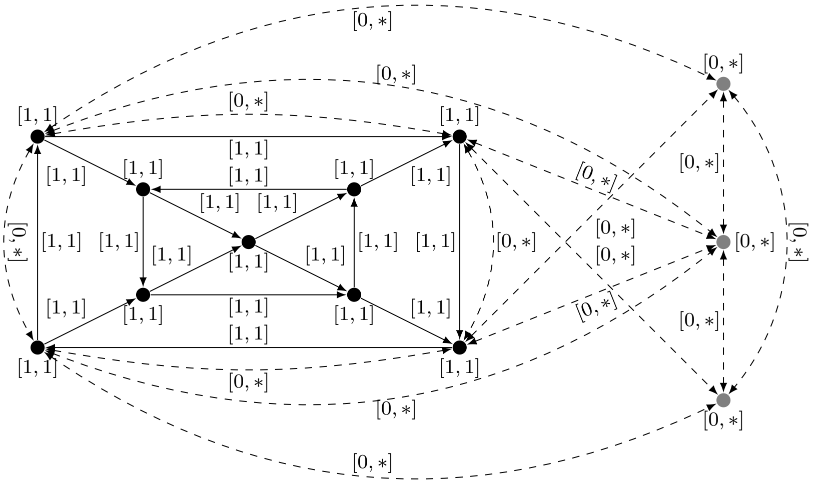
Next, we perform an invariant check for the language of an annotated type graph which consists of all 3-colorable graphs. The 3-coloring problem is known to be NP-complete. The corresponding type graph without annotations was already considered in Example 4.6 and coincides with the complete graph which consists of three nodes.

Let the following graph transformation rule ρ and the annotated type graph $T[\ell, u]$ be given. The rule morphisms are indicated by the numbers below the nodes. Similar to the 2-colorability case study we use a single edge label which we omit for the sake of simplicity:

$\rho =$ 

$T[\ell, u] =$ 

The invariant check finishes after a few milliseconds and yields a positive answer for the language of the annotated type graph $T[\ell, u]$ with respect to rule ρ. DrAGoM finds and investigates 6 different base morphisms and subsequently computes the strongest postconditions for each of them. In all cases the annotated type graph $T'_\rho[a_1, a_2]$ (shown below) which specifies the postcondition is the same graph up to isomorphism and it contains one pair of multiplicities:

$T'_\rho[a_1, a_2] =$ 

The outer nodes of the left-hand side graph of the rule can be colored with two alternating colors such that the third color remains for the inner node. Likewise, using the same 2-color scheme for the interface nodes of the right-hand side graph, we can color the inner rectangle with the complemented colors such that the third color again remains for the node in the center. Therefore, the graph language $\mathcal{L}(T[\ell, u])$ is indeed an invariant for the graph transformation rule ρ. The graph transformation system is provided in the file named `3-Colorability.sgf` and the file of the annotated type graph is named `3-Colorability.gxl`.

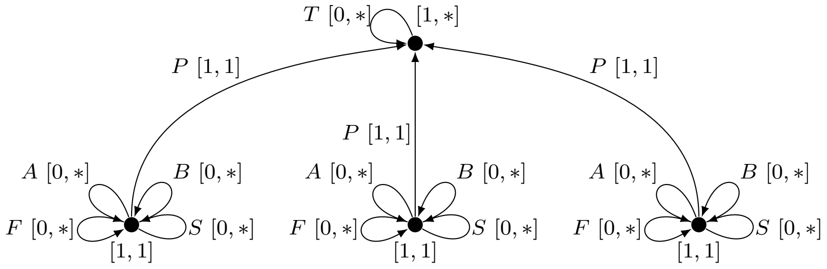
12.3. Invariant Check for a Rail System

We increase the number of rules, edge labels and possible base morphisms for our next case study. The following case study is inspired by [DG17]. We specify a rail system using an annotated type graph with the following edge label semantics:

T:	The **t**rack of the rail net.	**P**:	The **p**osition of a train.
F:	The train is **f**ast.	**A**:	The train is **a**ccelerating.
S:	The train is **s**low.	**B**:	The train is put on the **b**rakes.

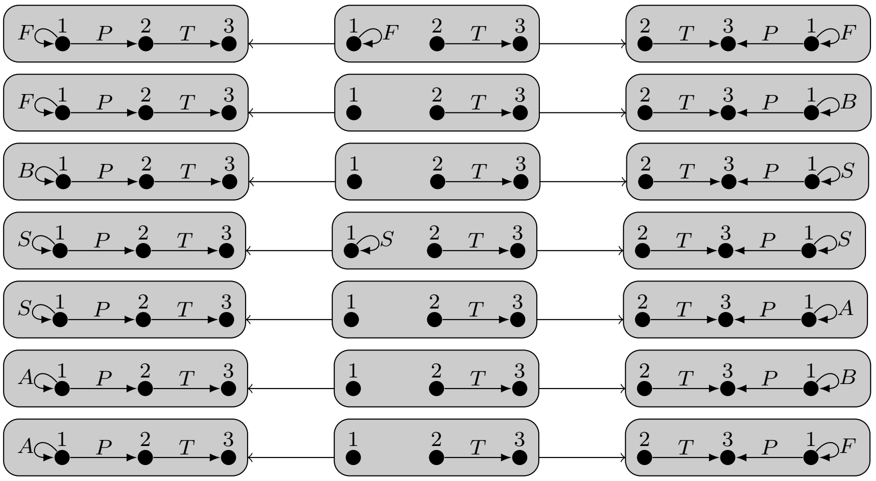
Now, we would like to model the following situation: Three trains are using the rail network which is infinitely large. Each train must be at one position and they can travel fast or slow. Furthermore each train can accelerate or brake to switch between these states. The graph language of this specification is specified by the following annotated abstract graph $T[\ell, u]$:

$T[\ell, u] =$



To specify the movement of the trains alongside their accelerations and brakes, we use the following graph transformation system $\mathcal{R} = \{\rho_i \mid 1 \le i \le 7\}$:

$\rho_1 =$
 $\rho_2 =$
 $\rho_3 =$
 $\rho_4 =$
 $\rho_5 =$
 $\rho_6 =$
 $\rho_7 =$



The rules in $\mathcal{R}$ have the following semantics: A fast train can travel over the tracks (ρ_1) or brake (ρ_2). While the train is put on brake it still travels over the tracks but becomes slow (ρ_3). Similar to a fast train, a slow train can travel over the tracks (ρ_4) or it accelerates (ρ_5). During the acceleration the train can brake again (ρ_6) or it can further accelerate to become fast (ρ_7).

The invariant check again finishes after a few milliseconds and yields a positive answer for the language of the annotated type graph $T[\ell, u]$ with respect to the graph transformation system $\mathcal{R}$. DrAGoM finds and investigates 3 different base morphisms for each of the 7 rules. In all cases the annotated type graph which specifies the postcondition consists of 9 annotations, 7 nodes and 48 edges.

Please note, that the main purpose of this case study was to increase the complexity of the invariant check, to stress test DrAGoM. Since the scenario does not forbid states where a train is fast, slow, accelerating and braking at the same time, the rule application boils down to a simple edge relabelling over multiple flower graphs of the specific train state loop labels A, B, F and S. Therefore, the graph transformation system clearly is an invariant.

However, if we restrict the states of the trains in the annotated type graph $T[\ell, u]$ (i.e. annotate the edges labelled A, B, F and S, by annotation bounds $[0, 1]$), the language is no longer an invariant for $\mathcal{R}$. The application of rule ρ_7 on a train which is already fast and accelerating would result in a graph which contains two F-labelled loops. Clearly such a graph is not part of the restricted language and DrAGoM is not able to prove the invariant for such a restricted scenario.

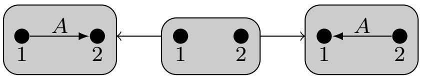
The source archive files for the graph transformation system and annotated type graph of this case study are named `RailTrack.sgf` and `RailTrack.gxl` respectively.

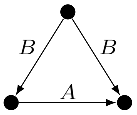
12.4. Invariant Check for Subgraph Containment

In this last case study we increase the size of the initial graph. The following example is inspired by [Blu14].

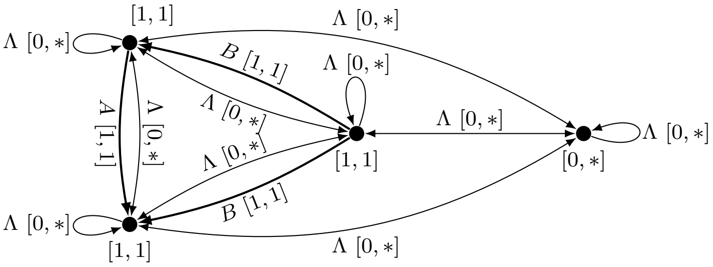
The framework of multiply annotated type graphs is expressive enough to specify graph languages which contain all graphs with a specific subgraph. For instance, an annotated materialization is able to specify exactly the set of graphs which contain a concrete image of the left-hand side of a rule. Therefore, we can use the materialization construction for any given graph S over the final object $T^{\Lambda}_{\circledast}$ of the category $\mathbf{Graph}_{\Lambda}$(cf. Example 4.4), to compute an annotated type graph which specifies the subgraph containment language, i.e. the set of all graphs which contain S as subgraph.

Let the following graph transformation rule ρ and the subgraph S over the edge label alphabet $\Lambda = \{A, B\}$ be given:

$\rho =$ 

$S =$ 

The materialization construction of S over the flower graph $T^{\{A,B\}}_{\circledast}$ takes DrAGoM a few milliseconds and results in the following annotated type graph $T[\ell, u]$ which specifies the initial subgraph containment language for S. Every Λ-labelled edge in the following graph represents a set of edges, one for each label in Λ:

$T[\ell, u] =$ 

The invariant check of $T[\ell, u]$ with respect to the rule ρ finishes after 4 hours 38 minutes 27 seconds and yields a positive answer. DrAGoM finds and investigates 14 different base morphisms and computes the strongest postcondition for each of them. It is worth to mention that the computation of the rewritable materialization and the strongest postcondition both take a few milliseconds. However, the inclusion check of the postcondition graphs into the initial annotated type graph $T[\ell, u]$ is very costly. A multiply annotated type graph which specifies the postcondition consists of 4 pairs of multiplicities, 6 nodes and 78 edges. Since the annotated type graph $T[\ell, u]$ consists of 4 nodes and 35 edges, DrAGoM has to check a huge number of possible embedding morphisms to prove that the languages are included.

Please note that DrAGoM employs the sufficient condition from Proposition 8.20 for the inclusion check, i.e., the tool tries to find a legal embedding morphism from the postcondition graph into the initial annotated type graph. However, finding a morphism from an input graph into another graph (even if the target graph is fixed) is NP-complete. DrAGoM uses a brute-force approach to find possible graph morphisms and checks their legality. Hence, it is not surprising that the inclusion check is the bottleneck in this approach.

The graph transformation system and annotated type graph of this case study are provided as files named `SubTriangle.sgf` and `SubTriangle.gxl`.

12.5. Overview of the Results

We summarize the results of the case studies which we conducted using DrAGoM. The following table shows the important key values for each experiment. Please note that the initial annotated type graph (ATG) $T[\ell, u]$ always started with one pair of multiplicities:

Case Study	GTS $\mathcal{R}$		ATG $T[\ell, u]$		Post ATG $T'[M]$			Result	
Name	$\lvert\Lambda\rvert$	$\lvert\mathcal{R}\rvert$	$\lvert V_T\rvert$	$\lvert E_T\rvert$	$\lvert V'_T\rvert$	$\lvert E'_T\rvert$	$\lvert M\rvert$	Invariant	Runtime
SimpleExample	4	1	1	2	3	10	1	✗	< 1s
WorkedExample	4	1	3	4	12	18	1	✗	< 1s
2-Colorability	1	2	2	2	5	10	1	✗	< 1s
3-Colorability	1	1	3	6	12	46	1	✓	< 1s
RailTrack	6	7	4	16	7	48	9	✓	< 1s
SubTriangle	2	1	4	35	6	78	4	✓	4h 38m

The generated outputs for all case studies conform to the expected results. The runtime results for invariant checking seem promising. However, as the last case study shows, the inclusion checks become too costly for larger initial graph sizes. This is due to the fact that a brute-force approach for the computation of legal morphisms is not efficient. Therefore, further exploration of the abstract state space yields no hope for efficiency unless the postcondition graphs can be simplified.

On the other hand, further tests on 100 randomly generated annotated type graphs (consisting of $10-20$ nodes, $10-40\%$ edge existence probability, minimum 10 multiplicities on $\mathcal{M}_n$ where $2 \leq n \leq 5$) showed that the SMT encodings for the computation of annotated rewritable materializations scales well. In one of these tests, an initial multiply annotated type graph using a single edge label consisted of 14 nodes, 72 edges and 64 multiplicities over $\mathcal{M}_2$. DrAGoM computed a rewritable materialization (for the single rule of the subgraph containment case study in Section 12.4) within 7 seconds. In the process, DrAGoM found 47 possible semi legal base morphisms to extract the left-hand side from. One of the computed rewritable materializations consisted of 16 nodes, 92 edges and 2048 annotations.

Therefore, we can conclude that the prototype tool DrAGoM shows that concrete instances of the general framework for abstract annotated objects can be implemented. The efficiency for computations of legal morphisms needs to be tweaked. Overall, DrAGoM can be used for strongest postcondition computations and for invariant checks of multiply annotated type graphs, up to sizes similar to the rail system case study of Section 12.3.

Part VI.

Conclusion

"If I'd had some set idea of a finish line, don't you think I would have crossed it years ago?"

Bill Gates (1955-present)

13 Conclusion and Future Work

The overall aim of this thesis was to analyse graph specification frameworks based on type graphs and show their applicability for verification techniques based on formal language theory. We first sum up the main contributions of the Parts II-IV and discuss how they fit into this aim. Overall, the results of the main parts can be summarized as follows:

Part II - Termination Analysis of Graph Transformation Systems

- A termination analysis approach for graph transformation systems based on weighted type graphs (Chapter 5).
- An interpretation for term rewriting systems to graph transformation systems which preserves termination arguments (Chapter 6).

Part III - Specifying Type Graph Languages

- An overview of closure and decidability properties for four graph specification frameworks based on type graphs (Chapters 7 and 8).

Part IV - Abstract Object Rewriting

- A construction for rewritable materializations for objects in an arbitrary topos (Chapter 9).
- A general framework to rewrite annotated objects and compute strongest postconditions (Chapter 10).

All these results originate from one simple idea: to use type graphs not only for typing purposes but also for the specification of graph languages. This change of perspective led to a basic graph specification framework which paved the path for several refinements. In this thesis, two of these refinements were successfully applied to verification techniques, i.e. weighted type graphs for termination analysis and multiply annotated type graphs for invariant checking. However, the key contribution of this thesis is a generalization of the type graph specification language to a framework of rewritable annotated objects. This framework can be instantiated to concrete specification languages for which strongest postconditions can be computed.

13.1. Summary and Conclusion

Detailed descriptions of related work and open questions were already provided in the conclusion chapters of the respective Parts II-IV. Therefore, in this section, we instead focus on the significance of the accomplished results. Furthermore, we discuss how these results fit into the broader scientific context and point out the possible research directions to where this thesis can lead.

In this thesis we focused on specification languages based on type graphs, where the language of a type graph T consists of all graphs that can be mapped homomorphically into T (with potentially extra constraints in refined frameworks). Many specification formalisms that are usually used in abstract graph transformation and verification, are based on type graphs. There exists no one-fits-all solution and different frameworks are suited well for different verification purposes. Inspired by matrix interpretations of string and cycle rewriting we first tried to adapt the basic idea of type graph languages to termination analysis for graph transformation systems.

Achievements in Termination Analysis

We describe the impact that the introduced termination approach and the encodings have on the automatic proving of termination, which has been a central topic in rewriting research over the last years.

In Part II of this thesis we introduced a new termination analysis technique for graph transformation systems based on the notion of weighted type graphs over different kinds of semirings. The weighted type graphs have the purpose to not only specify the language of all graphs, but also to assign weights to all graphs contained in this language. Then rules of the graph transformation system can be classified according to whether their application decreases the weight of every graph, or at least does not increase it. Please note that the weighted type graph approach does not subsume previous introduced termination analysis methods, but rather complements them. With respect to graph transformation systems, all variants of the termination problem, termination on all graphs as well as termination on a fixed set of initial graphs, are undecidable. Therefore, in practice one should always run several methods in parallel, when trying to prove the termination property of a rewriting system. Hence, by providing another termination technique, we contributed another piece to the bigger puzzle, to further increase chances for a successful termination analysis.

Furthermore, we studied how to transform term graph rewriting systems to graph transformation systems. Although it is widely agreed that term graph rewriting is the way to go for efficient implementation of term rewriting, quite surprisingly hardly any effort has been done in automatically proving termination of term graph rewriting. One reason could be that for term graph rewriting it is substantially harder since techniques strongly exploiting the term structure do not apply, such as path orders and dependency pairs. Being able to transform termination problems from term graph rewriting to graph transformation systems, our technique works subsequently for term rewriting systems as well. As a side effect, by applying our transformations to a selection of term rewriting systems from the Termination Problems Database, we provide a substantial set of test cases for automatically proving termination of graph transformation systems.

In the end, these results show us that it is worthwhile to build bridges between different kinds of rewriting systems to profit from established techniques. Verification techniques from the theory of formal languages are worked out very well in string and tree/term rewriting, but it is often non-trivial to use these techniques when it comes to graph rewriting. Therefore, it is natural to ask for generalizations of these verification techniques to the framework of graph rewriting and additionally to a theory of graph languages, where these techniques can be applied.

Graph Specification Frameworks based on Type Graphs

We tried to understand the essence of some selected graph specification languages, which grant them the possibility to use verification techniques from the theory of formal languages. In Part III, we classified several extensions of the basic type graph specification framework with respect to their decidability and closure properties. In order to be able to use type graph formalisms extensively in applications, it is necessary to provide a mechanism to compute weakest preconditions and strongest postconditions. That is, given a graph language and a set of rules, we want to specify the language of all successors (or the language of all predecessors) in our formalism. We found out that this is not feasible for pure type graphs, restriction graphs or the type graph logic, since neither formalism can count and hence can not express that all items of the newly created right-hand side occur exactly once. Hence, we characterized strongest postconditions in the setting of annotated type graphs. To cover specification frameworks based on type graphs with different annotations, we investigated the properties from a categorical point of view, thus we used notions from category theory to achieve general results.

A First Step Towards a General Framework

The key contribution of this thesis is the general framework introduced in Part IV. We have described a rewriting framework for abstract graphs that also applies to objects in any topos. In particular, we have given a construction for materialization in terms of the universal property of partial map classifiers in slice categories. This is a first theoretical milestone towards shape analysis as a general static analysis method for rule-based systems with graph-like objects as states. Soundness and completeness results for rewriting of abstract objects with annotations in an ordered monoid provide an effective verification method for the special case of graphs. Please note that this general framework is not to be confused with a one-fits-all solution. It is a blueprint from which instances of graph based specification languages, with computable postconditions, decidability of membership and closure under intersection, can be generated. The other properties depend on the used type of annotation. Furthermore, we have shown how annotated type graphs over multiplicities, one concrete instance of the general framework, can be implemented and used for verification purposes. The prototype tool DrAGoM witnesses the practicability of the level of abstraction with respect to invariant checking. However, the framework reaches its limits when it comes to further exploration of the state space. So far, the general framework does not provide techniques to collapse abstract objects derived via rewriting steps.

13.2. Future Work

Many interesting open questions connected to the individual contributions were already formulated in their respective chapters. Apart from these open questions, this thesis sets the starting point for a large set of possible next steps. The most interesting directions originate from the main contribution, i.e. the general framework for specification languages based on annotated type graphs.

First and foremost, the evaluation on DrAGoM showed how important it is to find widening or similar approximation techniques for the general framework, which collapse the abstract objects. Ideally, such techniques would lead to finite abstract transition systems that (over-)approximate the typically infinite transitions systems of graph transformation systems. We have already shown that the notion of cores does not apply to the framework of annotated type graphs over multiplicities.

If such an approximation technique is provided, DrAGoM can be efficiently applied to further verification techniques such as reachability analysis or non-termination analysis. The optimization and extension of DrAGoM would still serve the purpose to witness the practicability of further extensions. Among these optimizations, the computation of legal morphisms could be outsourced to SAT or SMT solvers. We showcased in [KNN18] how morphism computations (for core constructions) can efficiently be encoded using SAT and SMT solvers.

Another possible direction would be to further extend the general framework with respect to other types of annotations. The extension of annotations with logical formulas is just one possible next step, which could lead to a more flexible and versatile specification language. Furthermore, to build a bridge to applications used in practice, it is worth investigating how local multiplicities, which find use in UML, can be fully integrated into the general framework. For the practicability of the concrete instances it would be necessary to investigate how decidability and closure properties behave in such a refinement. Then it is also natural to ask which of the already established specification frameworks would end up as a concrete instance of such an extension.

Furthermore, since we managed to generalise specification frameworks based on annotated type graphs, it is worth to investigate if there exist other undiscovered generalisations for abstraction frameworks based on non graph-like structures. The take-home message of this thesis is: It is fine to generate ad-hoc solutions for specific problems - but from time to time one needs to take a higher perspective and try to organize the zoo of different established specification frameworks.

Part VII.

Appendix

A Proofs

A.1. Proofs of Chapter 5

Lemma 5.12. *Let S be an ordered commutative semiring and T a weighted type graph over S.*

(i) Whenever S is strongly ordered, for all graphs G, $\mathit{fl}_T(G)\colon G \to T$ exists and $w_T(\mathit{fl}_T(G)) > 0$.

(ii) Given the following diagram, where the square is a pushout and G_0 is discrete, it holds that $w_T(t) = w_T(t \circ \varphi_1) \otimes w_T(t \circ \varphi_2)$.

$$\begin{tikzcd}
& G_1 \arrow[dr, "\varphi_1"] & & \\
G_0 \arrow[ur, "\psi_1"] \arrow[dr, "\psi_2"'] & (\mathrm{PO}) & G \arrow[r, "t"] & T \\
& G_2 \arrow[ur, "\varphi_2"'] & &
\end{tikzcd}$$

Proof.

(i) $\mathit{fl}_T(G)$ exists by construction. Furthermore, since $w_T(e) > 0$ for all edges in the range of $\mathit{fl}_T(G)$, it holds that $\mathit{fl}_T(G) > 0$.

(ii) Since G_0 is discrete and the square is a pushout, the edge set E_G is (isomorphic to) the disjoint union of E_{G_1} and E_{G_2}. Thus:

$$\begin{aligned}
w_T(t) = \prod_{e \in E_G} w_T(t(e)) &= \prod_{e \in E_{G_1}} w_T((t \circ \varphi_1)(e)) \otimes \prod_{e \in E_{G_2}} w_T((t \circ \varphi_2)(e)) \\
&= w_T(t \circ \varphi_1) \otimes w_T(t \circ \varphi_2).
\end{aligned}$$

□

To prove Lemma 5.16 we give the following additional Lemma L.1.

Lemma L.1. Let a pushout PO consisting of objects G_0, G_1, G_2, G be given. Then there exists a bijection between pairs of commuting morphisms $t_1\colon G_1 \to T$, $t_2\colon G_2 \to T$ and morphisms $t\colon G \to T$ (see diagram to the right). For each t we obtain a unique pair of morphisms t_1, t_2 by composing with φ_1 and φ_2, respectively. Conversely, for each pair t_1, t_2 of morphisms with $t_1 \circ \psi_1 = t_2 \circ \psi_2$ we obtain a unique $t\colon G \to T$ as mediating morphism. In this case we will write $\mathit{med}_{PO}(t_1, t_2) = t$ and $\mathit{med}_{PO}^{-1}(t) = \langle t_1, t_2 \rangle$.

$$\begin{tikzcd}
& G_1 \arrow[dr, "\varphi_1"] \arrow[drr, bend left, "t_1"] & & \\
G_0 \arrow[ur, "\psi_1"] \arrow[dr, "\psi_2"'] & (\mathrm{PO}) & G \arrow[r, "t"] & T \\
& G_2 \arrow[ur, "\varphi_2"'] \arrow[urr, bend right, "t_2"'] & &
\end{tikzcd}$$

Proof. It is straightforward to verify that med_{PO} and med_{PO}^{-1} are indeed inverse to each other and hence both are bijections. □

Lemma 5.16. *Let S be a strictly ordered commutative semiring and T a weighted type graph over S. Furthermore, let ρ be a rule such that $G \Rightarrow_\rho H$.*

(i) If ρ is non-increasing, then $w_T(G) \geq w_T(H)$.

(ii) If ρ is decreasing, then $w_T(G) > w_T(H)$.

Proof. Let $\rho = L \leftarrow\!\varphi_L\!- I -\!\varphi_R\!\rightarrow R$. The rewriting step $G \Rightarrow_\rho H$ is depicted below on the left.

For every possibility to type G via $t_G\colon G \to T$, there exists a morphism $t_C = t_G \circ \psi_L\colon C \to T$ and we obtain $t_H\colon H \to T$ as mediating morphism of the right-hand pushout PO_2 (see diagram on the right).

$$\begin{tikzcd}
L \arrow[d, "m"'] & I \arrow[l, "\varphi_L"'] \arrow[r, "\varphi_R"] \arrow[d, "c"'] & R \arrow[d, "n"'] \\
G & C \arrow[l, "\psi_L"'] \arrow[r, "\psi_R"] & H
\end{tikzcd}
\qquad
\begin{tikzcd}
L \arrow[d, "m"'] \arrow[dr, phantom, "PO_1"] \arrow[ddr, bend right=60, "t_L"'] & I \arrow[l, "\varphi_L"'] \arrow[r, "\varphi_R"] \arrow[d, "c"'] \arrow[dr, phantom, "PO_2"] & R \arrow[d, "n"'] \arrow[ddl, bend left=60, "t_R"] \\
G \arrow[dr, "t_G"'] & C \arrow[l, "\psi_L"'] \arrow[r, "\psi_R"] \arrow[d, "t_C"] & H \arrow[dl, dashed, "t_H"] \\
& T &
\end{tikzcd}$$

Now we have (compare with the diagram above on the right):

$$\begin{aligned}
w_T(G) &= \sum_{t_G\colon G\to T} w(t_G) && \text{(A.1)}\\
&= \sum_{t_C\colon C\to T} \sum_{\substack{t_L\colon L\to T\\ t_L\circ\varphi_L = t_C\circ c}} w(med_{PO_1}(t_C, t_L)) && \text{(A.2)}\\
&= \sum_{t_C\colon C\to T} \sum_{\substack{t_L\colon L\to T\\ t_L\circ\varphi_L = t_C\circ c}} (w(t_L) \otimes w(t_C)) && \text{(A.3)}\\
&= \sum_{t_C\colon C\to T} \Big(w(t_C) \otimes \sum_{\substack{t_L\colon L\to T\\ t_L\circ\varphi_L = t_C\circ c}} w(t_L) \Big) && \text{(A.4)}\\
&= \sum_{t_C\colon C\to T} (w(t_C) \otimes w_{t_C\circ c}(\varphi_L)) && \text{(A.5)}
\end{aligned}$$

where
(A.2) follows from the fact that *med* is a bijection (see Lemma L.1),
(A.3) is an application of the equation $w(t) = w(t\circ\varphi_1) \otimes w(t\circ\varphi_2)$ of Lemma 5.12,
(A.4) follows from distributivity and
(A.5) holds by definition. Symmetrically, we have

$$w_T(H) = \sum_{t_C\colon C\to T} (w(t_C) \otimes w_{t_C\circ c}(\varphi_R)).$$

Using this, we can prove the two parts of the lemma.

(i) Since ρ is non-increasing, it holds by definition that $w_{t_C\circ c}(\varphi_L) \geq w_{t_C\circ c}(\varphi_R)$ for all $t_C\colon C \to T$, and thus $w(t_C) \otimes w_{t_C\circ c}(\varphi_L) \geq w(t_C) \otimes w_{t_C\circ c}(\varphi_R)$. From this it follows that $w_T(G) \geq w_T(H)$.

(ii) Since ρ is decreasing, and thus non-increasing, it holds by definition that $w_{t_C \circ c}(\varphi_L) \geq w_{t_C \circ c}(\varphi_R)$ for all $t_C \colon C \to T$. Additionally, it holds by assumption that $w_{\mathit{fl}(I)}(\varphi_L) > w_{\mathit{fl}(I)}(\varphi_R)$. Since $w_T(\mathit{fl}(C)) > 0$ by definition, and S is a strictly ordered semiring, we have that

$$w_T(\mathit{fl}(C)) \otimes w_{\mathit{fl}(I)}(\varphi_L) > w_T(\mathit{fl}(C)) \otimes w_{\mathit{fl}(I)}(\varphi_R)$$

(we take $t_C = \mathit{fl}(C)$ and $t_C \circ c = \mathit{fl}(I)$). From these two facts it follows that $w_T(G) > w_T(H)$ (again using the fact that S is a strictly ordered semiring).

□

Theorem 5.17 (Relative termination based on strictly ordered semirings). *Let S be a strictly ordered commutative semiring with a well-founded order $\leq$ and T a weighted type graph over S. Let R be a set of graph transformation rules, partitioned in two sets $R^<$ and $R^=$. If all rules of $R^<$ are decreasing and all rules of $R^=$ are non-increasing then R is terminating if and only if $R^=$ is terminating.*

Proof.
$\Rightarrow$: It is an immediate consquence of R being terminating that its subset $R^=$ is also terminating.
$\Leftarrow$: For a rule ρ and transition $G \Rightarrow_r H$, it holds that $w_T(G) > w_T(H)$ if $\rho \in R^<$ and $w_T(G) \geq w_T(H)$ if $\rho \in R^=$ (by Lemma 5.16). From this it follows that each infinite transition sequence of R ends in an infinite transition sequence of $R^=$, which do not exist by assumption. □

Lemma 5.19. *Let S be a strongly ordered commutative semiring and T a weighted type graph over S. Furthermore, let ρ be a rule such that $G \Rightarrow_\rho H$.*

(i) If ρ is non-increasing, then $w_T(G) \geq w_T(H)$.

(ii) If ρ is strongly decreasing, then $w_T(G) > w_T(H)$.

Proof. The proof proceeds analogously to the proof of Lemma 5.16. For non-increasing rules the proof is exactly the same.

For strongly decreasing rules we have to show

$$\sum_{t_C \colon C \to T} (w(t_C) \otimes w_{t_C \circ c}(\varphi_L)) > \sum_{t_C \colon C \to T} (w(t_C) \otimes w_{t_C \circ c}(\varphi_R))$$

This holds since $w_{t_C \circ c}(\varphi_L) > w_{t_C \circ c}(\varphi_R)$ for all t_C, hence the properties of strongly ordered semirings allow us to conclude. □

A.2. Proofs of Chapter 6

Theorem 6.21 (Termination argument via function encoding). *Let P be a set of term graph productions and let $\rho = L \leftarrowtail_\ell I \rightarrow_r R$ be a production from P. Let G, H be term graphs such that $G \Rightarrow_\rho H$, then $F(G) \Rightarrow_{F(\rho)} F(H)$. Hence, if $\{F(\rho) \mid \rho \in P\}$ is terminating, then P is terminating as well.*

Proof. The only effect of applying $F(\rho)$ is that the numbered outgoing edges of a node labeled by f of arity n, which are labeled by $f_1, \dots, f_n$, and the label of the node is removed. By doing this operation on every node, the term graph rewrite step $G \Rightarrow_\rho H$ can be fully mimicked by $F(G) \Rightarrow_{F(\rho)} F(H)$. This is due to the fact that ρ, due to our restrictions on rules, will only add a label and successor nodes if the previous label and successor nodes have been removed. Hence no unification and non-local effects take place.

Hence we obtain that any infinite reduction of a term graph transforms by F to an infinite reduction of graph transformation steps. □

Theorem 6.24 (Termination argument via basic number encoding). *Let P be a set of term graph productions, obtained by encoding term rewrite rules, all having an interface which consists only of the root and variable nodes. Then P is terminating on term graphs if and only if the graph transformation system $N(P) = \{N(\rho) \mid \rho \in P\}$ is terminating.*

Proof. (sketch) Let $(V, \mathsf{lab}, \mathsf{succ})$ be a term graph and $\rho = L \leftarrowtail_\ell I \rightarrow_r R$ be a production. Let G, H be term graphs such that $G \Rightarrow_\rho H$. Then $N(G) \Rightarrow_{N(\rho)} N(H)$, similar to the argument in the proof of Theorem 6.21. So an infinite P-reduction on term graphs gives rise to an infinite $N(P)$-reduction on graphs.

Conversely, assume there is an infinite $N(P)$-reduction on graphs. In the original graph mark every edge that is eventually part of a match with a left-hand side in some step of the reduction. As non-marked edges are never touched, they stay forever in the reduction and may be removed, while the rest is still part of an infinite reduction. Next remove the finite initial part containing marked edges that have not yet been rewritten.

Hence in the remaining infinite reduction all edges are created by the replacement of a left-hand side by a right-hand side. As all edges in right-hand sides satisfy the following properties, the same holds for any graph in the remaining infinite reduction:

- for every edge from v to w labeled by a symbol f of arity unequal to 1, the set of incoming edges of w only consists of this single edge, and the set of outgoing edges from w consists of exactly $\mathsf{ar}(f)$ edges, labeled $1, 2, \dots, \mathsf{ar}(f)$ (hence no edges if $\mathsf{ar}(f) = 0$), and
- for every edge from v to w labeled by a number i, the set of incoming edges of v only consists of a single edge labeled by a symbol f, and the set of outgoing edges from v is exactly $1, 2, \dots, \mathsf{ar}(f)$.

Now, there may still be nodes that have more than one outgoing edge labeled by (function) symbols. This can be avoided by the following argument: let the weight of a node with k symbol-labeled outgoing edges be $k - 1$ if $k > 0$, otherwise it is 0. Let the weight of a graph be the sum of the weight of its nodes. Take a graph with infinite reduction with minimal weight. If this weight is strictly larger than 0, then there is a node with $k > 1$ outgoing symbol-labeled edges. Due to the shape of the rules, during the whole infinite reduction this node preserves k outgoing symbol-labeled edges, and hence matches left-hand sides only at the root or at a variable. Hence we still get an infinite reduction if we add a fresh node without ingoing edges, taking over one of the outgoing symbol-labeled outgoing edges from our special node. But now the adjusted graph has a lower weight, contradicting minimality.

Hence the graph with infinite reduction with minimal weight has weight 0, hence it is the number encoding of a corresponding term graph. After these adjustments, we obtain an infinite rewriting sequence of graphs, where all graphs will be of the shape $N(G)$ for a term graph G, and every reduction step is of the shape $N(G') \Rightarrow_{N(\rho)} N(H')$. This gives rise to a P-reduction of G. □

A.3. Proofs of Chapter 7

Proposition 7.5 (Decidability results for type/restriction graph languages). *For a graph language $\mathcal{L}$ characterized by a type graph T (i.e. $\mathcal{L} = \mathcal{L}(T)$) or by a restriction graph R (i.e. $\mathcal{L} = \mathcal{L}_{\mathtt{R}}(R)$) the following problems are decidable:*

1. *Membership, i.e. for each graph G it is decidable if $G \in \mathcal{L}$ holds.*

2. *Emptiness, i.e. it is decidable if $\mathcal{L} = \emptyset$ holds.*

Furthermore, language inclusion is decidable for both classes of languages:

3. *Given type graphs T_1 and T_2, $\mathcal{L}(T_1) \subseteq \mathcal{L}(T_2)$ holds iff $T_1 \to T_2$.*

4. *Given restriction graphs R_1 and R_2, $\mathcal{L}_{\mathtt{R}}(R_1) \subseteq \mathcal{L}_{\mathtt{R}}(R_2)$ holds iff $R_1 \to\ R_2$.*

Proof.

1. To decide whether $G \in \mathcal{L}(T)$ (or $G \in \mathcal{L}_{\mathtt{R}}(R)$) holds, we need to check for the existence of a morphism $\varphi\colon G \to T$ (or for the non-existence of a morphism $\varphi\colon R \to G$), which is obviously possible because graphs are finite. Nevertheless, note that this problem is NP-complete. For instance, searching for a morphism from any graph into the 3-clique is equivalent to deciding if the graph is 3-colourable.

2. The emptiness problem is almost trivial. If $\mathcal{L} = \mathcal{L}(T)$ for a type graph T, then $\mathcal{L}(T) \neq \emptyset$ because it holds $\varnothing \in \mathcal{L}(T)$ (recall that $\varnothing$ is the initial object of $\mathbf{Graph}_\Lambda$).

   If instead $\mathcal{L} = \mathcal{L}_{\mathtt{R}}(R)$ for a restriction graph R, then $\mathcal{L} = \emptyset$ if and only if $R = \varnothing$. In fact, if $R = \varnothing$ then $R \to G$ for all $G \in \mathbf{Graph}_\Lambda$, and thus $\mathcal{L}_{\mathtt{R}}(R) = \emptyset$. Instead if $R \neq \varnothing$ then clearly $R \nrightarrow \varnothing$, thus $\varnothing \in \mathcal{L}_{\mathtt{R}}(R) \neq \emptyset$.

3. We show that $\mathcal{L}(T_1) \subseteq \mathcal{L}(T_2)$ iff $T_1 \to T_2$, which is decidable.
   $\Rightarrow$: Assume $\mathcal{L}(T_1) \subseteq \mathcal{L}(T_2)$ holds. Since $T_1 \in \mathcal{L}(T_1)$ holds then $T_1 \in \mathcal{L}(T_2)$ also holds and therefore $T_1 \to T_2$.
   $\Leftarrow$: Assume $T_1 \to T_2$ holds, and let $G \in \mathcal{L}(T_1)$. Therefore $G \to T_1$, and by transitivity $G \to T_2$, thus $G \in \mathcal{L}(T_2)$.

4. We show that $\mathcal{L}_{\mathtt{R}}(R_1) \subseteq \mathcal{L}_{\mathtt{R}}(R_2)$ iff $R_1 \to R_2$.
   $\Rightarrow$: Assume that $\mathcal{L}_{\mathtt{R}}(R_1) \subseteq \mathcal{L}_{\mathtt{R}}(R_2)$ holds. Equivalently, we obtain that $\{G \mid R_2 \to G\} = \overline{\mathcal{L}_{\mathtt{R}}(R_2)} \subseteq \overline{\mathcal{L}_{\mathtt{R}}(R_1)} = \{G \mid R_1 \to G\}$, where we wrote $\overline{\mathcal{L}}$ for the complement language ($|\mathbf{Graph}_\Lambda| \setminus \mathcal{L}$). Therefore, since obviously $R_2 \to R_2$, we obtain $R_1 \to R_2$.
   $\Leftarrow$: Assume that $R_1 \to R_2$ holds and that $G \in \mathcal{L}_{\mathtt{R}}(R_1)$, which means $R_1 \nrightarrow G$. If $G \notin \mathcal{L}_{\mathtt{R}}(R_2)$, then we have $R_2 \to G$ and, by transitivity, $R_1 \to G$, which is a contradiction.

□

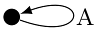
Proposition 7.6 (Closure properties of type/restriction graph languages). *Type graph languages are closed under intersection (by taking the product of type graphs) but not under union or complement.*

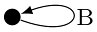
Restriction graph languages are closed under union (by taking the coproduct of restriction graphs) but not under intersection or complement.

Proof. From the universal property of the product $T_1 \times T_2$, it follows that for any graph G we have $G \to T_1 \times T_2$ if and only if $G \to T_1$ and $G \to T_2$. Hence, given two type graphs T_1 and T_2 we get immediately the following equality: $\mathcal{L}(T_1) \cap \mathcal{L}(T_2) = \mathcal{L}(T_1 \times T_2)$.

Dually, given two restriction graphs R_1 and R_2, we now show that we obtain $\mathcal{L}_{\mathtt{R}}(R_1) \cup \mathcal{L}_{\mathtt{R}}(R_2) = \mathcal{L}_{\mathtt{R}}(R_1 \oplus R_2)$, where $\oplus$ denotes coproduct in $\mathbf{Graph}_\Lambda$. In fact, $G \notin \mathcal{L}_{\mathtt{R}}(R_1 \oplus R_2)$ iff $R_1 \oplus R_2 \to G$ iff (by the universal property of coproducts) $R_1 \to G$ and $R_2 \to G$ iff $G \notin \mathcal{L}_{\mathtt{R}}(R_1)$ and $G \notin \mathcal{L}_{\mathtt{R}}(R_2)$ iff $G \notin \mathcal{L}_{\mathtt{R}}(R_1) \cup \mathcal{L}_{\mathtt{R}}(R_2)$.

For the negative results, we will show counterexamples using the following graphs over $\Lambda = \{A, B\}$:

$G_A =$ 

$G_B =$ 

First, we show by contradiction that type graph languages are not closed under union. Let the two type graph languages $\mathcal{L}(G_A)$ and $\mathcal{L}(G_B)$ be given. Assume that there exists a type graph T such that $\mathcal{L}(T) = \mathcal{L}(G_A) \cup \mathcal{L}(G_B)$. The type graph language $\mathcal{L}(G_A)$ contains all graphs which do not have any B-labelled edge, and $\mathcal{L}(G_B)$ contains all graphs which do not have any A-labelled edge. Since $G_A, G_B \in \mathcal{L}(G_A) \cup \mathcal{L}(G_B)$, we would have $G_A \to T$ and $G_B \to T$, and since type graph languages are closed under coproduct $\oplus$ this implies that there exists a morphism $G_A \oplus G_B \to T$, i.e. $G_A \oplus G_B \in \mathcal{L}(T)$. However, $G_A \oplus G_B$ is neither in $\mathcal{L}(G_A)$ nor in $\mathcal{L}(G_B)$, thus yielding a contradiction.

Now we show by contradiction that there is no restriction graph R such that $\mathcal{L}_{\mathtt{R}}(R) = \mathcal{L}_{\mathtt{R}}(G_A) \cap \mathcal{L}_{\mathtt{R}}(G_B)$. In fact, if such an R exists then $\mathcal{L}_{\mathtt{R}}(R) \subseteq \mathcal{L}_{\mathtt{R}}(G_A)$, and thus $R \to G_A$ by Proposition 7.5(4), and $\mathcal{L}_{\mathtt{R}}(R) \subseteq \mathcal{L}_{\mathtt{R}}(G_B)$, and thus $R \to G_B$. But $R \to G_A$ means that R has no B-edges, and $R \to G_B$ that it has no A-edges, thus R must be discrete. This implies that $G_A \not\to R$ and $G_B \not\to R$, i.e. $R \in \mathcal{L}_{\mathtt{R}}(G_A) \cap \mathcal{L}_{\mathtt{R}}(G_B)$, but clearly $R \notin \mathcal{L}_{\mathtt{R}}(R)$ yielding a contradiction.

The lack of closure under complement immediately follows from these negative results and the fact that union can be expressed using intersection and complement, and dually. □

Proposition 7.7 (Closure under DPO rewriting for restriction graphs). *A restriction graph language $\mathcal{L}_{\mathtt{R}}(S)$ is closed under a rule $\rho = (L \xleftarrow{\varphi_L} I \xrightarrow{\varphi_R} R)$ if and only if the following condition holds: for every pair of morphisms $\alpha\colon R \to F$, $\beta\colon S \to F$ which are jointly surjective, all graphs E that we obtain by applying the rule ρ with (co-)match α backwards to F, satisfy $S \to E$.*

$$\begin{tikzcd}
L \arrow[d] & I \arrow[l, "\varphi_L"'] \arrow[r, "\varphi_R"] \arrow[d] & R \arrow[r, dash, dashed] \arrow[d, "\alpha"'] \arrow[dll, dashed] & S \arrow[dl, "\beta"] \\
E & C' \arrow[l] \arrow[r] & F &
\end{tikzcd}$$

Proof.
$\Leftarrow$: Let G, H with $G \Rightarrow_\rho H$. By contraposition we show that $H \notin \mathcal{L}_{\mathtt{R}}(S)$ implies $G \notin \mathcal{L}_{\mathtt{R}}(S)$.

Since $G \Rightarrow_\rho H$ we have the following DPO diagram (below, on the left). Furthermore, since $H \notin \mathcal{L}_{\mathtt{R}}(S)$, there exists a morphism $\beta'\colon S \to H$. Now take the joint image F of R and S in H, i.e., factor the morphisms α', β' into $R \to F \rightarrowtail H$ and $S \to F \rightarrowtail H$, where the arrows $R \to F$, $S \to F$ are jointly surjective.[1]

[1] This is also known as pair factorization, see for instance [EE+06].

$$\begin{tikzcd}
L \arrow[d] & I \arrow[l, "\varphi_L"'] \arrow[r, "\varphi_R"] \arrow[d] & R \arrow[d, "\alpha'"'] & S \arrow[dl, "\beta'"] \\
G & C \arrow[l] \arrow[r] & H &
\end{tikzcd}
\qquad
\begin{tikzcd}
L \arrow[d] & I \arrow[l, "\varphi_L"'] \arrow[r, "\varphi_R"] \arrow[d] & R \arrow[d, "\alpha"'] & S \arrow[dl, "\beta"] \arrow[dlll, dashed] \\
E \arrow[d, tail] & C' \arrow[l] \arrow[r] \arrow[d, tail] & F \arrow[d, tail] & \\
G & C \arrow[l] \arrow[r] & H &
\end{tikzcd}$$

Since we are working in a topos, the pushouts split into pushouts (see the proof of Lemma 4.6 in [LS05] in combination with Corollary 9.5 in [Hei09], i.e. the fact that pushouts are stable under pullback in any topos). This is depicted in the diagram above on the right. Now, E is obtained from F by applying rule ρ backwards. Hence, the condition implies that there exists a morphism $S \to E$ and this means that there is a morphism $S \to G$, which implies $G \notin \mathcal{L}_{\mathtt{R}}(S)$.

$\Rightarrow$: Assume that $\mathcal{L}_{\mathtt{R}}(S)$ is closed under rewriting via a rule ρ. We show that the condition holds. Let $\alpha\colon R \to F$, $\beta\colon S \to F$ be a pair of morphisms which are jointly surjective and assume that E is obtained from F by applying ρ backwards.

Now, since $E \Rightarrow_\rho F$ and $F \notin \mathcal{L}_{\mathtt{R}}(S)$, we infer that $E \notin \mathcal{L}_{\mathtt{R}}(S)$, otherwise we would have a counterexample to closure under rewriting. Hence there exists a morphism $S \to E$. □

To prove Proposition 7.8 we recall the following lemma from [NT00].

Lemma L.2 (Lemma 2.1 of [NT00]). Let T be a graph and $\mathit{core}(T)$ be a core of T. Then for each morphism $f\colon T \to \mathit{core}(T)$ there exists a morphism $f'\colon \mathit{core}(T) \to T$ such that $f \circ f' = id_{\mathit{core}(T)}$. Vice versa, for each morphism $g'\colon \mathit{core}(T) \to T$ there exists a morphism $g\colon T \to \mathit{core}(T)$ such that $g \circ g' = id_{\mathit{core}(T)}$.

Proof. Let $f\colon T \to \mathit{core}(T)$ and $g'\colon \mathit{core}(T) \to T$ be arbitrary morphisms. Then, $h = f \circ g'$ is an automorphism of $\mathit{core}(T)$. Therefore, the morphisms $f' = g' \circ h^{-1}$ and $g = h^{-1} \circ f$ satisfy $f \circ f' = g \circ g' = id_{\mathit{core}(T)}$. □

Proposition 7.8 (Closure under DPO rewriting for type graphs). *A type graph language $\mathcal{L}(T)$ is closed under a rule $\rho = (L \xleftarrow{\varphi_L} I \xrightarrow{\varphi_R} R)$ if and only if for each morphism $t_L\colon L \to \mathit{core}(T)$ there exists a morphism $t_R\colon R \to \mathit{core}(T)$ such that $t_L \circ \varphi_L = t_R \circ \varphi_R$, that is:*

$$\mathcal{L}(T) \textit{ is closed under application of } \rho \Leftrightarrow
\begin{tikzcd}
& \overbrace{\hphantom{L \longleftarrow I \longrightarrow R}}^{\rho} & \\
L \arrow[dr, "\forall t_L"'] & I \arrow[l, "\varphi_L"'] \arrow[r, "\varphi_R"] & R \arrow[dl, "\exists t_R"] \\
& \mathit{core}(T) &
\end{tikzcd}$$

Proof.
$\Rightarrow$: Let $t_L\colon L \to \mathit{core}(T)$, and let $n : I \to \mathit{core}(T)$ be defined as $n = t_L \circ \varphi_L$. Consider the diagram below to the left, where the top span is the rule, and the two squares are built as pushouts (A is the pushout of φ_L and n; B is the pushout of φ_R and n).

Arrow $A \to \mathit{core}(T)$ is uniquely determined because the left square is a pushout and $id \circ n = t_L \circ \varphi_L$. This arrow witnesses that $A \in \mathcal{L}(T)$ (because $\mathit{core}(T) \to T$), and thus by assumption $B \in \mathcal{L}(T)$, because obviously $A \Rightarrow_\rho B$. Therefore we know that $B \to T$, and thus that there is an arrow $g : B \to \mathit{core}(T)$. In general, this arrow does not make the lower right triangle commute, but given that we also have an arrow $f : \mathit{core}(T) \to B$ as the base of the right pushout, it follows that $B \sim \mathit{core}(T)$ and hence $\mathit{core}(B) \cong \mathit{core}(\mathit{core}(T)) = \mathit{core}(T)$. Therefore by Lemma L.2, we know that there is an arrow $g' : B \to \mathit{core}(T)$ such that $g' \circ f = id_{\mathit{core}(T)}$ (in particular, $g' = (g \circ f)^{-1} \circ g$). Therefore, in the diagram below to the left also the lower right triangle commutes, and arrow $t_R = g' \circ m : R \to \mathit{core}(T)$ satisfies $t_L \circ \varphi_L = t_R \circ \varphi_R$.
$\Leftarrow$: Assume that $G \in \mathcal{L}(T)$ witnessed by morphism $t_G : G \to T$, and that G is rewritten to H via rule $\rho = (L \leftarrow\!\varphi_L\!- I -\!\varphi_R\!\rightarrow R)$. Also, let t be any arrow from T to $\mathit{core}(T)$. This gives us the diagram below right, where the two squares are pushouts, and the left triangle commutes by taking for $C \to \mathit{core}(T)$ the composition $t \circ t_G \circ \psi_L$.

$$\begin{tikzcd}
L \arrow[d] \arrow[ddr, bend right=60, "t_L"'] & I \arrow[l, "\varphi_L"'] \arrow[r, "\varphi_R"] \arrow[d, "n"'] & R \arrow[d, "m"'] \\
A \arrow[dr] & \mathit{core}(T) \arrow[l] \arrow[r, "f"] \arrow[d, "id"'] & B \arrow[dl, "g'"] \\
& \mathit{core}(T) &
\end{tikzcd}
\qquad
\begin{tikzcd}
L \arrow[d, "m"'] & I \arrow[l, "\varphi_L"'] \arrow[r, "\varphi_R"] \arrow[d, "n"'] & R \arrow[d, "m'"'] \arrow[ddl, dashed, bend left=60, "t_R"] \\
G \arrow[dr, "t \circ t_G"'] & C \arrow[l, "\psi_L"'] \arrow[r, "\psi_R"] \arrow[d] & H \arrow[dl, "t_H"'] \\
& \mathit{core}(T) &
\end{tikzcd}$$

By assumption, since $t \circ t_G \circ m \colon L \to \mathit{core}(T)$, there exists $t_R \colon R \to \mathit{core}(T)$ such that $t \circ t_G \circ m \circ \varphi_L = t_R \circ \varphi_R$. This means that the square consisting of $I, C, R, \mathit{core}(T)$ commutes, that is $t \circ t_G \circ \psi_L \circ n = t \circ t_G \circ m \circ \varphi_L = t_R \circ \varphi_R$. Hence there exists a mediating morphism $t_H \colon H \to \mathit{core}(T)$, which implies $H \in \mathcal{L}(T)$ because $\mathit{core}(T) \to T$. □

Proposition 7.14 (Closure properties of TGL). *Graph languages $\mathcal{L}(F)$ characterized by a TGL formula F, are closed under union, intersection and complement.*

Proof. Boolean closure properties come for free, due to boolean connectives. □

Proposition 7.15 (Decidability properties of TGL). *For a graph language $\mathcal{L}(F)$ characterized by a TGL formula F, the following problems are decidable:*

- *Membership, i.e. for all graphs G it is decidable if $G \in \mathcal{L}(F)$ holds.*
- *Emptiness, i.e. it is decidable if $\mathcal{L}(F) = \emptyset$ holds.*
- *Language inclusion, i.e. given two TGL formulas F_1 and F_2 it is decidable if $\mathcal{L}(F_1) \subseteq \mathcal{L}(F_2)$ holds.*

Proof.
Membership:
The membership problem for graph languages over *TGL* formulae is decidable since it is decidable for every type graph language $\mathcal{L}(T)$. We simply build the syntax tree of the formula F and search for morphisms $\varphi_i \colon G \to T_i$ at the leaves of

the tree. Afterwards we pass the boolean results up to the root to decide whether $G \in \mathcal{L}(F)$ holds.

Emptiness:
In order to show whether $\mathcal{L}(F) = \emptyset$ holds, we transform F into disjunctive normal form (DNF). It is sufficient to check whether all conjunctions of the form $(T_0 \wedge \neg T_1 \wedge \cdots \wedge \neg T_n)$ are unsatisfiable. We can assume that there is at most one positive type graph in every conjunction, since type graph languages are closed under conjunction/intersection (see Proposition 7.6). Furthermore we can even assume that there is exactly one positive type graph, since we can always add $T_{\circledast}^{\Lambda}$ (the flower graph).

Now we have:

$$\begin{aligned}
& \mathcal{L}(T_0 \wedge \neg T_1 \wedge \ldots \wedge \neg T_n) = \emptyset \\
\iff\ & \mathcal{L}(T_0 \wedge \neg (T_1 \vee \ldots \vee T_n)) = \emptyset \\
\iff\ & \mathcal{L}(T_0) \cap \overline{\mathcal{L}(T_1 \vee \ldots \vee T_n)} = \emptyset \\
\iff\ & \mathcal{L}(T_0) \subseteq \mathcal{L}(T_1 \vee \ldots \vee T_n) \\
\iff\ & \mathcal{L}(T_0) \subseteq \mathcal{L}(T_1) \cup \ldots \cup \mathcal{L}(T_n) \\
\iff\ & \exists \varphi \colon T_0 \to T_k \quad \text{for some index } 1 \le k \le n
\end{aligned}$$

Therefore, it is sufficient to check whether for each conjunction $(T_0 \wedge \neg T_1 \wedge \cdots \wedge \neg T_n)$ in the DNF of F, there exists a morphism $\varphi \colon T_0 \to T_k$ for some $1 \le k \le n$.

Inclusion:
The language inclusion problem can be reduced to the aforementioned emptiness problem. To solve the language inclusion we use the following equivalence:

$$\mathcal{L}(F_1) \subseteq \mathcal{L}(F_2) \iff \mathcal{L}(F_1 \wedge \neg F_2) = \emptyset$$

Since the emptiness problem is decidable we can conclude that the language inclusion problem is decidable as well. □

A.4. Proofs of Chapter 8

Lemma 8.13. *The composition of two legal morphisms is a legal morphism.*

Proof. Let $\varphi_1 \colon T_1[M_1] \to T_2[M_2]$ and $\varphi_2 \colon T_2[M_2] \to T_3[M_3]$ be two legal morphisms in the category of multiply annotated graphs. Since φ_1 is legal we get that for all $(\ell_1, u_1) \in M_1$ there exists $(\ell_2, u_2) \in M_2$ such that $\ell_2 \leq \mathcal{A}_{\varphi_1}(\ell_1)$ and $\mathcal{A}_{\varphi_1}(u_1) \leq u_2$ hold. Furthermore φ_2 is legal and therefore we get that for all $(\ell_2, u_2) \in M_2$ there exists $(\ell_3, u_3) \in M_3$ such that $\ell_3 \leq \mathcal{A}_{\varphi_2}(\ell_2)$ and $\mathcal{A}_{\varphi_2}(u_2) \leq u_3$ hold as well. We define $\varphi \colon T_1[M_1] \to T_3[M_3]$ to be the composed morphism with $\varphi = \varphi_2 \circ \varphi_1$ and due to the fact that $\mathcal{A}$ is a functor which preserves monotonicity, we get the following two inequalities:

$$\begin{array}{rc|rc}
 & \ell_2 \leq \mathcal{A}_{\varphi_1}(\ell_1) & & \mathcal{A}_{\varphi_1}(u_1) \leq u_2 \\
\Rightarrow & \mathcal{A}_{\varphi_2}(\ell_2) \leq \mathcal{A}_{\varphi_2}(\mathcal{A}_{\varphi_1}(\ell_1)) & \Rightarrow & \mathcal{A}_{\varphi_2}(\mathcal{A}_{\varphi_1}(u_1)) \leq \mathcal{A}_{\varphi_2}(u_2) \\
\Rightarrow & \ell_3 \leq \mathcal{A}_{\varphi_2}(\ell_2) \leq \mathcal{A}_{\varphi_2 \circ \varphi_1}(\ell_1) & \Rightarrow & \mathcal{A}_{\varphi_2 \circ \varphi 1}(u_1) \leq \mathcal{A}_{\varphi_2}(u_2) \leq u_3 \\
\Rightarrow & \ell_3 \leq \mathcal{A}_{\varphi}(\ell_1) & \Rightarrow & \mathcal{A}_{\varphi}(u_1) \leq u_3
\end{array}$$

Since both $\ell_3 \leq \mathcal{A}_{\varphi}(\ell_1)$ and $\mathcal{A}_{\varphi}(u_1) \leq u_3$ hold, the morphism φ is legal. □

Proposition 8.19 (Emptiness check for languages specified by multiply annotated graphs). *For a graph language $\mathcal{L}(T[M])$ characterized by a multiply annotated type graph $T[M]$ over $\mathcal{B}^n$ the emptiness problem is decidable: $\mathcal{L}(T[M]) = \emptyset$ iff $M = \emptyset$ or for each $(\ell, u) \in M$ there exists an edge $e \in E_T$ such that $\ell(e) \geq 1$ and $(u(src(e)) = 0$ or $u(tgt(e)) = 0)$.*

Proof.
$\Leftarrow$: Assume that $M = \emptyset$, in this case $\mathcal{L}(T[M])$ is clearly empty as well. Assume that there is an annotation $(\ell, u) \in M$ such that $\ell(e) \geq 1$ and $(u(src(e)) = 0$ or $u(tgt(e)) = 0)$ for some edge $e \in E_T$. Then no graph can satisfy these lower and upper bounds, since we are forced to map at least one edge to e, but are not allowed to map any node to the source respectively target node. If this is true for all annotations, the language of the type graph must be empty.

$\Rightarrow$: Let $\mathcal{L}(T[M]) = \emptyset$ and we assume by contradiction that $M \neq \emptyset$ and there exists one annotation $(\ell, u) \in M$ such that for every edge $e \in E_T$ with $\ell(e) \geq 1$ we have $u(src(e)) \geq 1$ and $u(tgt(e)) \geq 1)$.

Now take $T[\ell, u]$ and remove from T all edges and nodes x with $u(x) = 0$, resulting in a graph T'. If a node is removed all incident edges are removed as well. Note that in such a case only edges e with $\ell(e) = 0$ will be removed (due to the condition above). Next define $\ell' = \ell|_{T'}$ and $u' = u|_{T'}$. Due to the considerations above there exists a legal morphism (an embedding) $T'[\ell', u'] \to T[\ell, u]$, since the removed items had a lower bound of 0. Furthermore each remaining item has an upper bound of at least 1, i.e., it represents at least one node or edge.

Now construct a graph G from T' by proceeding as follows: starting with T' as base graph, for every node v with $\ell'(v) = k$ add $k - 1$ isolated nodes (zero isolated nodes if $k = 0$). For every edge e with $\ell'(e) = k$ add $k - 1$ parallel edges between $src(e), tgt(e)$. Clearly, there is a morphism $\varphi \colon G \to T'$ obtained by mapping every item to the item from which it originated.

Mapping $G[s_G, s_G]$ to T' via φ will give us an annotation $\mathcal{B}^n_\varphi(s_G)$. By construction, this annotation will coincide with the lower bound ℓ in all cases, but for the case where there is a node v with $\ell(v) = 0$. In this case $\mathcal{B}^n_\varphi(s_G)(v) = 1$, but this is covered by the upper bound which is at least 1. Hence $\varphi\colon G[s_G, s_G] \to T'[\ell', u']$ is a legal morphism, and by composing it with morphism $T'[\ell', u'] \to T[\ell, u]$ we get $G \in \mathcal{L}(T[M])$, thus $\mathcal{L}(T[M]) \neq \emptyset$, as desired. □

Proposition 8.20 (Language inclusion and legal morphisms). *The existence of a legal morphism $\varphi\colon T_1[M] \to T_2[N]$ implies $\mathcal{L}(T_1[M]) \subseteq \mathcal{L}(T_2[N])$.*

Proof. Every graph $G \in \mathcal{L}(T_1[M])$ has a legal morphism $\varphi'\colon G[s_G, s_G] \to T_1[M]$. Whenever there exists a legal morphism $\varphi\colon T_1[M] \to T_2[N]$ between the two multiply annotated type graphs, we obtain the morphism $\eta\colon G[s_G, s_G] \to T_2[N]$ with $\eta = \varphi \circ \varphi'$ which is legal due to Lemma 8.13. Therefore $G \in T_2[N]$ holds and we can conclude that $\mathcal{L}(T_1[M]) \subseteq \mathcal{L}(T_2[N])$ also holds. □

Proposition 8.25 (Language inclusion and bounded pathwidth). *The language inclusion problem is decidable for graph languages of bounded pathwidth characterized by multiply annotated type graphs over $\mathcal{B}^n$. That is, given $k \in \mathbb{N}$ and multiply annotated type graphs $T_1[M_1]$ and $T_2[M_2]$, it is decidable whether $\mathcal{L}(T_1[M_1])^{\leq k} \subseteq \mathcal{L}(T_2[M_2])^{\leq k}$, with $\mathcal{L}(T[M])^{\leq k} = \{G \in \mathcal{L}(T[M]) \mid pw(G) \leq\ k\}$.*

Proof. The proof is given in Section 8.3.3. □

Proposition 8.31 (Functoriality of C). *Let two cospans $c_1\colon J \to G \leftarrow K$ and $c_2\colon K \to H \leftarrow L$ be given and let $id_G\colon G \to G \leftarrow G$ be the identity cospan.*

The mapping $\mathcal{C}_{T[M]}\colon \mathrm{Cospan}_m(\mathbf{Graph}_\Lambda) \to \mathbf{Rel}$ is a functor, i.e.

1. $\mathcal{C}_{T[M]}(id_G) = id_{\mathcal{C}_{T[M]}(G)}$
2. $\mathcal{C}_{T[M]}(c_1; c_2) = \mathcal{C}_{T[M]}(c_1); \mathcal{C}_{T[M]}(c_2)$

Proof.

1. The identity relation $id_{C_{T[M]}(G)}$ consists of all pairs (i, i) with $i \in C_{T[M]}(G)$. Let the two states $i, j \in \mathcal{C}_{T[M]}(G)$ be given with $i = (f_1\colon G \to T, b_1)$ and $j = (f_2\colon G \to T, b_2)$. The pair (i, j) is in the relation $\mathcal{C}_{T[M]}(id_G)$ if and only if there exists a morphism $h\colon G \to T$ such that, for all[2] $x \in T$ the equation $b_2(x) = b_1(x) + |\{y \in (G \setminus id(G)) \mid h(y) = x\}|$ holds and the following diagram commutes:

$$\begin{tikzcd}
 & \overbrace{\hphantom{G \xrightarrow{id} G \xleftarrow{id} G}}^{id_G\colon G \looparrowright G} & \\
G \arrow[r, "id"] \arrow[dr, "f_1"'] & G \arrow[d, dashed, "\exists h"] & G \arrow[l, "id"'] \arrow[dl, "f_2"] \\
 & T &
\end{tikzcd}$$

Since the diagram commutes we obtain that $f_1 = f_2$ since $f_1 = h \circ id = f_2$ holds and for all $x \in T$ the annotation functions b_1 and b_2 are equal due to the following equation:

$$b_2(x) = b_1(x) + |\{y \in (G \setminus id(G)) \mid h(y) = x\}|$$

[2] We write $x \in T$ as an abbreviation for $x \in V_T \cup E_T$.

$$= b_1(x) + |\{y \in \emptyset \mid h(y) = x\}| = b_1(x) + 0 = b_1(x)$$

This is equivalent to $i = j$ and therefore for all $i \in C_{T[M]}(G)$ the following equation holds:

$$\begin{aligned} \mathcal{C}_{T[M]}(id_G) &= \{(i, j) \in C_{T[M]}(G) \times C_{T[M]}(G) \mid i = j\} \\ &= id_{C_{T[M]}(G)} \end{aligned}$$

Therefore $\mathcal{C}_{T[M]}(id_G) = id_{C_{T[M]}(G)}$ holds, as clearly all states (i, i) meet the condition.
In the following part let $c_1 \colon J \xrightarrow{g_1} G \xleftarrow{g_2} K$ and $c_2 \colon K \xrightarrow{g'_1} H \xleftarrow{g'_2} L$ be given and let $c = c_1; c_2$ with $c \colon J \xrightarrow{j_1 \circ g_1} G' \xleftarrow{j_2 \circ g'_2} L$ be the composed morphism of c_1 and c_2.

2.” $\subseteq$ ” : Let $(i, j) \in \mathcal{C}_{T[M]}(c_1; c_2)$ be given with $i \in \mathcal{C}_{T[M]}(J)$ and $j \in \mathcal{C}_{T[M]}(L)$ such that $i = (f_1 \colon J \to T, b_1)$ and $j = (f_3 \colon L \to T, b_3)$. Then there exists a morphism $h \colon G' \to T$ such that $b_3(x) = b_1(x) + |\{y \in (G' \setminus (j_2 \circ g'_2)(L)) \mid h(y) = x\}|$ holds for all $x \in T$ and the following diagram commutes:

$$\begin{tikzcd}
& c_1 \colon J \hookrightarrow K & & c_2 \colon K \hookrightarrow L & \\
J \arrow[r, "g_1"] \arrow[rrdd, "f_1"'] & G \arrow[rd, "j_1"'] & K \arrow[l, "g_2"'] \arrow[r, "g'_1"] & H \arrow[ld, "j_2"] & L \arrow[l, "g'_2"'] \arrow[lldd, "f_3"] \\
& & G' \arrow[d, dashed, "\exists h"] & & \\
& & T & &
\end{tikzcd}$$

To prove that $(i, j) \in \mathcal{C}_{T[M]}(c_1); \mathcal{C}_{T[M]}(c_2)$ follows from the above properties, we need to show that there exists a $k \in \mathcal{C}_{T[M]}(K)$ where $k = (f_2 \colon K \to T, b_2)$ such that $(i, k) \in \mathcal{C}_{T[M]}(c_1)$ and $(k, j) \in \mathcal{C}_{T[M]}(c_2)$. Let $f_2 = h \circ j_1 \circ g_2 = h \circ j_2 \circ g'_1$. Then there must exist two morphisms $h_1 \colon G \to T$, $h_2 \colon H \to T$ such that the following six properties hold:

$$(i, k) \in \mathcal{C}_{T[M]}(c_1) \begin{cases} (1) & h_1 \circ g_1 = f_1 \\ (2) & h_1 \circ g_2 = f_2 \\ (3) & \forall x \in T \quad b_2(x) = b_1(x) + |\{y \in (G \setminus g_2(K)) \mid h_1(y) = x\}| \end{cases}$$

$$(k, j) \in \mathcal{C}_{T[M]}(c_2) \begin{cases} (4) & h_2 \circ g'_1 = f_2 \\ (5) & h_2 \circ g'_2 = f_3 \\ (6) & \forall x \in T \quad b_3(x) = b_2(x) + |\{y \in (H \setminus g'_2(L)) \mid h_2(y) = x\}| \end{cases}$$

We define h_1, h_2 to be $h_1 = h \circ j_1$ and $h_2 = h \circ j_2$ which already satisfy the following four properties:

$$\begin{array}{llll} (1) & h_1 \circ g_1 = h \circ j_1 \circ g_1 = f_1 & (4) & h_1 \circ g'_1 = h \circ j_2 \circ g'_1 = f_2 \\ (2) & h_1 \circ g_2 = h \circ j_1 \circ g_2 = f_2 & (5) & h_1 \circ g'_2 = h \circ j_2 \circ g'_2 = f_3 \end{array}$$

We define b_2 with respect to property (3), such that for all $x \in T$ the equation $b_2(x) = b_1(x) + |\{y \in (G \setminus g_2(K)) \mid h_1(y) = x\}|$ holds. It remains to show property (6): we conduct the proof by first showing the following equation (7) for all $x \in T$, from which we can easily derive (6) afterwards:

$$
\begin{aligned}
&|\{y \in (G \setminus g_2(K)) \mid h_1(y) = x\}| + |\{y \in (H \setminus g_2'(L)) \mid h_2(y) = x\}| \\
=&|\{y \in (G' \setminus (j_2 \circ g_2')(L)) \mid h(y) = x\}| \qquad (7)
\end{aligned}
$$

Since the morphisms j_1 and j_2 are both injective and G' is the pushout object of G and H over the common graph K we get that $G' = j_1(G \setminus g_2(K)) \uplus j_2(H)$. Subtracting all elements $x \in L$ that are being mapped into H on both sides of the equation, we get that $G' \setminus (j_2 \circ g_2')(L) = j_1(G \setminus g_2(K)) \uplus j_2(H \setminus g_2'(L))$ holds as well. Using this fact we can prove equation (7) which holds for all $x \in T$:

$$
\begin{aligned}
&|\{y \in (G' \setminus (j_2 \circ g_2')(L)) \mid h(y) = x\}| \\
=&|\{y \in \big(j_1(G \setminus g_2(K)) \uplus j_2(H \setminus g_2'(L))\big) \mid h(y) = x\}| \\
=&|\{y \in j_1(G \setminus g_2(K)) \mid h(y) = x\} \uplus \{y \in j_2(H \setminus g_2'(L)) \mid h(y) = x\}| \\
=&|\{y \in G \setminus g_2(K) \mid (h \circ j_1)(y) = x\} \uplus \{y \in H \setminus g_2'(L) \mid (h \circ j_2)(y) = x\}| \\
=&|\{y \in G \setminus g_2(K) \mid h_1(y) = x\} \uplus \{y \in H \setminus g_2'(L) \mid h_2(y) = x\}| \\
=&|\{y \in (G \setminus g_2(K)) \mid h_1(y) = x\}| + |\{y \in (H \setminus g_2'(L)) \mid h_2(y) = x\}|
\end{aligned}
$$

Using equation (7) we conclude that property (6) always holds for all $x \in T$:

$$
\begin{aligned}
b_3(x) &= b_1(x) + |\{y \in (G' \setminus (j_2 \circ g_2')(L)) \mid h(y) = x\}| \\
&= b_1(x) + |\{y \in (G \setminus g_2(K)) \mid h_1(y) = x\}| + |\{y \in (H \setminus g_2'(L)) \mid h_2(y) = x\}| \\
&= b_2(x) + |\{y \in (H \setminus g_2'(L)) \mid h_2(y) = x\}|
\end{aligned}
$$

Therefore $(i, j) \in \mathcal{C}_{T[M]}(c_1); \mathcal{C}_{T[M]}(c_2)$ holds as well.

2." $\supseteq$ " : Let two pairs $(i, k) \in \mathcal{C}_{T[M]}(c_1)$ and $(k, j) \in \mathcal{C}_{T[M]}(c_2)$ be given with $i \in \mathcal{C}_{T[M]}(J)$, $k \in \mathcal{C}_{T[M]}(K)$ and $j \in \mathcal{C}_{T[M]}(L)$ such that $i = (f_1\colon J \to T, b_1)$, $k = (f_2\colon K \to T, b_2)$ and $j = (f_3\colon L \to T, b_3)$. Then in addition there exist two morphisms $h_1\colon G \to T$ and $h_2\colon H \to T$ such that for all $x \in T$ the two equations $b_2(x) = b_1(x) + |\{y \in (G \setminus g_2(K)) \mid h_1(y) = x\}|$ and
$b_3(x) = b_2(x) + |\{y \in (H \setminus g_2'(L)) \mid h_2(y) = x\}|$ both hold and the following diagram commutes:

$$\begin{tikzcd}
{} \arrow[rr, phantom, "\overbrace{\hspace{3cm}}^{c_1\colon J \looparrowright K}"] & & {} \arrow[rr, phantom, "\overbrace{\hspace{3cm}}^{c_2\colon K \looparrowright L}"] & & {} \\
J \arrow[r, "g_1"] \arrow[rrd, "f_1"'] & G \arrow[rd, dashed, "\exists h_1"] & K \arrow[l, "g_2"'] \arrow[r, "g_1'"] \arrow[d, "f_2"] & H \arrow[ld, dashed, "\exists h_2"'] & L \arrow[l, "g_2'"'] \arrow[lld, "f_3"] \\
 & & T & &
\end{tikzcd}$$

To prove that $(i, j) \in \mathcal{C}_{T[M]}(c_1; c_2)$ is satisfied from the properties gained so far, we need to show that $b_3(x) = b_1(x) + |\{y \in (G' \setminus (j_2 \circ g_2')(L)) \mid h(y) = x\}|$ holds and that there exists a morphism $h\colon G' \to T$ such that the following diagram commutes:

$$\begin{tikzcd}
 & c_1\colon J \looparrowright K & & c_2\colon K \looparrowright L & \\
J \arrow[r, "g_1"] \arrow[rrdd, "f_1"'] & G \arrow[rd, "j_1"] \arrow[rdd, "h_1"'] & K \arrow[l, "g_2"'] \arrow[r, "g_1'"] \arrow[d, phantom, "(PO)"] & H \arrow[ld, "j_2"'] \arrow[ldd, "h_2"] & L \arrow[l, "g_2'"'] \arrow[lldd, "f_3"] \\
 & & G' \arrow[d, dashed, "\exists h"'] & & \\
 & & T & &
\end{tikzcd}$$

The morphism $h\colon G' \to T$ exists and is unique due to the universal property of pushouts. From the two equations $b_2(x) = b_1(x) + |\{y \in (G \setminus g_2(K)) \mid h_1(y) = x\}|$ and $b_3(x) = b_2(x) + |\{y \in (H \setminus g_2'(L)) \mid h_2(y) = x\}|$ we can derive the following equation which holds for all $x \in T$:

$$b_3(x) = b_1(x) + |\{y \in (G \setminus g_2(K)) \mid h_1(y) = x\}| + |\{y \in (H \setminus g_2'(L)) \mid h_2(y) = x\}|$$

Using the results of equation (7) from the previous proof direction, we directly can conclude that $b_3(x) = b_1(x) + |\{y \in (G' \setminus (j_2 \circ g_2')(L)) \mid h(y) = x\}|$ also holds and therefore $(i, j) \in \mathcal{C}_{T[M]}(c_1; c_2)$ holds, which completes this proof. □

Proposition 8.32 (Language of C). *Let the multiply annotated type graph $T[M]$ (over $\mathcal{B}^n$) and the automaton functor $\mathcal{C}\colon \mathrm{Cospan}_m(\mathbf{Graph}_\Lambda) \to \mathbf{Rel}$ for $T[M]$ be given. Then $\mathcal{L}_\mathcal{C} = \mathcal{L}(T[M])$ holds, i.e. for a graph G we have $G \in \mathcal{L}(T[M])$ if and only if there exist states $i \in I \subseteq \mathcal{C}(\varnothing)$ and $f \in F \subseteq \mathcal{C}(\varnothing)$ such that $(i, f) \in \mathcal{C}(c)$, where $c\colon \varnothing \to G \leftarrow \varnothing$.*

Proof. We will prove the following equality:

$$G \in \mathcal{L}(T[M]) \iff \exists i \in I \subseteq \mathcal{C}(\varnothing), \exists j \in F \subseteq \mathcal{C}(\varnothing) : (i, j) \in \mathcal{C}(c)$$

"⇒": Since $(c\colon \varnothing \to G \leftarrow \varnothing) \in \mathcal{L}(T[M])$ holds, there exists a legal morphism $\varphi\colon G \to T$ and a pair of multiplicities $(\ell, u) \in M$ such that $\ell \leq \mathcal{B}^n_\varphi(s_G) \leq u$ holds. Let (i, j) be $i = (f_1\colon \varnothing \to T, 0) \in I$ and $j = (f_2\colon \varnothing \to T, \mathcal{B}^n_\varphi(s_G)) \in F$ which are clearly in the relation $\mathcal{C}(c)$, i.e. $(i, j) \in \mathcal{C}(c)$ since for all $x \in T$ the equation $\mathcal{B}^n_\varphi(s_G)(x) = 0 + |\{y \in (G \setminus g_2(\varnothing)) \mid \varphi(y) = x\}| = |\{y \in G \mid \varphi(y) = x\}|$ holds by definition and the following diagram commutes:

$$\begin{tikzcd}
 & c\colon \varnothing \looparrowright \varnothing & \\
\varnothing \arrow[r, "g_1"] \arrow[rd, "f_1"'] & G \arrow[d, "\varphi"] & \varnothing \arrow[l, "g_2"'] \arrow[ld, "f_2"] \\
 & T &
\end{tikzcd}$$

"⇐": There exists $i \in I \subseteq \mathcal{C}(\varnothing)$ and $j \in F \subseteq \mathcal{C}(\varnothing)$ with $i = (f_1\colon \varnothing \to T, 0)$ and $j = (f_2\colon \varnothing \to T, b)$ such that $(i, j) \in \mathcal{C}(c)$ holds. Therefore, there exists a pair of multiplicities $(\ell, u) \in M$ with $\ell \leq b \leq u$ and we get that there exists a morphism $\varphi\colon G \to T$ such that the following diagram commutes:

$$\begin{tikzcd}
 & \overbrace{\hspace{4cm}}^{c\colon \varnothing \hookrightarrow \varnothing} & \\
\varnothing \arrow[r, "g_1"] \arrow[dr, "f_1"'] & G \arrow[d, dashed, "\exists\varphi"] & \varnothing \arrow[l, "g_2"'] \arrow[dl, "f_2"] \\
 & T &
\end{tikzcd}$$

For all $x \in T$ the following equation holds:

$$\begin{aligned} b(x) &= 0 + |\{y \in (G \setminus g_2(\varnothing)) \mid \varphi(y) = x\}| \\ &= |\{y \in G \mid \varphi(y) = x\}| \\ &= \mathcal{B}^n_\varphi(s_G)(x) \end{aligned}$$

From $\ell \leq b \leq u$ we can infer that $\varphi\colon G \to T$ is a legal morphism due to the fact that $\ell \leq \mathcal{B}^n_\varphi(s_G) \leq u$ holds as well, and therefore $G \in \mathcal{L}(T[M])$. □

Proposition 8.33 (Closure under intersection). *Graph languages specified by annotated graphs are closed under intersection.*

Proof. Let two multiply annotated type graphs $T_1[M_1]$ and $T_2[M_2]$ be given. Let $T_1 \times T_2$ be the usual product graph in the underlying category $\mathbf{Graph}_\Lambda$.

We now consider the multiply annotated type graph $(T_1 \times T_2)[N]$ where the set of annotations N is defined as follows:

$$\begin{aligned} N = \{(\ell, u) \mid \ell, u \in \mathcal{A}(T_1 \times T_1) &\text{ such that} \\ \pi_1\colon (T_1 \times T_2)[\ell, u] \to T_1[M_1] &\text{ is legal and} \\ \pi_2\colon (T_1 \times T_2)[\ell, u] \to T_2[M_2] &\text{ is legal}\} \end{aligned}$$

Therefore for each $(\ell, u) \in N$ there exist $(\ell_1, u_1) \in M_1$ and $(\ell_2, u_2) \in M_2$ such that the following four properties hold:

$$\begin{array}{ll} \mathcal{A}_{\pi_1}(\ell) \geq \ell_1 & \mathcal{A}_{\pi_1}(u) \leq u_1 \\ \mathcal{A}_{\pi_2}(\ell) \geq \ell_2 & \mathcal{A}_{\pi_2}(u) \leq u_2 \end{array}$$

$$\begin{tikzcd}
 & (T_1 \times T_2)[N] \arrow[dl, "\pi_1"] \arrow[dr, "\pi_2"] & \\
T_1[M_1] & & T_2[M_2]
\end{tikzcd}$$

We will now prove the following equality:

$$\mathcal{L}(T_1[M_1]) \cap \mathcal{L}(T_2[M_2]) = \mathcal{L}((T_1 \times T_2)[N])$$

$\subseteq$: Let $G \in \mathcal{L}(T_1[M_1]) \cap \mathcal{L}(T_2[M_2])$. Then there exist two legal morphisms $\varphi_1\colon G[s_G, s_G] \to T_1[M_1]$ and $\varphi_2\colon G[s_G, s_G] \to T_2[M_2]$. Due to the universal property of pullbacks in the underlying category $\mathbf{Graph}_\Lambda$, there exists a unique graph morphism $\eta\colon G \to T_1 \times T_2$ such that the following diagram commutes:

$$\begin{tikzcd}
 & G[s_G, s_G] \arrow[d, dashed, "\eta"] \arrow[ddl, bend right, "\varphi_1"'] \arrow[ddr, bend left, "\varphi_2"] & \\
 & (T_1 \times T_2)[N] \arrow[dl, "\pi_1"] \arrow[dr, "\pi_2"] & \\
T_1[M_1] & & T_2[M_2]
\end{tikzcd}$$

Since $\varphi_i = \pi_i \circ \eta$ with $i \in \{1,2\}$ is a legal morphism, there exist annotations $(\ell_1, u_1) \in M_1$ and $(\ell_2, u_2) \in M_2$ such that the following inequalities hold:

$$\begin{aligned} \ell_1 &\leq \mathcal{A}_{\varphi_1}(s_G) = \mathcal{A}_{\pi_1 \circ \eta}(s_G) = \mathcal{A}_{\pi_1}(\mathcal{A}_\eta(s_G)) \leq u_1 \\ \ell_2 &\leq \mathcal{A}_{\varphi_2}(s_G) = \mathcal{A}_{\pi_2 \circ \eta}(s_G) = \mathcal{A}_{\pi_2}(\mathcal{A}_\eta(s_G)) \leq u_2 \end{aligned}$$

Therefore the pair $(\mathcal{A}_\eta(s_G), \mathcal{A}_\eta(s_G))$ is one of the annotations in N and we can conclude that $G \in \mathcal{L}((T_1 \times T_2)[N])$ holds.

$\supseteq$: We now assume $G \in \mathcal{L}((T_1 \times T_2)[N])$ holds. Then there exists a legal morphism $\eta\colon G[s_G, s_G] \to (T_1 \times T_2)[N]$ with a pair of annotations $(\ell, u) \in N$ such that it holds that $\ell \leq \mathcal{A}_\eta(s_G) \leq u$. For each such pair $(\ell, u) \in N$ we have two legal morphisms $\pi_1\colon (T_1 \times T_2)[\ell, u] \to T_1[M_1]$ and $\pi_2\colon (T_1 \times T_2)[\ell, u] \to T_2[M_2]$, by construction. We obtain two morphisms $\varphi_1\colon G[s_G, s_G] \to T_1[M_1]$ with $\varphi_1 = \pi_1 \circ \eta$ and $\varphi_2\colon G[s_G, s_G] \to T_2[M_2]$ with $\varphi_2 = \pi_2 \circ \eta$, which are legal due to Lemma 8.13. Therefore we can conclude that $G \in (\mathcal{L}(T_1[M_1]) \cap \mathcal{L}(T_2[M_2]))$. □

To prove Proposition 8.34 we need the following two Lemma L.3 and Lemma L.5 alongside a reduction operation given in Definition A.4.

Lemma L.3. Assume that we are working with annotations over $\mathcal{B}^n$. Let $i\colon A[M] \to T[N]$ and $\varphi\colon G[s_G, s_G] \to T[N]$ be two legal graph morphisms where i is injective. Let $(\ell, u) \in M$ be one of the double multiplicities of the graph A. Whenever $\mathcal{B}^n_\varphi(s_G) \leq \mathcal{B}^n_i(u)$, we can deduce that there exists a graph morphism $\zeta\colon G \to A$ with $i \circ \zeta = \varphi$, i.e. the diagram commutes.

$$\begin{tikzcd} & G[s_G, s_G] \arrow[dl, dashed, "\zeta"'] \arrow[d, "\varphi"] \\ A[M] \arrow[r, tail, "i"'] & T[N] \end{tikzcd}$$

Proof. The morphisms ζ exists if all elements of the form $\varphi(x)$ with $x \in G$ are in the range of i. For such an x we have $1 = s_G(x) \leq \mathcal{B}^n_\varphi(s_G)(\varphi(x))$, since $\mathcal{B}^n_\varphi(s_G)(\varphi(x))$ is the sum of the s_G-annotations of all preimages of x. Furthermore $\mathcal{B}^n_\varphi(s_G)(\varphi(x)) \leq \mathcal{B}^n_i(u)(\varphi(x))$. But $\mathcal{B}^n_i(u)(y) = 0$ for all $y \in T$ that are not in the range of i, since the empty sum evaluates to 0. But since $\mathcal{B}^n_i(u)(\varphi(x)) \geq 1$, we can conclude that $\varphi(x)$ has a preimage under i. □

In addition, we need the concept of reduction: the reduction operation shifts annotations over morphisms in the reverse direction.

Definition A.4 (Reduction)**.** Let $\mathcal{A}$ be an (annotation) functor. For a morphism $\varphi\colon G \to G'$ and a monoid element $a' \in \mathcal{A}(G')$ we define the reduction of a' to G as follows:

$$red_\varphi(a') = \bigvee \{a \mid \mathcal{A}_\varphi(a) \leq a'\}.$$

In the case of multiplicities, the reduction operator satisfies the needed property given in the next Lemma L.5. Further annotation properties which depend on the reduction operation are given in Definition 10.8.

Lemma L.5. Assume that we are working with annotations over $\mathcal{B}^n$. If $\varphi\colon G \to H$ is injective, we obtain the following equality for all $x \in G$:

$$red_\varphi(a')(x) = a'(\varphi(x))$$

Furthermore, if $\varphi\colon G \to G'$ is injective, it holds that $red_\varphi(\mathcal{B}^n_\varphi(a)) = a$ for every $a \in \mathcal{B}^n(G)$.

Proof. Straightforward from the definition of multiplicities (Definition 8.6). □

Proposition 8.34 (Closure under union). *Graph languages specified by multiply annotated graphs over functor $\mathcal{B}^n$ are closed under union.*

Proof. Let two multiply annotated type graphs $T_1[M_1]$ and $T_2[M_2]$ be given. Let $T_1 \oplus T_2$ be the usual coproduct graph in the underlying category $\mathbf{Graph}_\Lambda$, together with the embedding morphisms $i_1\colon T_1 \to T_1 \oplus T_2$ and $i_2\colon T_2 \to T_1 \oplus T_2$:

$$\begin{tikzcd}
T_1[M_1] \arrow[dr, tail, "i_1"'] & & T_2[M_2] \arrow[dl, tail, "i_2"] \\
& T_1 \oplus T_2[N] &
\end{tikzcd}$$

We define the set of annotations N for $(T_1 \oplus T_2)[N]$ using the following two sets:

$$N_1 = \{\ (\mathcal{B}^n_{i_1}(\ell_1), \mathcal{B}^n_{i_1}(u_1))\ \mid\ (\ell_1, u_1) \in M_1\}$$
$$N_2 = \{(\mathcal{B}^n_{i_2}(\ell_2), \mathcal{B}^n_{i_2}(u_2))\mid\ (\ell_2, u_2) \in M_2\}$$

Finally we define $N = N_1 \cup N_2$.

By this definition, for all elements $x \in T_1$ and for all $(\ell, u) \in M_1$ there exists $(\ell_1, u_1) \in N_1$ such that $\mathcal{B}^n_{i_1}(\ell)(i_1(x)) = \ell_1(i_1(x))$ and $\mathcal{B}^n_{i_1}(u)(i_1(x)) = u_1(i_1(x))$. This makes i_1 a legal morphism since $N_1 \subseteq N$. The same holds for i_2 analogously. We will now prove the following equality:

$$\mathcal{L}(T_1[M_1]) \cup \mathcal{L}(T_2[M_2]) = \mathcal{L}((T_1 \oplus T_2)[N])$$

$\subseteq$: Let $G \in (\mathcal{L}(T_1[M_1]) \cup \mathcal{L}(T_2[M_2]))$. Then there exists at least one legal morphism $\varphi_1\colon G[s_G, s_G] \to T_1[M_1]$ or $\varphi_2\colon G[s_G, s_G] \to T_2[M_2]$. We assume that $G \in \mathcal{L}(T_1[M_1])$. Let $\eta\colon G[s_G, s_G] \to (T_1 \oplus T_2)[N]$ be the composed morphism of i_1 and φ_1 with $\eta = i_1 \circ \varphi_1$. Then η is legal due to Lemma 8.13 and therefore $G \in \mathcal{L}((T_1 \oplus T_2)[N])$ holds. The proof for the case where $G \in \mathcal{L}(T_2[M_2])$ works in the same way.

$\supseteq$: We now assume $G \in \mathcal{L}((T_1 \oplus T_2)[N])$. Then, there exists a legal morphism $\eta\colon G[s_G, s_G] \to (T_1 \oplus T_2)[N]$ with an annotation $(\ell, u) \in N$ and $\ell \le \mathcal{B}^n_\eta(s_G) \le u$. For each $(\ell, u) \in N$, we know that the pair belongs to N_1 or N_2. Assume that $(\ell, u) \in N_1$. Then there exists $(\ell_1, u_1) \in M_1$ such that $\ell = \mathcal{B}^n_{i_1}(\ell_1)$, $u = \mathcal{B}^n_{i_1}(u_1)$. Hence $\mathcal{B}^n_\eta(s_G) \le u = \mathcal{B}^n_{i_1}(u_1)$. From Lemma L.3 it follows that there exists a graph morphism $\zeta_1\colon G \to T_1$ with $\eta = i_1 \circ \zeta_1$ such that the following diagram commutes in the underlying category $\mathbf{Graph}_\Lambda$:

$$\begin{tikzcd}
 & G[s_G, s_G] \arrow[dl, dashed, "\zeta_1"'] \arrow[dd, "\eta"] & \\
T_1[M_1] \arrow[dr, tail, "i_1"'] & & T_2[M_2] \arrow[dl, tail, "i_2"] \\
 & (T_1 \oplus T_2)[N] &
\end{tikzcd}$$

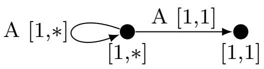
We need to prove that ζ_1 is a legal graph morphism in the category of multiply annotated graphs. We get that $\mathcal{B}^n_\eta(s_G) = \mathcal{B}^n_{i_1 \circ \zeta_1}(s_G) = \mathcal{B}^n_{i_1}(\mathcal{B}^n_{\zeta_1}(s_G))$ and since i_1 is injective, the following inequality holds due to the fact that red_φ is monotone and $red_\varphi(\mathcal{B}^n_\varphi(a)) = a$ holds for every $a \in \mathcal{B}^n(G)$, whenever φ is injective (cf. Lemma L.5):

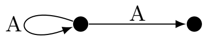
$$\begin{array}{rrcr}
 & \mathcal{B}^n_{i_1}(\ell_1) \leq & \mathcal{B}^n_\eta(s_G) & \leq \mathcal{B}^n_{i_1}(u_1) \\
\Rightarrow & \mathcal{B}^n_{i_1}(\ell_1) \leq & \mathcal{B}^n_{i_1}(\mathcal{B}^n_{\zeta_1}(s_G)) & \leq \mathcal{B}^n_{i_1}(u_1) \\
\Rightarrow & red_{i_1}(\mathcal{B}^n_{i_1}(\ell_1)) \leq & red_{i_1}(\mathcal{B}^n_{i_1}(\mathcal{B}^n_{\zeta_1}(s_G))) & \leq red_{i_1}(\mathcal{B}^n_{i_1}(u_1)) \\
\Rightarrow & \ell_1 \leq & \mathcal{B}^n_{\zeta_1}(s_G) & \leq u_1
\end{array}$$

Therefore $\zeta_1 \colon G[s_G, s_G] \to T_1[M_1]$ is a legal morphism and we can conclude that $G \in \mathcal{L}(T_1[M_1])$. For a legal morphism $\eta \colon G[s_G, s_G] \to (T_1 \oplus T_2)[N]$ with a pair $(\ell, u) \in N_2$ we get a similar proof which shows that $G \in \mathcal{L}(T_2[M_2])$. Summarizing, in all cases $G \in (\mathcal{L}(T_1[M_1]) \cup \mathcal{L}(T_2[M_2]))$ holds. □

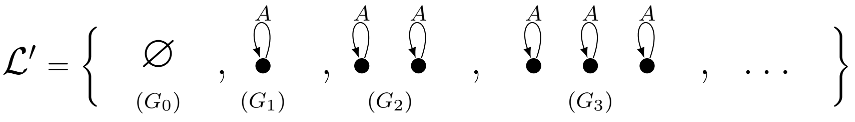
Proposition 8.35 (Non-closure under complement). *Graph languages specified by multiply annotated graphs over functor $\mathcal{B}^n$ are not closed under complement.*

Proof. We show that there exist languages specified by annotated type graphs for which there exists no finite type graph which specifies the complement language. For a counterexample we use the multiply annotated type graph $T[\ell, u]$ (depicted below, on the left) and we observe that for the graph H below on the right, it holds that $H \in \mathcal{L}(T[\ell, u])$:

$T[\ell, u] =$ 

$H =$ 

Now assume, that graph languages specified by annotated type graphs are closed under complement and let $\mathcal{L}(T'[M]) = \overline{\mathcal{L}(T[\ell, u])}$ be the complement language, represented by a multiply annotated type graph $T'[M]$. We consider the sublanguage $\mathcal{L}' \subset \mathcal{L}(T'[M])$ of the complement language which contains all graphs G_n with a fixed number $n \in \mathbb{N}_0$ of discrete A-labeled loops, that is:



Indeed for $n \in \mathbb{N}_0$ and a graph $G_n \in \mathcal{L}'$ we obtain that $G_n \notin \mathcal{L}(T[\ell, u])$ since there exists no morphism which can fulfill the lower bound of the right node in $T[\ell, u]$. Therefore $\mathcal{L}' \subset \overline{\mathcal{L}(T[\ell, u])}$ holds. We show that there can not exist a finite graph $T'[M]$ with the following two properties:

1. for all $n \in \mathbb{N}_0$ and $G_n \in \mathcal{L}'$, there exists an annotation $(\ell_n, u_n) \in M$ such that there exists a legal morphism $\varphi_n \colon G_n[s_{G_n}, s_{G_n}] \to T'[\ell_n, u_n]$.

2. for all graphs $H' \in \mathcal{L}(T[\ell, u])$ we have that $H' \notin \mathcal{L}'$.

We first show that given any $n \in \mathbb{N}_0$, whenever there exists an annotation $(\ell_n, u_n) \in M$ and a legal morphism $\varphi_n \colon G_n[s_{G_n}, s_{G_n}] \to T'[\ell_n, u_n]$ which fulfils property (1), where φ_n is *non-injective*, then property (2) becomes unsatisfiable.

Assume that the legal morphism $\varphi_n \colon G_n[s_{G_n}, s_{G_n}] \to T'[\ell_n, u_n]$ is non-injective and maps at least two nodes of G_n (denoted by n_1, n_2) to the same target node t in $T'[\ell_n, u_n]$. Then the upper bound $u_n(t)$ must be greater than or equal to 2.

Then, there exists a graph $H' \in \mathcal{L}(T[\ell, u])$ that allows a legal morphism into $T'[\ell_n, u_n]$ which can be constructed in the following way: Let $H \in \mathcal{L}(T[\ell, u])$ be the graph defined earlier. Then construct H' as the disjoint union of H and G_{n-2}. The graph H' is an element of $\mathcal{L}(T[\ell, u])$ since we can map all additional $n-2$ nodes with an A-labeled loop to the left node in $T[\ell, u]$.

Now, in order to show that $H' \in \mathcal{L}(T'[\ell_n, u_n])$, consider the two nodes n_1 and n_2 of G_n which are both mapped to t via φ_n. The two nodes of H can both be mapped to t since the upper bound of t admits such a morphism. The former non-looping edge and the loop are mapped to the target loops of n_1 and n_2 respectively. Now map the remaining $n-2$ nodes with loops in H' by mimicking the mapping of the nodes with loops of G_n via φ_n, except n_1 and n_2. This gives us the same annotations on T' as those generated by φ_n, which are consequently within the bound $[\ell_n, u_n]$.

Hence the morphism from H' to $T'[\ell_n, u_n]$ is legal, $H' \in \mathcal{L}(T'[\ell_n, u_n])$ holds and therefore property (2) does not hold. We conclude that all φ_n must be injective to guarantee that there exists no graph $H' \in \mathcal{L}(T[\ell, u])$ that is also contained in the complement language.

However, to ensure that the morphisms $\varphi_n \colon G_n \to T'[\ell_n, u_n]$ are injective for any $n \in \mathbb{N}_0$, the type graph T' would need to consist of at least n different nodes with A-labeled loops attached to them. Since there are infinitely many graphs G_n there can not exist a type graph $T'[M]$ with a finite number of nodes. We conclude that there exists no finite graph $T'[M]$ specifying the complement language of $\mathcal{L}(T[\ell, u])$. □

A.5. Proofs of Chapter 9

The following result is known, we give the proof sketch for the convenience of the reader, since the construction plays an important role in Chapter 9 and Chapter 10.

Proposition 9.15 (Final pullback complements, subobject and partial map classifiers). *Let* $\mathbf{C}$ *be a category with finite limits. Then the following are equivalent:*

(1) $\mathbf{C}$ *has a subobject classifier* $\mathtt{true}\colon \mathbf{1} \rightarrowtail \Omega$ *and final pullback complements for each pair of arrows* $I \overset{\alpha}{\to} L \overset{m}{\rightarrowtail} G$ *where* m *is a mono;*

(2) $\mathbf{C}$ *has a partial map classifier* $(\mathcal{F} : \mathbf{C} \to \mathbf{C}, \eta : Id_{\mathbf{C}} \dot{\to} \mathcal{F})$*.*

Sketch. We just report the corresponding constructions from [DT87], omitting the proofs of the relevant properties.

(1) $\Rightarrow$ ***(2)*** The component $\eta_Y : Y \rightarrowtail \mathcal{F}(Y)$ of the natural transformation η at object $Y \in \mathbf{C}$ is obtained as the final pullback complement of $Y \overset{!_Y}{\to} \mathbf{1} \overset{\mathtt{true}}{\rightarrowtail} \Omega$, as shown in Diagram (A.6).

(2) $\Rightarrow$ ***(1)*** We first observe that, given a partial map classifier $(\mathcal{F}, \eta)$, the subobject classifier is obtained as $\mathbf{1} \overset{\eta_{\mathbf{1}}}{\rightarrowtail} \mathcal{F}(\mathbf{1})$.

We show how to construct a final pullback complement: Given $I \overset{\alpha}{\to} L \overset{m}{\rightarrowtail} G$, consider the components of the natural transformation at I and L, and arrow $\mathcal{F}(\alpha)\colon \mathcal{F}(I) \to \mathcal{F}(L)$, as in (A.7). The mono $L \overset{m}{\rightarrowtail} G$ can be seen as a partial map $G \overset{m}{\leftarrowtail} L \overset{id_L}{\rightarrowtail} L$ from G to L, and this induces a unique arrow $\varphi(m, id_L)$ making the square a pullback. Now let $G \overset{h}{\leftarrow} P \to \mathcal{F}(I)$ be the pullback of $G \overset{\varphi(m,id_L)}{\longrightarrow} \mathcal{F}(L) \overset{\mathcal{F}(\alpha)}{\longleftarrow} \mathcal{F}(I)$. It is easy to see that there is an induced mono (mediating arrow) $n : I \rightarrowtail P$ and it can be shown that $I \overset{n}{\rightarrowtail} P \overset{h}{\to} G$ is the final pullback complement of $I \overset{\alpha}{\to} L \overset{m}{\rightarrowtail} G$.

$$\begin{tikzcd}
Y \arrow[r, tail, "\eta_Y"] \arrow[d, "!_Y"'] \arrow[dr, phantom, "\text{(FPBC)}"] & \mathcal{F}(Y) \arrow[d, "\chi_{\eta_Y}"] \\
\mathbf{1} \arrow[r, tail, "\mathtt{true}"'] & \Omega
\end{tikzcd} \tag{A.6}$$

$$\begin{tikzcd}
 & L \arrow[dd, tail, "m"'] \arrow[dr, tail, "\eta_L"] & & I \arrow[ll, "\alpha"'] \arrow[dr, tail, "\eta_I"] \arrow[dd, dashed, tail, "\eta"'] & \\
L \arrow[ur, tail, "id_L"] \arrow[dr, tail, "m"'] & & \mathcal{F}(L) & & \mathcal{F}(I) \arrow[ll, "\mathcal{F}(\alpha)"'] \\
 & G \arrow[ur, "\varphi(m,id_L)"'] & & P \arrow[ll, "h"] \arrow[ur] &
\end{tikzcd} \tag{A.7}$$

□

Proposition 9.19 (Existence of materialization). *Let* $\varphi\colon L \to A$ *be an arrow in* $\mathbf{C}$*, and let* $\eta_\varphi\colon \varphi \to \mathcal{F}(\varphi)$*, with* $\mathcal{F}(\varphi)\colon \bar{A} \to A$*, be the partial map classifier of* φ *in the slice category* $\mathbf{C}{\downarrow}A$ *(which also is a topos)*[3]*. Then* $L \overset{\eta_\varphi}{\to} \bar{A} \overset{\mathcal{F}(\varphi)}{\to} A$ *is the materialization of* φ*, hence* $\langle\varphi\rangle = \bar{A}$*.*

Proof. Let $L \overset{m}{\rightarrowtail} X \overset{\alpha}{\to} A$ be an object of $\mathbf{Mat}_\varphi$, i.e. a factorization such that $\varphi = \alpha \circ m$. Note that this defines a partial map $(m, id_L)\colon \alpha \rightharpoonup \varphi$ in $\mathbf{C} \downarrow A$

consisting of the span $\alpha \overset{m}{\leftarrowtail} \varphi \overset{id_L}{\rightarrow} \varphi$. Since $\eta_\varphi : \varphi \to \mathcal{F}(\varphi)$ is the component of the partial map classifier, there exists a unique arrow $\varphi(m, id_L)\colon X \to \langle\varphi\rangle$ from $\alpha\colon X \to A$ to $\mathcal{F}(\varphi)\colon \langle\varphi\rangle \to A$ for which the left square in the following diagram is a pullback and the right triangle commutes. The latter holds since $\varphi(m, id_L)$ is an arrow in the slice category.

$$\begin{tikzcd}
L \arrow[r, tail, "m"] \arrow[rr, bend left, "\varphi"] \arrow[d, tail, "id_L"'] \arrow[dr, phantom, "\text{(PB)}"] & X \arrow[r, "\alpha"] \arrow[d, dashed, "\varphi(m{,}id_L)"] & A \\
L \arrow[r, tail, "\eta_\varphi"'] & \langle\varphi\rangle \arrow[ur, bend right, "\mathcal{F}(\varphi)"'] &
\end{tikzcd}$$

□

Corollary 9.20 (Construction of the materialization). *Let $\varphi\colon L \to A$ be an arrow of $\mathbf{C}$ and let $\mathtt{true}_A\colon A \rightarrowtail A \times \Omega$ be the subobject classifier in the slice category $\mathbf{C} \downarrow A$ mapping from $id_A\colon A \to A$ to the projection $\pi_1\colon A \times \Omega \to A$ (see also Fact 9.11). Then the terminal object $L \overset{\eta_\varphi}{\rightarrowtail} \langle\varphi\rangle \overset{\psi}{\to} A$ in the materialization category consists of the arrows $\eta_\varphi, \psi = \pi_1 \circ \chi_{\eta_\varphi}$, where $L \overset{\eta_\varphi}{\rightarrowtail} \langle\varphi\rangle \overset{\chi_{\eta_\varphi}}{\to} A \times \Omega$ is the final pullback complement of $L \overset{\varphi}{\to} A \overset{\mathtt{true}_A}{\rightarrowtail} A \times \Omega$.*

$$\begin{tikzcd}
L \arrow[r, tail, dashed, "\eta_\varphi"] \arrow[d, "\varphi"'] \arrow[dr, phantom, "\text{(FPBC)}"] & \langle\varphi\rangle \arrow[d, dashed, "\chi_{\eta_\varphi}"] \arrow[dr, dashed, "\psi"] & \\
A \arrow[r, tail, dashed, "\mathtt{true}_A"'] & A \times \Omega \arrow[r, "\pi_1"'] & A
\end{tikzcd}$$

Proof. Straightforward from Propositios 9.15 and 9.19 (and the fact that final pullback complements in the slice category correspond to those in the base category [Löw10]). □

Proposition 9.22 (Language of the materialization). *Let $\varphi\colon L \to A$ be an arrow in $\mathbf{C}$ and let $L \overset{\eta_\varphi}{\rightarrowtail} \langle\varphi\rangle \to A$ be the corresponding materialization. Then*

$$\mathcal{L}(L \overset{\eta_\varphi}{\rightarrowtail} \langle\varphi\rangle) = \{L \overset{m_L}{\rightarrowtail} X \mid \exists\psi\colon (X \to A).\ (\varphi = \psi \circ m_L)\}.$$

Proof. We show that the two sets are included into each other:

- ($\subseteq$) Given the materialization $L \overset{\eta_\varphi}{\rightarrowtail} \langle\varphi\rangle \overset{g}{\to} A$ of a $\mathbf{C}$-arrow $\varphi\colon L \to A$, let $L \overset{m_L}{\rightarrowtail} X$ be a mono in the language $\mathcal{L}(L \overset{\eta_\varphi}{\rightarrowtail} \langle\varphi\rangle)$, i.e., it holds that $(L \overset{m_L}{\rightarrowtail} X) \in \mathcal{L}(L \overset{\eta_\varphi}{\rightarrowtail} \langle\varphi\rangle)$. Spelling out Definition 9.17 we obtain the following commuting diagram where the square is a pullback:

  $$\begin{tikzcd}
  L \arrow[r, tail, "m_L"] \arrow[d, tail, "id_L"'] \arrow[dr, phantom, "\text{(PB)}"] & X \arrow[d, "f"] \arrow[dr, dashed, "\psi"] & \\
  L \arrow[r, tail, "\eta_\varphi"] \arrow[rr, bend right, "\varphi"'] & \langle\varphi\rangle \arrow[r, "g"] & A
  \end{tikzcd}$$

  Then we define $\psi = g \circ f\colon X \to A$ and observe that the following equation holds:

  $$\varphi = g \circ \eta_\varphi = g \circ \eta_\varphi \circ id_L = g \circ f \circ m_L = \psi \circ m_L$$

- ($\supseteq$) Let the mono $L \overset{m_L}{\rightarrowtail} X$ be a factorization of the $\mathbf{C}$-arrow $\varphi\colon L \to A$, i.e. there exists an arrow $\psi\colon X \to A$ such that $\varphi = \psi \circ m_L$. By terminality of the materialization $L \overset{\eta_\varphi}{\rightarrowtail} \langle\varphi\rangle \to A$ there exists an arrow $X \to \langle\varphi\rangle$ such that the following diagram commutes and the square is a pullback:

$$\begin{tikzcd}
L \arrow[r, tail, "m_L"] \arrow[d, tail, "id_L"'] \arrow[dr, phantom, "\text{(PB)}"] & X \arrow[d, dashed] \arrow[dr, "\psi"] & \\
L \arrow[r, tail, "\eta_\varphi"] \arrow[rr, bend right, "\varphi"'] & \langle\varphi\rangle \arrow[r] & A
\end{tikzcd}$$

Therefore $(L \overset{m_L}{\rightarrowtail} X) \in \mathcal{L}(L \overset{\eta_\varphi}{\rightarrowtail} \langle\varphi\rangle)$ holds.

□

Proposition 9.25 (Construction of the rewritable materialization). *Let $\varphi\colon L \to A$ be an arrow and let $\varphi_L\colon I \rightarrowtail L$ be a mono of $\mathbf{C}$. Then the* rewritable materialization of φ w.r.t. φ_L *exists and can be constructed as the following factorization $L \overset{n_L}{\rightarrowtail} \langle\!\langle\varphi, \varphi_L\rangle\!\rangle \xrightarrow{\psi\circ\alpha} A$ of φ. In the diagram shown below to the left, F is obtained as the final pullback complement of $I \overset{\varphi_L}{\rightarrowtail} L \rightarrowtail \langle\varphi\rangle$, where $L \rightarrowtail \langle\varphi\rangle \overset{\psi}{\to} A$ is the materialization of φ (Definition 9.18). Next the diagram shown below to the right $L \overset{n_L}{\rightarrowtail} \langle\!\langle\varphi, \varphi_L\rangle\!\rangle \overset{\beta}{\leftarrowtail} F$ is the pushout of the span $L \overset{\varphi_L}{\leftarrowtail} I \rightarrowtail F$ and α is the resulting mediating arrow.*

$$\begin{tikzcd}
 & L \arrow[dl, "\varphi"'] \arrow[d, tail] \arrow[dr, phantom, "\text{(FPBC)}"] & I \arrow[l, tail, "\varphi_L"'] \arrow[d, tail] \\
A & \langle\varphi\rangle \arrow[l, "\psi"] & F \arrow[l, tail]
\end{tikzcd} \quad \text{(A.8)}$$

$$\begin{tikzcd}
 & L \arrow[dl, "\varphi"'] \arrow[d, tail] & L \arrow[l, tail, "id_L"'] \arrow[d, tail, "n_L"'] \arrow[dr, phantom, "\text{(PO)}"] & I \arrow[l, tail, "\varphi_L"'] \arrow[d, tail] \\
A & \langle\varphi\rangle \arrow[l, "\psi"] & \langle\!\langle\varphi, \varphi_L\rangle\!\rangle \arrow[l, dashed, tail, "\alpha"'] & F \arrow[l, tail, "\beta"'] \arrow[ll, tail, bend left]
\end{tikzcd} \quad \text{(A.9)}$$

Proof. First note that in Diagram (A.8), F is obtained as the final pullback complement of $I \overset{\varphi_L}{\rightarrowtail} L \rightarrowtail \langle\varphi\rangle$, where $L \rightarrowtail \langle\varphi\rangle \overset{\psi}{\to} A$ is the materialization of φ (Definition 9.18). Arrow $I \rightarrowtail F$ is monic because it is reflected, while $F \rightarrowtail \langle\varphi\rangle$ is monic by properties of final pullback complements since $\varphi_L\colon I \rightarrowtail L$ is monic (see [CH+06]).

In Diagram (A.9) $L \overset{n_L}{\rightarrowtail} \langle\!\langle\varphi, \varphi_L\rangle\!\rangle \overset{\beta}{\leftarrowtail} F$ is the pushout of the span $L \overset{\varphi_L}{\leftarrowtail} I \rightarrowtail F$. Since the right square is a pushout and the outer square commutes, there is a unique arrow $\alpha\colon \langle\!\langle\varphi, \varphi_L\rangle\!\rangle \to \langle\varphi\rangle$ making the diagram commute. Note that arrow $L \overset{n_L}{\rightarrowtail} \langle\!\langle\varphi, \varphi_L\rangle\!\rangle$ is indeed monic, as pushouts preserve monos in a topos, and α is monic because topoi have effective unions. Therefore the rewritable materialization $L \overset{n_L}{\rightarrowtail} \langle\!\langle\varphi, \varphi_L\rangle\!\rangle \xrightarrow{\psi\circ\alpha} A$ is an object of $\mathbf{Mat}_\varphi$, and clearly it is also an object of the subcategory $\mathbf{Mat}_\varphi^{\varphi_L}$, as by Diagram (A.9) $I \overset{\varphi_L}{\rightarrowtail} L \overset{n_L}{\rightarrowtail} \langle\!\langle\varphi, \varphi_L\rangle\!\rangle$ has a pushout complement.

We next prove that the left square of Diagram (A.9) is a pullback, to show that α is the unique arrow from the rewritable materialization to the materialization in $\mathbf{Mat}_\varphi$. Let the diagram below to the right be given. We already know that the inner square commutes and therefore $\eta_\varphi \circ id_L = \alpha \circ n_L$. We will now show that the pullback property for the inner square holds:

For any other object X and two arrows $f\colon X \to L$ and $g\colon X \to \langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ where the outer square commutes, there exists a unique arrow $h\colon X \to L$ such that $f = id_L \circ h$ and $g = n_L \circ h$. It is clear that $h = f$ by this assumption. Since α is a mono, it is a left-cancellative arrow e.g. for any two arrows $f_1, f_2\colon X \to \langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ we get that $\alpha \circ f_1 = \alpha \circ f_2$ implies $f_1 = f_2$.

$$\begin{tikzcd}
X \arrow[dr, dashed, "h"] \arrow[drr, bend left, "f"] \arrow[ddr, bend right, "g"'] & & \\
& L \arrow[r, tail, "id_L"] \arrow[d, tail, "n_L"'] \arrow[dr, phantom, "\text{(PB)}"] & L \arrow[d, tail, "\eta_\varphi"] \\
& \langle\!\langle \varphi, \varphi_L \rangle\!\rangle \arrow[r, tail, "\alpha"'] & \langle \varphi \rangle
\end{tikzcd}$$

We obtain the following equation: $\alpha \circ g = \eta_\varphi \circ f = \eta_\varphi \circ id_L \circ h = \alpha \circ n_L \circ h$, which implies that $g = n_L \circ h$ since α is a mono. Hence the inner square is a pullback. Now let $L \overset{p}{\rightarrowtail} X \overset{q}{\to} A$ be an object of $\mathbf{Mat}_\varphi^{\varphi_L}$, i.e. a factorization of φ such that the pushout complement of $I \overset{\varphi_L}{\rightarrowtail} L \overset{p}{\rightarrowtail} X$ exists, and let $I \rightarrowtail C \rightarrowtail X$ be such a pushout complement. Then the following Diagram (A.10) commutes, where $g\colon X \to \langle \varphi \rangle$ is the unique arrow making the left square a pullback by finality of the materialization, and the right square is a pullback because it is a pushout along a mono. From the pasting lemma (pullback version) we can conclude that the composed square is a pullback as well.

$$\begin{tikzcd}
& L \arrow[dl, "\varphi"'] \arrow[d, tail, "\eta_\varphi"'] \arrow[dr, phantom, "\text{(PB)}"] & L \arrow[l, tail] \arrow[d, tail, "p"] \arrow[dr, phantom, "\text{(PB)}"] & I \arrow[l, tail, "\varphi_L"'] \arrow[d, tail] \\
A & \langle \varphi \rangle \arrow[l, "\psi"'] & X \arrow[l, "g"'] \arrow[ll, bend left, "q"] & C \arrow[l, tail]
\end{tikzcd} \tag{A.10}$$

$$\begin{tikzcd}
& & I \arrow[dll, bend right, tail, "\varphi_L"'] \arrow[dl, tail] \arrow[ddd, tail] \\
L \arrow[d, tail, "\eta_\varphi"'] \arrow[dr, phantom, "\text{(FPBC)}"] & I \arrow[l, tail, "\varphi_L"'] \arrow[d, tail] & \\
\langle \varphi \rangle & F \arrow[l, tail] & \\
& & C \arrow[ul, dashed, "\gamma"] \arrow[ull, bend left]
\end{tikzcd} \tag{A.11}$$

Combining the outer pullback of Diagram (A.10) with the final pullback complement of Diagram (A.8) we get Diagram (A.11). By Definition 9.13 there exists a unique arrow γ such that the diagram commutes (especially the lower triangle and the square to the right).

By composing the arrows $\gamma\colon C \to F$ from Diagram (A.11) and $\beta\colon F \rightarrowtail \langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ from Diagram (A.9) we get the arrow $c = \beta \circ \gamma\colon C \to \langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ shown in the commuting Diagram (A.12) where the right square is a pushout. The universal property of pushouts gives us a unique mediating arrow $\delta : X \to \langle\!\langle \varphi, \varphi_L \rangle\!\rangle$. To show that δ defines an arrow in $\mathbf{Mat}_\varphi$ from $L \overset{p}{\rightarrowtail} X \overset{q}{\to} A$ to the rewritable materialization $L \overset{n_L}{\rightarrowtail} \langle\!\langle \varphi, \varphi_L \rangle\!\rangle \overset{\psi \circ \alpha}{\longrightarrow} A$ we need to prove that $q = \psi \circ \alpha \circ \delta$ (which is easily checked by diagram chasing) and that the left square is a pullback.

$$\begin{tikzcd}
L \arrow[d, tail, "n_L"'] \arrow[dr, phantom, "\text{(?)}"] & L \arrow[l, tail] \arrow[d, tail, "p"] \arrow[dr, phantom, "\text{(PO)}"] & I \arrow[l, tail] \arrow[d, tail] \\
\langle\!\langle \varphi, \varphi_L \rangle\!\rangle & X \arrow[l, dashed, "\delta"'] & C \arrow[l, tail] \arrow[ll, bend left, "c"]
\end{tikzcd} \tag{A.12}$$

$$\begin{tikzcd}
L \arrow[d, tail] \arrow[dr, phantom, "\text{(PB)}"] & L \arrow[l, tail] \arrow[d, tail, "n_L"] \arrow[dr, phantom, "\text{(PB)}"] & L \arrow[l, tail] \arrow[d, tail, "p"] \\
\langle \varphi \rangle & \langle\!\langle \varphi, \varphi_L \rangle\!\rangle \arrow[l, tail, "\alpha"'] & X \arrow[l, "\delta"'] \arrow[ll, bend left, "g"]
\end{tikzcd} \tag{A.13}$$

In order to show that the square marked (?) is a pullback we consider Diagram (A.13). The left square is a pullback as we have shown earlier, and the outer square is a pullback by Diagram (A.10). From the pasting lemma (pullback version) we can conclude that the right square is a pullback. Also note that the diagram clearly commutes as the three arrows at the bottom are all unique. □

Proposition 9.27 (Language of the rewritable materialization). *Assume there is a production $p\colon L \overset{\varphi_L}{\leftarrowtail} I \overset{\varphi_R}{\rightarrowtail} R$ and let $L \overset{n_L}{\rightarrowtail} \langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ be the match for the rewritable materialization for φ and φ_L. Then we have*

$$\mathcal{L}(L \overset{n_L}{\rightarrowtail} \langle\!\langle \varphi, \varphi_L \rangle\!\rangle) = \{L \overset{m_L}{\rightarrowtail} X \mid \exists \psi\colon (X \to A).\ (\varphi = \psi \circ m_L \wedge X \overset{p,m_L}{\Longrightarrow})\}$$

where $X \overset{p,m_L}{\Longrightarrow}$ denotes that the object X can be rewritten.

Proof. We show that the two sets of arrows are included in one another:

- ($\supseteq$) Let $L \overset{m_L}{\rightarrowtail} X$ such that there exists an arrow ψ with $\varphi = \psi \circ m_L$ and $X \overset{p,m_L}{\Longrightarrow}$. Then $L \overset{m_L}{\rightarrowtail} X \overset{\psi}{\to} A$ is an object of the materialization category of rewritable objects (since the production can be applied, the pushout complement exists) and we obtain a unique arrow $X \to \langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ that creates a pullback L, L, X, A. Hence $m_L \in \mathcal{L}(L \overset{n_L}{\rightarrowtail} \langle\!\langle \varphi, \varphi_L \rangle\!\rangle)$.

- ($\subseteq$) Assume that $m_L \in \mathcal{L}(L \overset{n_L}{\rightarrowtail} \langle\!\langle \varphi, \varphi_L \rangle\!\rangle)$. This implies the existence of an arrow $X \to \langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ such that the left square in Diagram (A.14) is a pullback. The arrow $\psi\colon X \to A$ is given by composing $X \to \langle\!\langle \varphi, \varphi_L \rangle\!\rangle \overset{\alpha}{\rightarrowtail} \langle \varphi \rangle \to A$ and by retracing the construction of $\langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ (see Proposition 9.25) it can be shown that $\varphi = \psi \circ m_L$.

  Furthermore we constructed the outer square in Diagram (A.14) as a pushout, which is therefore also a pullback.

$$\begin{tikzcd}
L \arrow[d, tail] & L \arrow[l, tail] \arrow[d, tail, "m_L"] \arrow[ld, phantom, "\text{(PB)}"] & I \arrow[l, tail, "\varphi_L"'] \arrow[d, tail] \\
\langle\!\langle \varphi, \varphi_L \rangle\!\rangle & X \arrow[l] & F \arrow[ll, tail, bend left]
\end{tikzcd} \tag{A.14}$$

$$\begin{tikzcd}
 & & I \arrow[lld, tail, bend right, "m_L \circ \varphi_L"'] \arrow[ld, tail, dashed] \arrow[ld, phantom, "="] \arrow[ld, tail, bend left, start anchor=south, end anchor=east, phantom] \\
X \arrow[d] & C \arrow[l, tail] \arrow[d] \arrow[ld, phantom, "\text{(PB)}"] \arrow[r, phantom, "="] & \\
\langle\!\langle \varphi, \varphi_L \rangle\!\rangle & F \arrow[l, tail] &
\end{tikzcd} \tag{A.15}$$

  Now we take the pullback of $X \to \langle\!\langle \varphi, \varphi_L \rangle\!\rangle \leftarrowtail F$ and obtain the pullback object C with the corresponding arrows (See Diagram (A.15)). Since the outer square commutes, we get a unique arrow $I \rightarrowtail C$ due to the property of pullbacks. Note that $I \rightarrowtail C$ is a mono since $I \rightarrowtail F$ is a mono. All we need to show is that C is the pushout complement for our rewritable object X.

  In order to show that it is a pushout we consider the diagram to the right. The bottom square is a Van Kampen square[4], furthermore the left square is trivially a pullback, the front square is a pullback according to Diagram (A.14) and the right square is a pullback by construction (see Diagram (A.15)). Then it follows from classical pullback splitting that the back square is also a pullback. Finally it follows from the properties of adhesive categories that the top square is a pushout.

$$\begin{tikzcd}
 & I \arrow[rr, tail] \arrow[ld, tail] \arrow[dd, tail] & & C \arrow[ld] \arrow[dd] \\
L \arrow[rr, tail] \arrow[dd, tail] & & X \arrow[dd] & \\
 & I \arrow[rr, tail] \arrow[ld, tail] & & F \arrow[ld, tail] \\
L \arrow[rr, tail] & & \langle\!\langle \varphi, \varphi_L \rangle\!\rangle &
\end{tikzcd}$$

[4] Since every topos is adhesive, the Van Kampen square property holds. For more details see [LS05].

Therefore X can be rewritten. The existence of the pushout complement is guaranteed using the described construction. This completes the proof. □

Proposition 9.28 (Rewriting abstract matches). *Let a match $n_L\colon L \rightarrowtail \tilde{A}$ and a production $p\colon L \leftarrowtail I \rightarrowtail R$ be given. Assume that $\tilde{A}$ is rewritten along the match n_L, i.e., $(L \overset{n_L}{\rightarrowtail} \tilde{A}) \overset{p}{\Rrightarrow} (R \overset{n_R}{\rightarrowtail} B)$. Then*

$$\mathcal{L}(R \overset{n_R}{\rightarrowtail} B) = \{R \overset{m_R}{\rightarrowtail} Y \mid \exists (L \overset{m_L}{\rightarrowtail} X) \in \mathcal{L}(L \overset{n_L}{\rightarrowtail} \tilde{A})\colon\ ((L \overset{m_L}{\rightarrowtail} X) \overset{p}{\Rrightarrow} (R \overset{m_R}{\rightarrowtail} Y))\}$$

Proof.

- ($\subseteq$) Assume that $(L \overset{n_L}{\rightarrowtail} \tilde{A}) \overset{p}{\Rrightarrow} (R \overset{n_R}{\rightarrowtail} B)$ and let $m_R \in \mathcal{L}(R \overset{n_R}{\rightarrowtail} B)$ where $m_R\colon R \rightarrowtail Y$. That is we have the diagram below, where the bottom squares are pushouts and the remaining squares are pullbacks (the squares in the back are actually pushouts as well).

$$\begin{tikzcd}
 & L \arrow[dd, "id_L"] & & I \arrow[ll, tail] \arrow[rr, tail] \arrow[dd, "id_I"] & & R \arrow[dl, tail, "m_R"'] \arrow[dd, "id_R"] \\
 & & & & Y \arrow[dd] & \\
 & L \arrow[dl, tail, "n_L"'] & & I \arrow[ll, tail] \arrow[rr, tail] \arrow[dl, tail] & & R \arrow[dl, tail, "n_R"] \\
\tilde{A} & & C \arrow[ll, tail] \arrow[rr, tail] & & B &
\end{tikzcd}$$

  Now take the pullback of $C \rightarrowtail B$ and $Y \to B$, obtaining Z, which gives us $I \to Z$ as mediating arrow into the pullback object (see diagram below). In the right cube the right square is a pullback, the back square is trivially pullback and the front square is a pullback by construction. This means that the left square is also a pullback by pullback splitting. Due to the Van Kampen square property this implies that the top square is a pushout. Since all pushouts along monos are pullbacks in adhesive categories, the arrow $I \to Z$ must be a mono.

  Finally, take the pushout of $I \rightarrowtail Z$ and $I \rightarrowtail L$, resulting in X, which give us $X \to \tilde{A}$ as a mediating arrow.

$$\begin{tikzcd}
 & L \arrow[dl, tail, "m_L"'] \arrow[dd, "id_L"] & & I \arrow[ll, tail] \arrow[rr, tail] \arrow[dl, tail] \arrow[dd, "id_I"] & & R \arrow[dl, tail, "m_R"'] \arrow[dd, "id_R"] \\
X \arrow[dd] & & Z \arrow[ll, tail] \arrow[rr, tail] \arrow[dd] & & Y \arrow[dd] & \\
 & L \arrow[dl, tail, "n_L"'] & & I \arrow[ll, tail] \arrow[rr, tail] \arrow[dl, tail] & & R \arrow[dl, tail, "n_R"] \\
\tilde{A} & & C \arrow[ll, tail] \arrow[rr, tail] & & B &
\end{tikzcd}$$

  This illustrates that $(L \overset{m_L}{\rightarrowtail} X) \overset{p}{\Rrightarrow} (R \overset{m_R}{\rightarrowtail} Y)$. Since in the left cube the back square is trivially a pullback and the right square is a pullback as well (see argument above), the front and left squares are pullbacks as well. This implies that $(L \overset{m_L}{\rightarrowtail} X) \in \mathcal{L}(L \overset{n_L}{\rightarrowtail} \tilde{A})$, as required.

- ($\supseteq$) Assume that $(L \overset{m_L}{\rightarrowtail} X) \overset{p}{\Rrightarrow} (R \overset{m_R}{\rightarrowtail} Y)$ and $(L \overset{m_L}{\rightarrowtail} X) \in \mathcal{L}(L \overset{n_L}{\rightarrowtail} \tilde{A})$ hold. Together with the fact that $(L \overset{n_L}{\rightarrowtail} \tilde{A}) \overset{p}{\Rrightarrow} (R \overset{n_R}{\rightarrowtail} B)$, this results in the diagram below (without the dotted arrows), where the top and bottom squares of the cubes are all pushouts and the vertical squares are pullbacks.

$$\begin{tikzcd}
& L \arrow[dl, tail, "m_L"] \arrow[dd, tail, "id_L"] & & I \arrow[ll, tail] \arrow[rr, tail] \arrow[dl, tail] \arrow[dd, tail, "id_I"] & & R \arrow[dl, tail, "m_R"] \arrow[dd, tail, "id_R"] \\
X \arrow[dd] & & Z \arrow[ll, tail] \arrow[rr, tail] \arrow[dd, dashed] & & Y \arrow[dd, dashed] & \\
& L \arrow[dl, tail, "n_L"] & & I \arrow[ll, tail] \arrow[rr, tail] \arrow[dl, tail] & & R \arrow[dl, tail, "n_R"] \\
\tilde{A} & & C \arrow[ll, tail] \arrow[rr, tail] & & B &
\end{tikzcd}$$

  Due to the Van Kampen square property and the fact that pushout complements of mono arrows are unique, the object Z can be constructed in two ways: either by taking the pullback of $X \to \tilde{A}$ and $C \rightarrowtail \tilde{A}$ or by taking the pushout complement of $I \rightarrowtail L$, $L \rightarrowtail X$ as shown above. Hence there must be an arrow $Z \to C$ arising from the pullback and the front and right square of the left cube are pullbacks as well.

  Now the arrow $Y \to B$ is obtained as a mediating arrow into the pushout object and the front and right faces of the right cube are again pullbacks. This implies that $(R \overset{m_R}{\rightarrowtail} Y) \in \mathcal{L}(R \overset{n_R}{\rightarrowtail} B)$, as desired. □

Corollary 9.30 (Co-match language of the rewritable materialization). *Let $\varphi\colon L \to A$ and a production $p\colon L \overset{\varphi_L}{\leftarrowtail} I \overset{\varphi_R}{\rightarrowtail} R$ be given. Assume that $\langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ is obtained as the rewritable materialization of φ and φ_L with the match $L \overset{n_L}{\rightarrowtail} \langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ (see Proposition 9.25) and let $(L \overset{n_L}{\rightarrowtail} \langle\!\langle \varphi, \varphi_L \rangle\!\rangle) \overset{p}{\Rrightarrow} (R \overset{n_R}{\rightarrowtail} B)$. Then*

$$\begin{aligned}\mathcal{L}(R \overset{n_R}{\rightarrowtail} B) = \{R \overset{m_R}{\rightarrowtail} Y \mid \exists (L \overset{m_L}{\rightarrowtail} X), (X \overset{\psi}{\to} A)\colon \\ (\varphi = \psi \circ m_L \wedge (L \overset{m_L}{\rightarrowtail} X) \overset{p}{\Rrightarrow} (R \overset{m_R}{\rightarrowtail} Y))\}\end{aligned}$$

Proof. Straightforward from Propositions 9.27 and 9.28. □

A.6. Proofs of Chapter 10

Lemma L.6. The multiplicity functor from Definition 8.6 satisfies the homorphism property, the pushout property, the adjunction property, the Beck-Chevalley property and the isomorphism property.

Proof.

Homomorphism property: Assume $\varphi\colon A \to B$ to be an injective graph morphism.

We first show that $\mathcal{B}^n_\varphi$ preserves the unit, which is a map $a\colon V_A \cup E_A \to \mathcal{M}_n$ with $a(x) = 0$ for all $x \in V_A \cup E_A$. For $y \in V_B \cup E_B$ we have the annotation $\mathcal{B}^n_\varphi(a)(y) = \sum_{\varphi(x)=y} a(x)$. Either y has a unique preimage x with $a(x) = 0$ and in this case the result is 0. Or y has no preimage, in which case we have the empty sum and the result is also 0.

Next, we show that $\mathcal{B}^n_\varphi$ preservers the monoid operation: let two annotations $a_1, a_2 \in V_A \cup E_A \to \mathcal{M}_n$ be given. Then for the monoid operation we have $\mathcal{B}^n_\varphi(a_1 + a_2)(y) = \sum_{\varphi(x)=y}(a_1(x) + a_2(x))$. We distinguish two cases:

- Either y has a unique preimage x and then the result is

$$a_1(x) + a_2(x) = \sum_{\varphi(x)=y} a_1(x) + \sum_{\varphi(x)=y} a_2(x) = \mathcal{B}^n_\varphi(a_1)(y) + \mathcal{B}^n_\varphi(a_2)(y)$$

- Or y has no preimage under φ and we obtain

$$0 = 0 + 0 = \sum_{\varphi(x)=y} a_1(x) + \sum_{\varphi(x)=y} a_2(x) = \mathcal{B}^n_\varphi(a_1)(y) + \mathcal{B}^n_\varphi(a_2)(y)$$

Preservation of subtraction can be shown analogously.

Note that preservation of the monoid operation (but not preservation of subtraction) holds for any (also non-injective) graph morphism.

Adjunction property: Assume that $\varphi\colon A \to B$ is an injective graph morphism.

- First we show that the right adjoint of $\mathcal{B}^n_\varphi\colon \mathcal{B}^n(A) \to \mathcal{B}^n(B)$ is the functor $\mathit{red}_\varphi\colon \mathcal{B}^n(B) \to \mathcal{B}^n(A)$ where for $b\colon V_B \cup E_B \to \mathcal{M}_n$ we have $\mathit{red}_\varphi(b)(x) = b(\varphi(x))$ (for $x \in V_A \cup E_A$). Clearly, red_φ is monotone.

  Furthermore for $a \in \mathcal{B}^n(A)$ and $x \in V_A \cup E_A$ we can show the following, using the fact that φ is injective:

$$\mathit{red}_\varphi(\mathcal{B}^n_\varphi(a))(x) = \mathcal{B}^n_\varphi(a)(\varphi(x)) = \sum_{\varphi(x')=\varphi(x)} a(x') = a(x)$$

  Finally for $b \in \mathcal{B}^n(B)$ and $y \in V_B \cup E_B$ we have:

$$\mathcal{B}^n_\varphi(\mathit{red}_\varphi(b))(y) = \sum_{\varphi(x)=y} \mathit{red}_\varphi(b)(x) = \sum_{\varphi(x)=y} b(\varphi(x))$$

$$= \left\{ \begin{array}{ll} b(y) & \text{if } y \in \mathit{img}(\varphi) \\ 0 & \text{otherwise} \end{array} \right\} \leq b(y)$$

- We have to show that red_φ is a monoid homomorphism that preserves subtraction.

  Let $b\colon V_B \cup E_B \to \mathcal{M}_n$ be the unit map that satisfies $b(y) = 0$ for all $y \in V_B \cup E_B$. Then $\mathit{red}_\varphi(b)(x) = b(\varphi(x)) = 0$ for all $x \in V_A \cup E_A$, i.e., $\mathit{red}_\varphi(b)$ is also the unit map.

  Furthermore for $b_1, b_2\colon V_B \cup E_B \to \mathcal{M}_n$ we have

  $$\begin{aligned} & \mathit{red}_\varphi(b_1 + b_2)(x) = (b_1 + b_2)(\varphi(x)) = b_1(\varphi(x)) + b_2(\varphi(x)) \\ = \quad & \mathit{red}_\varphi(b_1)(x) + \mathit{red}_\varphi(b_2)(x) \end{aligned}$$

  Preservation of subtraction can be shown analogously.

- red_φ preserves standard annotations:
  $\mathit{red}_\varphi(s_B)(x) = s_B(\varphi(x)) = 1 = s_A(x)$.

Pushout property: Assume that we have a pushout as in Definition 10.8 (pushout property) and let $d \in \mathcal{B}^n(D)$. We have to show that

$$d = \mathcal{B}^n_{\psi_1}(\mathit{red}_{\psi_1}(d)) + (\mathcal{B}^n_{\psi_2}(\mathit{red}_{\psi_2}(d)) - \mathcal{B}^n_\eta(\mathit{red}_\eta(d)))$$

Let $y \in V_D \cup E_D$, then we obtain:

$$\begin{aligned} & \mathcal{B}^n_{\psi_1}(\mathit{red}_{\psi_1}(d))(y) + (\mathcal{B}^n_{\psi_2}(\mathit{red}_{\psi_2}(d))(y) - \mathcal{B}^n_\eta(\mathit{red}_\eta(d))(y)) \\ = \quad & \sum_{\psi_1(x_1)=y} d(\psi_1(x_1)) + (\sum_{\psi_2(x_2)=y} d(\psi_2(x_2)) - \sum_{\eta(x_0)=y} d(\eta(x_0))) \end{aligned}$$

We distinguish the following cases:

- y has a (unique) preimage x_1 under ψ_1, but no preimage under ψ_2. This means that y has no preimage under η as well. In this case we obtain

  $$\sum_{\psi_1(x_1)=y} d(\psi_1(x_1)) = d(y), \qquad \sum_{\psi_2(x_2)=y} d(\psi_2(x_2)) = \sum_{\eta(x_0)=y} d(\eta(x_0)) = 0,$$

  from which the required equality follows.

- y has a (unique) preimage x_2 under ψ_2, but no preimage under ψ_1. This case is analogous to the previous one.

- y has a (unique) preimage x_1 under ψ_1 and a (unique) preimage x_2 under ψ_2. Hence it must also have a (unique) preimage x_0 under η such that $\varphi_1(x_0) = x_1$, $\varphi_2(x_0) = x_2$. In this case we obtain

  $$\sum_{\psi_1(x_1)=y} d(\psi_1(x_1)) = \sum_{\psi_2(x_2)=y} d(\psi_2(x_2)) = \sum_{\eta(x_0)=y} d(\eta(x_0)) = d(y),$$

  yielding the result $d(y) + (d(y) - d(y)) = d(y)$.

Beck-Chevalley property: First, observe that since the square from Definition 10.8 (Beck-Chevalley property) is a pullback, we can assume that the elements (vertices and edges) of A are as follows:

$$V_A \cup E_A = \{(x_1, x_2) \mid x_1 \in V_B \cup E_B, x_2 \in V_C \cup E_C, \psi_1(x_1) = \psi_2(x_2)\}$$

Now let $b\colon V_B \cup E_B \to \mathcal{M}_n$ and $x_2 \in V_C \cup E_C$. Then we have:

$$\begin{aligned}
&\mathcal{A}_{\varphi_2}(red_{\varphi_1}(b))(x_2) = \sum_{\varphi_2((x_1',x_2'))=x_2} b(\varphi_1((x_1',x_2'))) \\
= &\sum_{\psi_1(x_1')=\psi_2(x_2)} b(\varphi_1((x_1',x_2))) = \sum_{\psi_1(x_1')=\psi_2(x_2)} b(x_1) = \mathcal{A}_{\psi_1}(b)(\psi_2(x_2)) \\
= &\ red_{\psi_2}(\mathcal{A}_{\psi_1}(b))(x_2)
\end{aligned}$$

Isomorphism property: Assume that $\varphi\colon X[s_X, s_X] \to Y[s_Y, s_Y]$ is a legal morphism. Then, since the standard annotation s_Y is a lower and upper bound, every element Y must have exactly one preimage in X under φ. This is equivalent to the fact that φ is an isomorphism. □

Lemma L.7. The path annotation functor from Definition 8.8 satisfies the homorphism property and the pushout property for standard annotations.

Proof.

Homomorphism property: Assume $\varphi\colon A \to B$ to be an injective graph morphism.

First observe that $\mathcal{T}_\varphi(\emptyset) = \emptyset$.

Now let $P_0, P_1 \in \mathcal{T}(A)$, we have to show that $\mathcal{T}_\varphi(P_0+P_1) = \mathcal{T}_\varphi(P_0)+\mathcal{T}_\varphi(P_1)$.

(⊆) Let $(w_0, w_n) \in \mathcal{T}_\varphi(P_0 + P_1)$ where $w_0, w_n \in V_B$. Then w_0, w_n have (unique) preimages $v_0, v_n \in V_A$ with $\varphi(v_0) = w_0$, $\varphi(v_n) = w_n$ and $(w_0, w_n) \in (P_0 + P_1)$. Then there exist vertices $v_1, \dots, v_{n-1} \in V_A$ such that $(v_i, v_{i+1}) \in P_{j_i}$, $j_i \in \{0, 1\}$, $j_{i+1} = 1 - j_i$, $i \in \{0, \dots, n-1\}$. This implies that $(\varphi(v_i), \varphi(v_{i+1})) = (w_i, w_{i+1}) \in \mathcal{T}_\varphi(P_{j_i})$. By definition of the monoid operation $+$ we have $(w_0, w_n) \in (\mathcal{T}_\varphi(P_0) + \mathcal{T}_\varphi(P_1))$.

(⊇) Let $(w_0, w_n) \in (\mathcal{T}_\varphi(P_0) + \mathcal{T}_\varphi(P_1))$. Then there exist $w_1, \dots, w_{n-1} \in V_B$ such that $(w_i, w_{i+1}) \in \mathcal{T}_\varphi(P_{j_i})$ with $j_i \in \{0, 1\}$, $j_{i+1} = 1 - j_i$ where $i \in \{0, \dots, n-1\}$.

Hence there are preimages $v_0^{j_0}, v_1^{j_0}, v_1^{j_1}, \dots, v_{n-1}^{j_{n-1}}, v_n^{j_{n-1}} \in V_A$ of the w_i. In particular $\varphi(v_i^j) = w_i$ and $(v_i^{j_i}, v_{i+1}^{j_i}) \in P_{j_i}$. Then since we have $\varphi(v_i^{j_i}) = w_i = \varphi(v_i^{j_{i+1}})$ and φ is injective, we can infer $v_i^{j_i} = v_i^{j_{i+1}}$. This means that $(v_0, v_n) \in (P_0+P_1)$ by definition of the monoid operation $+$. Finally, this implies that $(w_0, w_n) = (\varphi(v_0), \varphi(v_n)) \in \mathcal{T}_\varphi(P_0 + P_1)$.

Furthermore $\mathcal{T}_\varphi$ trivially preserves subtraction:
$\mathcal{T}_\varphi(P_0 - P_1) = \mathcal{T}_\varphi(P_0) = \mathcal{T}_\varphi(P_0) - \mathcal{T}_\varphi(P_1)$.

Pushout property for standard annotations: Consider the pushout of injective graph morphisms depicted below where $\eta = \psi_0 \circ \varphi_0 = \psi_1 \circ \varphi_1$:

$$\begin{tikzcd}
A \arrow[r, tail, "\varphi_1"] \arrow[d, tail, "\varphi_0"'] \arrow[dr, dashed, tail, "\eta"'] & B_1 \arrow[d, "\psi_1"] \\
B_0 \arrow[r, tail, "\psi_0"'] & D
\end{tikzcd}$$

We have to show that

$$s_D = \mathcal{T}_{\psi_1}(s_{B_0}) + (\mathcal{T}_{\psi_2}(s_{B_1}) - \mathcal{T}_\eta(s_A)) = \mathcal{T}_{\psi_1}(s_{B_0}) + \mathcal{T}_{\psi_2}(s_{B_1})$$

(⊆) Let $(v_0, v_n) \in s_D$. This means that there exists a path in graph D, consisting of edges $e_0, \dots, e_{n-1} \in E_D$, from v_0 to v_n. In particular $s(e_i) = v_i$, $t(e_i) = v_{i+1}$.

Since D is a pushout, each edge has a preimage in B_0 or in B_1 (or in both). Hence we can group consecutive edges according to the origin of their preimages and we can (possibly non-uniquely) choose indices $i_0 = 0, \dots, i_k = n+1$ such that $e_{i_\ell}, \dots, e_{i_{\ell+1}-1}$ have preimages in B_{j_ℓ} where $\ell \in \{0, \dots, k-1\}$, $j_\ell \in \{0,1\}$ and $j_{\ell+1} = 1 - j_\ell$.

Now assume that the preimages of the e_i are $f_0, \dots, f_{n-1} \in E_{B_0} \cup E_{B_1}$ where $\psi_0(f_i) = e_i$ and $\psi_1(f_i) = e_i$ whenever ψ_0 respectively ψ_1 are defined on f_i.

Since ψ_0, ψ_1 are injective, the edges $f_{i_\ell}, \dots, f_{i_{\ell+1}-1}$ form a path in B_{j_ℓ}, hence $(s(f_{i_\ell}), t(f_{i_{\ell+1}-1})) \in s_{B_{j_\ell}}$. This implies that

$$\begin{aligned} & (v_{i_\ell}, v_{i_{\ell+1}}) = (s(e_{i_\ell}), t(e_{i_{\ell+1}-1})) = (s(\psi_{j_\ell}(f_{i_\ell})), t(\psi_{j_\ell}(f_{i_{\ell+1}-1}))) \\ = \; & (\psi_{j_\ell}(s(f_{i_\ell})), \psi_{j_\ell}(t(f_{i_{\ell+1}-1}))) \in \mathcal{T}_{\psi_{j_\ell}}(s_{B_{j_\ell}}) \end{aligned}$$

Hence, by the definition of the monoid operation $+$ we can infer that $(v_0, v_n) \in \mathcal{T}_{\psi_1}(s_{B_0}) + \mathcal{T}_{\psi_2}(s_{B_1})$.

(⊇) Let $(v_0, v_n) \in \mathcal{T}_{\psi_1}(s_{B_0}) + \mathcal{T}_{\psi_2}(s_{B_1})$. Hence there are $v_1, \dots, v_{n-1} \in V_D$ such that $(v_i, v_{i+1}) \in \mathcal{T}(s_{B_{j_i}})$ with $j_i \in \{0,1\}$, $j_{i+1} = 1 - j_i$ where $i \in \{0, \dots, n-1\}$.

This means that there are preimages $w_0^{j_0}, w_1^{j_0}, w_1^{j_1}, \dots, w_{n-1}^{j_{n-1}}, w_n^{j_{n-1}}$ of the v_i. In particular $w_i^j \in V_{B_j}$ and $\psi_j(w_i^j) = v_i$. Furthermore there exists a path from $w_i^{j_i}$ to $w_{i+1}^{j_i}$ in B_{j_i}. Hence there must also be a path from $v_i = \psi^{j_i}(w_i^{j_i})$ to $v_{i+1} = \psi^{j_i}(w_{i+1}^{j_i})$ in D. This in turn implies that there is a path from v_1 to v_n in D and hence $(v_1, v_n) \in D$. □

Lemma L.8. The local annotation functor from Definition 10.6 satisfies the homorphism property and the pushout property for standard annotations.

Proof.

Homomorphism property: Assume $\varphi\colon A \to B$ to be an injective graph morphism.

We first show that $\mathcal{S}_\varphi^n$ preserves the unit, which is a map $a\colon V_A \to \mathcal{M}_n$ with $a(v) = 0$ for all $v \in V_A$. For $w \in V_B$ we have $\mathcal{S}_\varphi^n(a)(w) = \bigvee_{\varphi(v)=w} a(v)$. Either w has a unique preimage v with $a(v) = 0$ and in this case the result is 0. Or w has no preimage, in which case we have the empty supremum and the result is also 0.

We show that $\mathcal{S}_\varphi^n$ preservers the monoid operation: let $a_1, a_2 \in V_A \to \mathcal{M}_n$. Then we have $\mathcal{S}_\varphi^n(a_1 + a_2)(w) = \bigvee_{\varphi(v)=w}(a_1(v) + a_2(v))$. We distinguish two cases:

- Either w has a unique preimage v and then the result is

$$a_1(v) + a_2(v) = \bigvee_{\varphi(v)=w} a_1(v) + \bigvee_{\varphi(v)=w} a_2(v) = \mathcal{S}_\varphi^n(a_1)(w) + \mathcal{S}_\varphi^n(a_2)(w)$$

- Or w has no preimage under φ and we obtain

$$0 = 0 + 0 = \bigvee_{\varphi(v)=w} a_1(v) + \bigvee_{\varphi(v)=w} a_2(v) = \mathcal{S}^n_\varphi(a_1)(w) + \mathcal{S}^n_\varphi(a_2)(w)$$

Preservation of subtraction can be shown analogously.

Pushout property for standard annotations: In the following we will use the function $out\colon V \to \mathcal{M}_n$ to assign to a vertex $v \in V$ its out-degree, respectively m if the out-degree is larger than n.

Assume that we have a pushout as in Definition 10.8. We show:

$$s_D = \mathcal{S}^n_{\psi_1}(s_B) + (\mathcal{S}^n_{\psi_2}(s_C) - \mathcal{S}^n_\eta(s_A))$$

Now let $w \in V_D$ and we distinguish the following cases:

- w has a (unique) preimage under ψ_1, but no preimage under ψ_2. This means that w has no preimage under η as well. In this case we have $out(w) = out(v)$ and furthermore:

$$s_D(w) = out(w) = out(v) = s_B(v) = \bigvee_{\psi_1(v)=w} s_B(v) = \mathcal{S}^n_\varphi(s_B)(w)$$

  Also $\mathcal{S}^n_{\psi_2}(s_C)(w) = 0$ and $\mathcal{S}^n_\eta(s_A)(w) = 0$ which completes this case.
- w has a (unique) preimage under ψ_2, but no preimage under ψ_1. This case is analogous to the previous case.
- w has a (unique) preimage v_1 under ψ_1 and a (unique) preimage v_2 under ψ_2. Hence it must also have a (unique) preimage v_0 under η such that $\varphi_1(v_0) = v_1$, $\varphi_2(v_0) = v_2$.

  Due to the properties of a pushout $out(w) = out(v_1) + (out(v_2) - out(v_0))$ holds. (Note that due to the placement of the brackets, the left-hand side equals m if and only if the right-hand side equals m.)

  Hence we obtain:

$$\begin{aligned} & s_D(w) = out(w) = out(v_1) + (out(v_2) - out(v_0)) \\ = \;& s_B(v_1) + (s_C(v_2) - s_A(v_0)) \\ = \;& \bigvee_{\psi_1(v)=w} s_B(v) + (\bigvee_{\psi_2(v)=w} s_C(v) - \bigvee_{\eta(v)=w} s_A(v)) \\ = \;& \mathcal{S}^n_{\psi_1}(s_B)(w) + (\mathcal{S}^n_{\psi_2}(s_C)(w) - \mathcal{S}^n_\eta(s_A)(w)) \end{aligned}$$ □

Lemma L.9.

1. The pushout property for standard annotations implies that for every mono $\varphi\colon A \rightarrowtail B$ we have $\mathcal{A}_\varphi(s_A) \le s_B$.
2. The adjunction property and the Beck-Chevalley property imply that $red_\varphi(\mathcal{A}_\varphi(a)) = a$ for $\varphi\colon A \rightarrowtail B$, $a \in \mathcal{A}(A)$.
3. The pushout property and the adjunction property imply the pushout property for standard annotations.
4. The adjunction property implies $red_{\varphi\circ\psi} = red_\psi \circ red_\varphi$ for $A \overset{\psi}{\rightarrowtail} B \overset{\varphi}{\rightarrowtail} C$.

Proof.

1. Consider the pushout below.

$$\begin{tikzcd} A \arrow[r, tail, "\varphi"] \arrow[d, tail, "id_A"'] \arrow[dr, dashed, "\varphi"] & B \arrow[d, tail, "id_B"] \\ A \arrow[r, tail, "\varphi"'] & B \end{tikzcd}$$

   According to the pushout property for standard annotations we have

$$s_B = \mathcal{A}_\varphi(s_A) + (\mathcal{A}_{id_B}(s_B) - \mathcal{A}_\varphi(s_A)) \geq \mathcal{A}_\varphi(s_A),$$

   since $\mathcal{A}_{id_B}(s_B) - \mathcal{A}_\varphi(s_A) \geq 0$ (0 is the bottom element).

2. First, consider the identity morphism $id_A \colon A \rightarrowtail A$: then, for $a \in \mathcal{A}(A)$ we have $a \leq red_{id_A}(\mathcal{A}_{id_A}(a)) = red_{id_A}(a)$ and $red_{id_A}(a) = \mathcal{A}_{id_A}(red_{id_A}(a)) \leq a$. Hence $red_{id_A}(a) = a$.

   Since $\varphi \colon A \to B$ is a mono, the following diagram is a pullback.

$$\begin{tikzcd} A \arrow[r, tail, "id_A"] \arrow[d, tail, "id_A"'] \arrow[dr, phantom, "\text{(PB)}"] & A \arrow[d, tail, "\varphi"] \\ A \arrow[r, tail, "\varphi"'] & B \end{tikzcd}$$

   From the Beck-Chevalley property it follows that

$$red_\varphi(\mathcal{A}_\varphi(a)) = red_{id_A}(\mathcal{A}_{id_A}(a)) \ = \ a.$$

3. Consider a pushout of A, B, C, D as in the pushout property for standard annotations with $\eta = \psi_1 \circ \varphi_1 = \psi_2 \circ \varphi_2$. Due to the pushout property and the adjunction property we have

$$\begin{aligned} s_D &= \mathcal{A}_{\psi_1}(red_{\psi_1}(s_D)) + (\mathcal{A}_{\psi_2}(red_{\psi_2}(s_D)) - \mathcal{A}_\eta(red_\eta(s_D))) \\ &= \mathcal{A}_{\psi_1}(s_B) + (\mathcal{A}_{\psi_2}(s_C) - \mathcal{A}_\eta(s_A)) \end{aligned}$$

4. We have to show that $red_{\varphi\circ\psi}$, $red_\psi \circ red_\varphi$ are both left adjoints of $\mathcal{A}_{\varphi\circ\psi}$, then the result follows from the fact that adjoints are unique. This is obvious for $red_{\varphi\circ\psi}$ and in the other case we obtain for $c \in \mathcal{A}(C)$:

$$\begin{aligned} \mathcal{A}_{\varphi\circ\psi}(red_\psi(red_\varphi(c))) &= \mathcal{A}_\varphi(\mathcal{A}_\psi(red_\psi(red_\varphi(c)))) \\ &\leq \mathcal{A}_\varphi(red_\varphi(c)) \\ &\leq c \end{aligned}$$

   and similarly for the other inequality. □

Proposition 10.12 (Annotated rewritable materialization is terminal). *Let $p\colon L \overset{\varphi_L}{\leftarrowtail} I \overset{\varphi_R}{\rightarrowtail} R$ be a production and let $L \overset{m_L}{\rightarrowtail} X$ be the match of L in an object X such that $X \overset{p,m_L}{\Longrightarrow}$, i.e., X can be rewritten. Assume that X is abstracted by $A[a_1, a_2]$, witnessed by ψ. Let $\varphi = \psi \circ m_L$ and let $L \overset{n_L}{\rightarrowtail} \langle\!\langle \varphi, \varphi_L \rangle\!\rangle \overset{\psi'}{\to} A$ be the corresponding rewritable materialization. Then there exists an arrow ζ_A and a pair of annotations $(a'_1, a'_2) \in M$ for $\langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ (as described in Definition 10.11) such that the diagram below commutes and the square is a pullback in the underlying category. Furthermore the triangle consists of legal arrows. This means in particular that ζ_A is legal.*

$$\begin{tikzcd}
L[s_L, s_L] \arrow[r, tail, "m_L"] \arrow[d, tail, "id_L"'] \arrow[dr, phantom, "\text{(PB)}"] & X[s_X, s_X] \arrow[r, "\psi"] \arrow[d, "\zeta_A"] & A[a_1, a_2] \\
L[s_L, s_L] \arrow[r, tail, "n_L"'] & \langle\!\langle \varphi, \varphi_L \rangle\!\rangle[a'_1, a'_2] \arrow[ur, "\psi'"'] &
\end{tikzcd}$$

Proof. The existence of the arrow ζ_A follows from the fact that $L \rightarrowtail \langle\!\langle \varphi, \varphi_L \rangle\!\rangle \to A$ is the rewritable materialization (see Definition 9.24). This makes the left-hand square a pullback. We show that there exists a pair $(a'_1, a'_2) \in M$ (for M as in Definition 10.11) for which $a'_1 \le \mathcal{A}_{\zeta_A}(s_X) \le a'_2$.

It holds that $\mathcal{A}_{\psi'}(\mathcal{A}_{\zeta_A}(s_X)) = \mathcal{A}_{\psi}(s_X) \ge a_1$ and $\mathcal{A}_{\psi'}(\mathcal{A}_{\zeta_A}(s_X)) \le a_2$. Furthermore $\mathcal{A}_{n_L}(s_L) = \mathcal{A}_{\zeta_A}(\mathcal{A}_{m_L}(s_L)) \le \mathcal{A}_{\zeta_A}(s_X)$ (using functoriality, Lemma L.9(1.) and monotonicity). Then either $(\mathcal{A}_{\zeta_A}(s_X), \mathcal{A}_{\zeta_A}(s_X)) \in M$ or it is subsumed by another, maximal, pair $(a'_1, a'_2) \in M$. In both cases this is the desired pair of annotations. □

Proposition 10.14 (Soundness for $\rightsquigarrow$). *The relation $\rightsquigarrow$ is sound, i.e. if $X \in \mathcal{L}(A[a_1, a_2])$ (witnessed via a legal arrow $\psi\colon X[s_X, s_X] \to A[a_1, a_2]$) and $X \overset{p,m_L}{\Longrightarrow} Y$, then there exists an abstract rewriting step $A[a_1, a_2] \overset{p,\psi\circ m_L}{\rightsquigarrow} B[b_1, b_2]$ such that $Y \in \mathcal{L}(B[b_1, b_2])$.*

Proof. Since $X \overset{p,m_L}{\Longrightarrow} Y$ we have that $(L \overset{m_L}{\rightarrowtail} X) \overset{p}{\Rightarrow} (R \overset{m_R}{\rightarrowtail} Y)$ for some co-match m_R. We set $\varphi = \psi \circ m_L$ and Corollary 9.30 implies that $(R \overset{m_R}{\rightarrowtail} Y) \in \mathcal{L}(R \overset{n_R}{\rightarrowtail} B)$ where $(L \overset{n_L}{\rightarrowtail} \langle\!\langle \varphi, \varphi_L \rangle\!\rangle) \overset{p}{\Rightarrow} (R \overset{n_R}{\rightarrowtail} B)$ and $\langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ is the rewritable materialization with $L \overset{n_L}{\rightarrowtail} \langle\!\langle \varphi, \varphi_L \rangle\!\rangle \overset{\psi'}{\to} A$ (such that $\psi' \circ \zeta_A = \psi$). This situation can be summarized in the diagram from the proof of Proposition 9.28 which is depicted below in a simplified form, but with added annotations.

$$\begin{tikzcd}
L[s_L, s_L] \arrow[d, tail, "m_L"] \arrow[dd, tail, bend right=50, "n_L"'] & I[s_I, s_I] \arrow[l, tail, "\varphi_L"'] \arrow[r, tail, "\varphi_R"] \arrow[d, tail, "m_I"] & R[s_R, s_R] \arrow[d, tail, "m_R"'] \arrow[dd, tail, bend left=50, "n_R"] \\
X[s_X, s_X] \arrow[d, "\zeta_A"] & Z[s_Z, s_Z] \arrow[l, tail, "\varphi_X"'] \arrow[r, tail, "\varphi_Y"] \arrow[d, "\zeta_C"] & Y[s_Y, s_Y] \arrow[d, "\zeta_B"'] \\
\langle\!\langle \varphi, \varphi_L \rangle\!\rangle[a'_1, a'_2] & C[c_1, c_2] \arrow[l, tail, "\varphi_A"'] \arrow[r, tail, "\varphi_B"] & B[b_1, b_2]
\end{tikzcd}$$

Due to Proposition 10.12 there exists a pair of annotations $(a'_1, a'_2) \in M$ and a legal arrow $\zeta_A\colon X[s_X, s_X] \to \langle\!\langle \varphi, \varphi_L \rangle\!\rangle[a'_1, a'_2]$. Furthermore we assume c_1, c_2, b_1, b_2 as in Definition 10.13.

It is left to show that ζ_C and in particular ζ_B are legal morphisms. To show that ζ_C is legal, we observe that, due to functoriality, the homomorphism property and the pushout property for standard annotations, we have:

$$\begin{array}{rl} & \mathcal{A}_{\varphi_A}(\mathcal{A}_{\zeta_C}(s_Z)) + (\mathcal{A}_{n_L}(s_L) - \mathcal{A}_{n_L\circ\varphi_L}(s_I)) \\ = & \mathcal{A}_{\zeta_A}(\mathcal{A}_{\varphi_X}(s_Z)) + (\mathcal{A}_{\zeta_A}(\mathcal{A}_{m_L}(s_L)) - \mathcal{A}_{\zeta_A}(\mathcal{A}_{m_L\circ\varphi_L}(s_I))) \\ = & \mathcal{A}_{\zeta_A}(\mathcal{A}_{\varphi_X}(s_Z) + (\mathcal{A}_{m_L}(s_L) - \mathcal{A}_{m_L\circ\varphi_L}(s_I))) \\ = & \mathcal{A}_{\zeta_A}(s_X) \end{array}$$

Since $a_1' \le \mathcal{A}_{\zeta_A}(s_X) \le a_2'$ we know from Definition 10.13 that there is a (maximal) annotation (c_1, c_2) satisfying the respective inequalities, i.e., $c_1 \le \mathcal{A}_{\zeta_C}(s_Z) \le c_2$, which implies that ζ_C is legal.

Second, to show that ζ_B is legal, we observe that due to the pushout property for standard annotations, the homomorphism property and functoriality:

$$\begin{array}{rcl} \mathcal{A}_{\zeta_B}(s_Y) & = & \mathcal{A}_{\zeta_B}(\mathcal{A}_{\varphi_Y}(s_Z) + (\mathcal{A}_{m_R}(s_R) - \mathcal{A}_{m_R\circ\varphi_R}(s_I))) \\ & = & \mathcal{A}_{\zeta_B}(\mathcal{A}_{\varphi_Y}(s_Z)) + (\mathcal{A}_{\zeta_B}(\mathcal{A}_{m_R}(s_R)) - \mathcal{A}_{\zeta_B}(\mathcal{A}_{m_R\circ\varphi_R}(s_I))) \\ & = & \mathcal{A}_{\varphi_B}(\mathcal{A}_{\zeta_C}(s_Z)) + (\mathcal{A}_{n_R}(s_R) - \mathcal{A}_{n_R\circ\varphi_R}(s_I)) \end{array}$$

Since ζ_C is legal and we have $c_1 \le \mathcal{A}_{\zeta_C}(s_Z) \le c_2$, we obtain from the definition of b_1, b_2 and monotonicity that $b_1 \le \mathcal{A}_{\zeta_B}(s_Y) \le b_2$. □

Proposition 10.17 (Soundness for $\hookrightarrow$). *The relation $\hookrightarrow$ is sound, i.e. if $X \in \mathcal{L}(A[a_1, a_2])$ (witnessed via a legal arrow $\psi\colon X[s_X, s_X] \to A[a_1, a_2]$) and $X \overset{p,m_L}{\Longrightarrow} Y$, then there exists an abstract rewriting step $A[a_1, a_2] \overset{p,\psi\circ m_L}{\hookrightarrow} B[b_1, b_2]$ such that $Y \in \mathcal{L}(B[b_1, b_2])$.*

Proof. We modify the proof of Proposition 10.12, on which Proposition 10.14 relies. We have to show that there always exists a pair of annotations $(a_1', a_2') \in M$ for which we have a legal arrow $\zeta_A\colon X[s_X, s_X] \to \langle\!\langle \varphi, \varphi_L \rangle\!\rangle[a_1', a_2']$. (The rest of the proof of Proposition 10.14 proceeds as before.)

As in Proposition 10.12 we show that $(\mathcal{A}_{\zeta_A}(s_X), \mathcal{A}_{\zeta_A}(s_X))$ is an annotation (a_1', a_2') which satisfies $a_1 \le \mathcal{A}_\psi(a_1')$ and $\mathcal{A}_\psi(a_2') \le a_2$. Since the square consisting of id_L, m_L, ζ_A, n_L is a pushout, we use the Beck-Chevally property and the adjunction property to prove $red_{n_L}(\mathcal{A}_{\zeta_A}(s_X)) = \mathcal{A}_{id_L}(red_{m_L}(s_X)) = red_{m_L}(s_X) = s_L$ holds. Hence either $(\mathcal{A}_{\zeta_A}(s_X), \mathcal{A}_{\zeta_A}(s_X))$ or an annotation subsuming it is contained in the set M of Definition 10.15. □

Proposition 10.18 (Completeness for $\hookrightarrow$). *The relation $\hookrightarrow$ is complete, i.e. if $A[a_1, a_2] \overset{p,\varphi}{\hookrightarrow} B[b_1, b_2]$ and $Y \in \mathcal{L}(B[b_1, b_2])$, then there exists $X \in \mathcal{L}(A[a_1, a_2])$ (witnessed via a legal arrow $\psi\colon X[s_X, s_X] \to A[a_1, a_2]$) such that $X \overset{p,m_L}{\Longrightarrow} Y$ and $\varphi = \psi \circ m_L$.*

Proof. Since there is a rewriting step from $A[a_1, a_2]$ to $B[b_1, b_2]$ we obtain $\langle\!\langle \varphi, \varphi_L \rangle\!\rangle$ as the materialization (with $L \overset{n_L}{\rightarrowtail} \langle\!\langle \varphi, \varphi_L \rangle\!\rangle \overset{\psi'}{\to} A$ where $\varphi = \psi' \circ n_L$) and the following two pushouts below.

$$\begin{tikzcd}
L[s_L, s_L] \arrow[d, tail, "n_L"'] & I[s_I, s_I] \arrow[l, tail, "\varphi_L"'] \arrow[r, tail, "\varphi_R"] \arrow[d, tail, "n_I"] & R[s_R, s_R] \arrow[d, tail, "n_R"] & Y[s_Y, s_Y] \arrow[dl, tail, "\zeta_B"] \\
\langle\!\langle \varphi, \varphi_L \rangle\!\rangle [a_1', a_2'] & C[c_1, c_2] \arrow[l, tail, "\varphi_A"'] \arrow[r, tail, "\varphi_B"] & B[b_1, b_2] &
\end{tikzcd}$$

Furthermore $(a'_1, a'_2) \in M$ and

$$a'_1 \leq \mathcal{A}_{\varphi_A}(c_1) + (\mathcal{A}_{n_L}(s_L) - \mathcal{A}_{n_L \circ \varphi_L}(s_I)) \quad \mathcal{A}_{\varphi_A}(c_2) + (\mathcal{A}_{n_L}(s_L) - \mathcal{A}_{n_L \circ \varphi_L}(s_I)) \leq a'_2$$

$$b_i = \mathcal{A}_{\varphi_B}(c_i) + (\mathcal{A}_{n_R}(s_R) - \mathcal{A}_{n_R \circ \varphi_R}(s_I)) \qquad \text{for } i \in \{1,2\}$$

The arrow ζ_B is legal and witnesses $Y \in \mathcal{L}(B[b_1, b_2])$, i.e. $b_1 \leq \mathcal{A}_{\zeta_B}(s_Y) \leq b_2$.

- We first observe that there is a unique maximal pair (c_1, c_2) satisfying the above inequalities, in particular $c_i = \mathit{red}_{\varphi_A}(a'_i)$. We have

  $$\begin{array}{ll} a'_i & \\ = & [\text{PO property}] \\ \mathcal{A}_{\varphi_A}(\mathit{red}_{\varphi_A}(a'_i)) + (\mathcal{A}_{n_L}(\mathit{red}_{n_L}(a'_i)) - \mathcal{A}_{n_L \circ \varphi_L}(\mathit{red}_{n_L \circ \varphi_L}(a'_i)) & \\ = & [\mathit{red}_{n_L}(a'_i) = s_L, \text{ Definition of } M \text{ (from Definition 10.15)}] \\ \mathcal{A}_{\varphi_A}(\mathit{red}_{\varphi_A}(a'_i)) + (\mathcal{A}_{n_L}(s_L) - \mathcal{A}_{n_L \circ \varphi_L}(s_I)) & \end{array}$$

  Furthermore let c_1 be an annotation satisfying above inequality. Then:

  $$\begin{array}{ll} \mathit{red}_{\varphi_A}(a'_1) & \\ \leq & [\text{Mon.}] \\ \mathit{red}_{\varphi_A}(\mathcal{A}_{\varphi_A}(c_1) + (\mathcal{A}_{n_L}(s_L) - \mathcal{A}_{n_L \circ \varphi_L}(s_I))) & \\ = & [\text{Adj. prop.}] \\ \mathit{red}_{\varphi_A}(\mathcal{A}_{\varphi_A}(c_1)) + (\mathit{red}_{\varphi_A}(\mathcal{A}_{n_L}(s_L)) - \mathit{red}_{\varphi_A}(\mathcal{A}_{n_L \circ \varphi_L}(s_I))) & \\ = & [\text{Lemma L.9(2.)}] \\ c_1 + (\mathit{red}_{\varphi_A}(\mathcal{A}_{n_L}(s_L)) - \mathit{red}_{\varphi_A}(\mathcal{A}_{n_L \circ \varphi_L}(s_I))) & \\ = & [\text{Funct.}] \\ c_1 + (\mathit{red}_{\varphi_A}(\mathcal{A}_{n_L}(s_L)) - \mathit{red}_{\varphi_A}(\mathcal{A}_{\varphi_A \circ n_I}(s_I))) & \\ = & [\text{Lemma L.9(2.)}] \\ c_1 + (\mathit{red}_{\varphi_A}(\mathcal{A}_{n_L}(s_L)) - \mathcal{A}_{n_I}(s_I)) & \\ = & [\text{Beck-Chevalley}] \\ c_1 + (\mathcal{A}_{n_I}(\mathit{red}_{\varphi_L}(s_L)) - \mathcal{A}_{n_I}(s_I)) & \\ = & [\text{Adj. prop.}] \\ c_1 + (\mathcal{A}_{n_I}(s_I) - \mathcal{A}_{n_I}(s_I)) & \\ = & [\text{Subtr. well-behaved}] \\ c_1 & \end{array}$$

  And similarly $\mathit{red}_{\varphi_A}(a'_2) \geq c_2$ for an annotation c_2 satisfying above equality.

- We will next show that there exists a mono $m_R \colon R \rightarrowtail Y$ with $(R \overset{m_R}{\rightarrowtail} Y) \in \mathcal{L}(R \overset{n_R}{\rightarrowtail} B)$. We do this by taking the pullback of the arrows n_R, ζ_B, obtaining the diagram shown to the right.

  $$\begin{tikzcd} R'[s'_R, s'_R] \arrow[r, "\iota"] \arrow[d, tail, "m_R"'] & R[s_R, s_R] \arrow[d, tail, "n_R"] \\ Y[s_Y, s_Y] \arrow[r, "\zeta_B"] & B[b_1, b_2] \end{tikzcd}$$

  According to the Beck-Chevalley property we have

  $$\mathcal{A}_\iota(s_{R'}) = \mathcal{A}_\iota(\mathit{red}_{m_R}(s_Y)) = \mathit{red}_{n_R}(\mathcal{A}_{\zeta_B}(s_Y)).$$

We know that $b_1 \le \mathcal{A}_{\zeta_B}(s_Y) \le b_2$ since ζ_B is legal and it follows with monotonicity of red_{n_R} that

$$\mathit{red}_{n_R}(b_1) \le \mathcal{A}_\iota(s_{R'}) \le \mathit{red}_{n_R}(b_2).$$

Next, we show that $\mathit{red}_{n_R}(b_1) = \mathit{red}_{n_R}(b_2) = s_R$:

$$
\begin{aligned}
& \mathit{red}_{n_R}(b_i) \\
= \quad & [\text{Definition}] \\
& \mathit{red}_{n_R}(\mathcal{A}_{\varphi_B}(c_i) + (\mathcal{A}_{n_R}(s_R) - \mathcal{A}_{n_R \circ \varphi_R}(s_I))) \\
= \quad & [\text{Adj. prop.}] \\
& \mathit{red}_{n_R}(\mathcal{A}_{\varphi_B}(c_i)) + (\mathit{red}_{n_R}(\mathcal{A}_{n_R}(s_R)) - \mathit{red}_{n_R}(\mathcal{A}_{n_R \circ \varphi_R}(s_I))) \\
= \quad & [\text{Lemma L.9(2.)}] \\
& \mathit{red}_{n_R}(\mathcal{A}_{\varphi_B}(c_i)) + (s_R - \mathcal{A}_{\varphi_R}(s_I)) \\
= \quad & [\text{Beck-Chevalley}] \\
& \mathcal{A}_{\varphi_R}(\mathit{red}_{n_I}(c_i)) + (s_R - \mathcal{A}_{\varphi_R}(s_I)) \\
= \quad & [\text{Adj. prop.}] \\
& \mathcal{A}_{\varphi_R}(\mathit{red}_{n_I}(\mathit{red}_{\varphi_A}(a'_i))) + (s_R - \mathcal{A}_{\varphi_R}(s_I)) \\
= \quad & [\text{Lemma L.9(4.)}] \\
& \mathcal{A}_{\varphi_R}(\mathit{red}_{\varphi_L}(\mathit{red}_{n_L}(a'_i))) + (s_R - \mathcal{A}_{\varphi_R}(s_I)) \\
= \quad & [\mathit{red}_{n_L}(a'_i) = s_L, \text{ Definition of } M] \\
& \mathcal{A}_{\varphi_R}(\mathit{red}_{\varphi_L}(s_L)) + (s_R - \mathcal{A}_{\varphi_R}(s_I)) \\
= \quad & [\text{Adj. prop.}] \\
& \mathcal{A}_{\varphi_R}(s_I) + (s_R - \mathcal{A}_{\varphi_R}(s_I)) \\
= \quad & [\text{Subtr. well-behaved}] \\
& s_R
\end{aligned}
$$

The last equality holds since $\mathit{red}_{\varphi_R}(s_R) = s_I$ and hence due to the adjunction property $\mathcal{A}_{\varphi_R}(s_I) = \mathcal{A}_{\varphi_R}(\mathit{red}_{\varphi_R}(s_R)) \le s_R$.

This means that ι is a legal arrow and we can infer from the isomorphism property that it is an iso, without loss of generality we can assume that it is the identity. Hence $(R \overset{m_R}{\rightarrowtail} Y) \in \mathcal{L}(R \overset{n_R}{\rightarrowtail} B)$.

- Since $(L \overset{n_L}{\rightarrowtail} \langle\!\langle \varphi, \varphi_L \rangle\!\rangle) \overset{p}{\Rightarrow} (R \overset{n_R}{\rightarrowtail} B)$ we can infer from Corollary 9.30 that there exists a match $m_L \colon L \rightarrowtail X$ where $(L \overset{m_L}{\rightarrowtail} X) \in \mathcal{L}(L \overset{n_L}{\rightarrowtail} \langle\!\langle \varphi, \varphi_L \rangle\!\rangle)$ and $(L \overset{m_L}{\rightarrowtail} X) \overset{p}{\Rightarrow} (R \overset{m_R}{\rightarrowtail} Y)$. This situation can be summarized in the diagram from the proof of Proposition 9.28 which is depicted below with added annotations.

$$\begin{tikzcd}
L[s_L, s_L] \arrow[d, tail, "m_L"] \arrow[dd, tail, bend right, "n_L"'] & I[s_I, s_I] \arrow[l, tail, "\varphi_L"'] \arrow[r, tail, "\varphi_R"] \arrow[d, tail, "m_I"] & R[s_R, s_R] \arrow[d, tail, "m_R"'] \arrow[dd, tail, bend left, "n_R"] \\
X[s_X, s_X] \arrow[d, "\zeta_A"] & Z[s_Z, s_Z] \arrow[l, tail, "\varphi_X"'] \arrow[r, tail, "\varphi_Y"] \arrow[d, "\zeta_C"] & Y[s_Y, s_Y] \arrow[d, "\zeta_B"'] \\
\langle\!\langle \varphi, \varphi_L \rangle\!\rangle[a'_1, a'_2] & C[c_1, c_2] \arrow[l, tail, "\varphi_A"'] \arrow[r, tail, "\varphi_B"] & B[b_1, b_2]
\end{tikzcd}$$

It is left to show that ζ_C and in particular ζ_A are legal.

- For ζ_C we show that, due to the adjunction property, the Beck-Chevally property and monotonicity:

  $$\mathcal{A}_{\zeta_C}(s_C) = \mathcal{A}_{\zeta}(\mathit{red}_{\varphi_Y}(s_Y)) = \mathit{red}_{\varphi_B}(\mathcal{A}_{\zeta_B}(s_Y)) \geq \mathit{red}_{\varphi_B}(b_1)$$

  and similarly $\mathcal{A}_{\zeta}(s_C) = \mathit{red}_{\varphi_B}(\mathcal{A}_{\zeta_A}(s_Y)) \leq \mathit{red}_{\varphi_B}(b_2)$.

  Therefore, $\mathit{red}_{\varphi_B}(b_1) \leq \mathcal{A}_{\zeta}(s_C) \leq \mathit{red}_{\varphi_B}(b_2)$ holds and it is only left to show that $\mathit{red}_{\varphi_B}(b_i) = c_i$ for $i \in \{1,2\}$. In particular, we have to show that $\mathit{red}_{\varphi_B}(\mathcal{A}_{\varphi_B}(c_i) + (\mathcal{A}_{n_R}(s_R) - \mathcal{A}_{n_L \circ \varphi_R}(s_I)) = c_i$ and this is analogous to the proof concerning the left-hand square above.

- Now, we show that ζ_A is legal:

  $$\begin{aligned}
  &\mathcal{A}_{\zeta_A}(s_X)\\
  &= \qquad \text{[PO prop. for std. ann.]}\\
  &\mathcal{A}_{\zeta_A}(\mathcal{A}_{\varphi_X}(s_Z) + (\mathcal{A}_{m_L}(s_L) - \mathcal{A}_{m_L \circ \varphi_L}(s_I)))\\
  &= \qquad \text{[Homom. prop.]}\\
  &\mathcal{A}_{\zeta_A}(\mathcal{A}_{\varphi_X}(s_Z)) + (\mathcal{A}_{\zeta_A}(\mathcal{A}_{m_L}(s_L)) - \mathcal{A}_{\zeta_A}(\mathcal{A}_{m_L \circ \varphi_L}(s_I)))\\
  &= \qquad \text{[Funct.]}\\
  &\mathcal{A}_{\varphi_A}(\mathcal{A}_{\zeta_C}(s_Z)) + (\mathcal{A}_{n_L}(s_L) - \mathcal{A}_{n_L \circ \varphi_L}(s_I))\\
  &\geq \qquad \text{[Mon.]}\\
  &\mathcal{A}_{\varphi_A}(c_1) + (\mathcal{A}_{n_L}(s_L) - \mathcal{A}_{n_L \circ \varphi_L}(s_I))\\
  &\geq \qquad \text{[Definition of } c_1\text{]}\\
  &a_1'
  \end{aligned}$$

  Similarly $\mathcal{A}_{\zeta_A}(s_X) \leq a_2'$.

Hence we have found $m_L \colon L \rightarrowtail X$ such that $X \in \mathcal{L}(\langle\!\langle \varphi, \varphi_L \rangle\!\rangle[a_1', a_2'])$ (witnessed by ζ_A) and $X \overset{p,m_L}{\Longrightarrow} Y$. Since, due to the materialization $\psi' \colon \langle\!\langle \varphi, \varphi_L \rangle\!\rangle[a_1', a_2'] \to A[a_1, a_2]$ is a legal arrow, we have that $X \in \mathcal{L}(A[a_1, a_2])$, witnessed by $\psi := \psi' \circ \zeta_A$ and it holds that $\psi \circ m_L = \psi' \circ \zeta_A \circ m_L = \psi' \circ n_L = \varphi$. □

Corollary 10.19 (Strongest post-condition). *Let $A[a_1, a_2]$ be an annotated object and let $\varphi \colon L \to A$. We obtain (several) abstract rewriting steps $A[a_1, a_2] \overset{p,\varphi}{\hookrightarrow} B[b_1, b_2]$, where we always obtain the same object B. Now let $N = \{(b_1, b_2) \mid A[a_1, a_2] \overset{p,\varphi}{\hookrightarrow} B[b_1, b_2]\}$. Then the* strongest post-condition *is the language of the multiply annotated object $B[N]$, i.e.*

$$\begin{aligned}
\mathcal{L}(B[N]) \quad = \quad & \{Y \mid \exists (X \in \mathcal{L}(A[a_1, a_2]), \mathit{witnessed\ by\ } \psi),\\
& \quad (L \overset{m_L}{\rightarrowtail} X) \colon (\varphi = \psi \circ m_L \wedge X \overset{p,m_L}{\Longrightarrow} Y)\}
\end{aligned}$$

Proof. Straightforward from Propositions 10.17 and 10.18. □

A.7. Proofs of Chapter 11

Proposition 11.2 (Constructed materialization is terminal). *Let $L \overset{\varphi}{\to} A$ be a fixed graph morphism in $\mathbf{Graph}_\Lambda$. Then the factorization $L \overset{\alpha}{\rightarrowtail} \tilde{A} \overset{\psi}{\to} A$ from Definition 11.1 is the terminal object in the category $\mathbf{Mat}_\varphi$.*

Proof. Given the factorization $L \overset{\alpha}{\rightarrowtail} \tilde{A} \overset{\psi}{\to} A$ of $L \overset{\varphi}{\to} A$ from Definition 11.1 with $\varphi = \psi \circ \alpha$. The morphism $\alpha\colon L \to \tilde{A}$ is the embedding morphism from L into $\tilde{A}$ and by the construction of $\tilde{A}$ there exists a second embedding morphism $\gamma\colon A \to \tilde{A}$ with $\mathrm{img}(\alpha) \cap \mathrm{img}(\gamma) = \emptyset$ and

$$\gamma(x) = \begin{cases} x & \text{if } x \in V_A \\ (x, \mathit{src}_A(x), \mathit{tgt}_A(x), \mathit{lab}_A(x)) & \text{if } x \in E_A \end{cases}$$

It is easy to see that γ is well-defined.

Let $L \overset{\beta}{\rightarrowtail} G \overset{g}{\to} A$ be another factorization of $L \overset{\varphi}{\to} A$ with $\varphi = g \circ \beta$. If the object $L \overset{\alpha}{\rightarrowtail} \tilde{A} \overset{\psi}{\to} A$ is the terminal object in the materialization category, there must exist a unique graph morphism $f\colon G \to \tilde{A}$ such that the diagram to the right commutes and the square is a pullback.

$$\begin{tikzcd}
L \arrow[r, tail, "\beta"] \arrow[d, tail, "id"'] \arrow[rr, bend left, "\varphi"] \arrow[dr, phantom, "(PB)"] & G \arrow[r, "g"] \arrow[d, dashed, "f"] & A \arrow[dl, hook', bend left, "\gamma"] \\
L \arrow[r, tail, "\alpha"] & \tilde{A} \arrow[ur, "\psi"'] &
\end{tikzcd}$$

Define $f = (f_V, f_E)$ in the following way:

$$f_V\colon\ V_G \to V_{\tilde{A}} \quad f_V(x) = \begin{cases} \alpha_V \circ \beta_V^{-1}(x) & \text{if } x \in \mathrm{img}(\beta_V) \\ \gamma_V \circ g_V(x) & \text{otherwise} \end{cases}$$

$$f_E\colon\ E_G \to E_{\tilde{A}} \quad f_E(x) = \begin{cases} \alpha_E \circ \beta_E^{-1}(x) & \text{if } x \in \mathrm{img}(\beta_E) \\ (g_E(x), f_V(\mathit{src}_G(x)), f_V(\mathit{tgt}_G(x)), \mathit{lab}_G(x)) & \text{otherwise} \end{cases}$$

Note that since β is an injection, the element $\beta^{-1}(x)$ is unique whenever x is in the image of β.

We will next prove that f preserves the structure of G, i.e., that it is a well-defined graph morphism. We need to prove that the following three properties hold for every edge $x \in E_G$:

$$f_V(\mathit{src}_G(x)) = \mathit{src}_{\tilde{A}}(f_E(x)) \tag{A.16}$$

$$f_V(\mathit{tgt}_G(x)) = \mathit{tgt}_{\tilde{A}}(f_E(x)) \tag{A.17}$$

$$\mathit{lab}_G(x) = \mathit{lab}_{\tilde{A}}(f_E(x)) \tag{A.18}$$

There are the following two cases:

Case 1: Suppose $x \in \mathrm{img}(\beta)$. Then there exists $y \in L$ such that $x = \beta_E(y)$. In this case we obtain

$$\begin{aligned} f_V(\mathit{src}_G(x)) &= \alpha_V(\beta_V^{-1}(\mathit{src}_G(x))) = \alpha_V(\beta_V^{-1}(\mathit{src}_G(\beta_E(y)))) \\ &= \alpha_V(\beta_V^{-1}(\beta_V(\mathit{src}_L(y)))) = \alpha_V(\mathit{src}_L(y)) = \mathit{src}_{\tilde{A}}(\alpha_E(y)) \\ &= \mathit{src}_{\tilde{A}}(\alpha_E(\beta_E^{-1}(x))) = \mathit{src}_{\tilde{A}}(f_E(x)) \end{aligned}$$

$$\begin{aligned}
lab_G(x) &= lab_G(\beta_E(y)) = lab_L(y) = lab_{\tilde{A}}(\alpha_E(y)) \\
&= lab_{\tilde{A}}(\alpha_E(\beta_E^{-1}(\beta_E(y)))) = lab_{\tilde{A}}(\alpha_E(\beta_E^{-1}(x))) \\
&= lab_{\tilde{A}}(f_E(x))
\end{aligned}$$

The case of the target function (*tgt*) is equivalent to the source function (*src*).

Case 2: Whenever $x \notin \operatorname{img}(\beta)$, we get that

$$f_E(x) = \big(g_E(x), f_V(src_G(x)), f_V(tgt_G(x)), lab_G(x)\big).$$

Since $x \notin \operatorname{img}(\beta)$ we obtain the following equations:

$$\begin{aligned}
src_{\tilde{A}}(f_E(x)) &= src_{\tilde{A}}\big((g_E(x), f_V(src_G(x)), f_V(tgt_G(x)), lab_G(x))\big) \\
&= f_V(src_G(x)) \\
lab_{\tilde{A}}(f_E(x)) &= lab_{\tilde{A}}\big((g_E(x), f_V(src_G(x)), f_V(tgt_G(x)), lab_G(x))\big) \\
&= lab_G(x)
\end{aligned}$$

Again, the case of the target function is equivalent to the case of the source function.

Therefore $f\colon G \to \tilde{A}$ is a graph morphism.

We now prove that the following three properties hold for f:

$$\psi \circ f = g \tag{A.19}$$

$$f \circ \beta = \alpha \tag{A.20}$$

$$\forall x \in G,\ x \notin \operatorname{img}(\beta) \implies f(x) \notin \operatorname{img}(\alpha) \tag{A.21}$$

Properties (A.20) and (A.21) together ensure that every element of img(α) has a unique preimage under f, which – together with the commutativity of the square – guarantees that it is a pullback.

Proof of (A.19): Assume $x \in \operatorname{img}(\beta)$. Since $\psi \circ \alpha = \varphi = g \circ \beta$ we get:

$$(\psi \circ f)(x) = \psi(f(x)) = \psi(\alpha(\beta^{-1}(x))) = \varphi(\beta^{-1}(x)) = g(\beta(\beta^{-1}(x))) = g(x)$$

Assume $x \notin \operatorname{img}(\beta)$. Then x is either a node or an edge of G.
First we assume that $x \in V_G$ and $x \notin \operatorname{img}(\beta_V)$. Since $\psi_V \circ \gamma_V = id_V$ we get:

$$(\psi_V \circ f_V)(x) = \psi_V(f_V(x)) = \psi_V(\gamma_V(g_V(x))) = id_V(g_V(x)) = g_V(x)$$

Now assume $x \in E_G$ and $x \notin \operatorname{img}(\beta_E)$:

$$(\psi_E \circ f_E)(x) = \psi_E\big((g_E(x), f_V(src_G(x)), f_V(tgt_G(x)), lab_G(x))\big) = g_E(x)$$

Proof of (A.20): Since β is a mono, we get that for all $x \in L$, there exists a unique $y \in \operatorname{img}(\beta)$ such that $\beta(x) = y$ and $\beta^{-1}(y) = x$. By the construction of f, the following equation holds:

$$(f \circ \beta)(x) = f(\beta(x)) = f(y) = (\alpha \circ \beta^{-1})(y) = \alpha(\beta^{-1}(y)) = \alpha(x)$$

Proof of (A.21): Let $x \in G$ be given and $x \notin \operatorname{img}(\beta)$. Then x is either a node or an edge of G. First we assume that $x \in V_G$. Then $f_V(x) = \gamma_V \circ g_V(x)$. By the

construction of $\tilde{A}$ it follows that $\operatorname{img}(\alpha) \cap \operatorname{img}(\gamma) = \emptyset$ and therefore we get that $f_V(x) \notin \operatorname{img}(\alpha)$.
Now assume $x \in E_G$ and $f_E(x) = \big(g_E(x), f_V(src_G(x)), f_V(tgt_G(x)), lab_G(x)\big)$. By the construction of $\tilde{A}$ we have that all edges of $E_{\tilde{A}}$ are either of the form (e,s,t,l), with $(e,s,t,l) \notin \operatorname{img}(\alpha)$ or an edge from E_L and therefore in $\operatorname{img}(\alpha)$. We get that $f_E(x) \notin \operatorname{img}(\alpha)$.

To prove that f is unique, we show that any other morphism $f'\colon G \to \tilde{A}$, satisfying the properties (A.19), (A.20) or (A.21), equals f. We show equality by checking that $f(x) = f'(x)$ for all $x \in G$.

Case 1: Suppose $x \in \operatorname{img}(\beta)$. Then there exists an element $y \in L$ such that $\beta(y) = x$ and we obtain:

$$f'(x) = f'(\beta(y)) \overset{(A.20)}{=} \alpha(y) = \alpha(\beta^{-1}(x)) = f(x)$$

Case 2: Suppose $x \notin \operatorname{img}(\beta)$ and x is a node of G (e.g. $x \notin \operatorname{img}(\beta_V)$). If $f'_V(x) \in V_L = \operatorname{img}(\alpha_V)$, we would get that $x \in \operatorname{img}(\beta_V)$, due to property (A.21), which is a contradiction. We can hence conclude that $f'_V(x) \in V_A$, which implies $\gamma_V(\psi_V(f'_V(x))) = f'_V(x)$, and furthermore:

$$f'_V(x) = \gamma_V(\psi_V(f'_V(x))) \overset{(A.19)}{=} \gamma_V(g_V(x)) = f_V(x)$$

Case 3: Suppose $x \notin \operatorname{img}(\beta)$ and x is an edge of G (e.g. $x \notin \operatorname{img}(\beta_E)$). If $f'_E(x) \in E_L = \operatorname{img}(\alpha_E)$, we would get that $x \in \operatorname{img}(\beta_E)$, due to property (A.21), which is a contradiction. We can hence conclude that $f'_E(x) \in E_A$, which implies that $f_E(x)$ must be of the form $(e,s,t,l) \in E_{\tilde{A}}$. We will now show that

$$(e,s,t,l) = (g_E(x), f_V(src_G(x)), f_V(tgt_G(x)), lab_G(x)),$$

which implies $f'_E(x) = f_E(x)$.

$$\begin{aligned}
g_E(x) &\overset{(A.19)}{=} \psi_E(f'_E(x)) = \psi_E(e,s,t,l) = e \\
f_V(src_G(x)) &= src_{\tilde{A}}(f_E(x)) = src_{\tilde{A}}((e,s,t,l)) = s \\
f_V(tgt_G(x)) &= tgt_{\tilde{A}}(f_E(x)) = tgt_{\tilde{A}}((e,s,t,l)) = t \\
lab_G(x) &= lab_{\tilde{A}}(f_E(x)) = lab_{\tilde{A}}((e,s,t,l)) = l
\end{aligned}$$

Hence the graph morphism $f\colon G \to \tilde{A}$ exists and it is unique for all factorizations $L \overset{\beta}{\rightarrowtail} G \overset{g}{\to} A$ of $L \overset{\varphi}{\to} A$ with $\varphi = g \circ \beta$. Therefore the constructed object $L \overset{\alpha}{\rightarrowtail} \tilde{A} \overset{\psi}{\to} A$ is the terminal object in the materialization category, i.e. $\tilde{A} = \langle \varphi \rangle$. □

B

Termination Analysis Experiments

The Termination Problems Database (short: TPDB) consists of a folder named *TRS Standard*, which contains a total of 1498 term rewriting systems. In our framework we are interested in non-collapsing left-linear term rewriting systems. Only 621 of these systems are both left-linear and non-collapsing. We discarded 386 of the term rewriting systems that exceeded tractability for the resulting graph transformation systems. Another 34 were left out since they were obviously non-terminating. Of the 201 remaining term rewriting systems 95 are right-linear.

Experimental Results (Overview)

We ran Grez on 201 examples (including the examples of this paper) using a Windows workstation with a $2,67$ Ghz, 4-core CPU and 8 GB RAM. We used the weighted type graph technique over ordered semirings (see Chapter 5) and tried to find weighted type graphs which consist of 2 nodes. For all graph transformation systems, where Grez could find a termination proof, the weighted type graphs were generated within a few seconds. Some term rewriting systems satisfy SN but not SN^b due to cycles. Therefore, using the type graph technique, it is impossible to prove termination for these examples. To summarize the results, we present the following table:

Termination Analysis using Grez				
TPDB (Standard)	1498			
Left-linear + Non-Collapsing	621			
Too Many Rules (> 9)	-235			
Generated Graphs Too Large	-151			
Non-Terminating	-34			
Tested Total	**201**			
No Result Found	84			
Terminating Total	**117**		**24**	
Terminating using	117	115	24	24
	Number Encoding	Function Encoding	Number Encoding	Function Encoding
Version	Basic		Extended	

B.1. Termination Proofs of Chapter 6

Version	Basic		Extend		Version	Basic		Extend	
Encoding	N	F	N	F	Encoding	N	F	N	F
TPDB Term Rewriting System					TPDB Term Rewriting System				
AG01/3.15	✓	✗	✗	✗	EEGIJCAR12/emmes-nloop-ex11	✓	✓	✗	✗
AG01/3.23	✓	✓	✗	✗	EEGIJCAR12/emmes-nloop-ex22	✗	✗	✗	✗
AG01/3.24	✓	✓	✗	✗	EEGIJCAR12/enger-nloop-expayet	✗	✗	✗	✗
AG01/3.26	✗	✗	✗	✗	EEGIJCAR12/enger-nloop-isDNat	✓	✓	✗	✗
AG01/3.31	✗	✗	✗	✗	EEGIJCAR12/enger-nloop-isList-List	✓	✓	✗	✗
AG01/3.33	✓	✓	✗	✗	EEGIJCAR12/enger-nloop-isList	✓	✓	✗	✗
AG01/3.35	✗	✗	✗	✗	EEGIJCAR12/enger-nloop-isTrueList	✓	✓	✗	✗
AG01/3.37	✓	✓	✗	✗	EEGIJCAR12/enger-nloop-swapX	✓	✓	✗	✗
AG01/3.38	✓	✓	✗	✗	EEGIJCAR12/enger-nloop-swapXY	✓	✓	✗	✗
AG01/3.42	✗	✗	✗	✗	EEGIJCAR12/enger-nloop-swapXY2	✓	✓	✗	✗
AG01/3.47	✓	✓	✗	✗	EEGIJCAR12/enger-nloop-swapdecr	✓	✓	✗	✗
AG01/3.49	✓	✓	✗	✗	EEGIJCAR12/enger-nloop-toOne	✓	✓	✗	✗
AG01/3.52	✗	✗	✗	✗	EEGIJCAR12/enger-nloop-unbound	✓	✓	✗	✗
AG01/3.7	✓	✓	✗	✗	EEGIJCAR12/enger-nloop-while-lt	✓	✓	✗	✗
AotoYamada05/019	✓	✓	✗	✗	GTSSK07/cade01	✓	✓	✗	✗
AotoYamada05/025	✓	✓	✗	✗	GTSSK07/cade03	✓	✓	✗	✗
Applicative05/Ex261Composition	✓	✓	✗	✗	GTSSK07/cade05t	✓	✓	✗	✗
AProVE04/forwardinst	✓	✓	✗	✗	GTSSK07/cade07	✓	✓	✗	✗
AProVE04/forwardinst2	✓	✓	✗	✗	GTSSK07/cade09	✗	✗	✗	✗
AProVE04/IJCAR1	✓	✓	✗	✗	GTSSK07/cade10	✗	✗	✗	✗
AProVE04/Liveness6.1	✗	✗	✗	✗	GTSSK07/cade11	✓	✓	✗	✗
AProVE04/Liveness6.2	✗	✗	✗	✗	HirokawaMiddeldorp04/n002	✗	✗	✗	✗
AProVE04/rta2	✓	✓	✗	✗	HirokawaMiddeldorp04/n004	✗	✗	✗	✗
AProVE04/rta3	✓	✓	✗	✗	HirokawaMiddeldorp04/n005	✗	✗	✗	✗
AProVE07/otto03	✓	✓	✗	✗	HirokawaMiddeldorp04/n006	✗	✗	✗	✗
AProVE07/otto07	✓	✓	✗	✗	HirokawaMiddeldorp04/n008	✗	✗	✗	✗
AProVE07/thiemann27	✓	✓	✗	✗	HirokawaMiddeldorp04/t007	✓	✓	✓	✓
AProVE10/andIsNat	✓	✓	✗	✗	HirokawaMiddeldorp04/t008	✗	✗	✗	✗
AProVE10/challengefab	✓	✓	✗	✗	HirokawaMiddeldorp04/t010	✓	✓	✓	✓
AProVE10/double	✓	✓	✗	✗	HirokawaMiddeldorp04/t011	✗	✗	✗	✗
AProVE10/downfrom	✓	✓	✗	✗	MixedTRS/5	✓	✓	✗	✗
AProVE10/ex1	✓	✓	✗	✗	MixedTRS/motivation	✓	✓	✗	✗
AProVE10/ex2	✓	✓	✗	✗	MixedTRS/test1	✗	✗	✗	✗
AProVE10/ex3	✓	✓	✗	✗	MixedTRS/while	✓	✓	✗	✗
AProVE10/ex5	✓	✓	✗	✗	Rubio04/aoto	✓	✓	✗	✗
AProVE10/halfdouble	✗	✗	✗	✗	Rubio04/bn122	✗	✗	✗	✗
AProVE10/isList	✓	✓	✗	✗	Rubio04/division	✓	✓	✗	✗
AProVE10/isNat	✓	✓	✗	✗	Rubio04/lescanne	✗	✗	✗	✗
AProVE10/scnp	✓	✓	✗	✗	Rubio04/lindau	✗	✗	✗	✗
AProVE10/Zan06-03-mod	✗	✗	✗	✗	Rubio04/mfp90b	✓	✓	✓	✓
Beerendonk07/4	✓	✓	✗	✗	Rubio04/mfp95	✓	✓	✓	✓
Der95/03	✓	✓	✗	✗	Rubio04/nestrec	✗	✗	✗	✗
Der95/04	✓	✓	✗	✗	Rubio04/p266	✓	✓	✗	✗
Der95/06	✗	✗	✗	✗	Rubio04/prov	✓	✓	✗	✗
Der95/07	✗	✗	✗	✗	Rubio04/revlist	✓	✓	✗	✗
Der95/08	✗	✗	✗	✗	Rubio04/test4	✓	✓	✗	✗
Der95/09	✗	✗	✗	✗	Rubio04/test829	✓	✓	✗	✗
Der95/13	✗	✗	✗	✗	Secret05TRS/cime4	✓	✓	✗	✗
Der95/18	✗	✗	✗	✗	Secret05TRS/matchbox1	✓	✓	✗	✗
Der95/27	✗	✗	✗	✗	Secret06TRS/4	✗	✗	✗	✗

Version	Basic		Extend		Version	Basic		Extend	
Encoding	N	F	N	F	Encoding	N	F	N	F
TPDB Term Rewriting System					TPDB Term Rewriting System				
SK90/2.08	✗	✗	✗	✗	StratRemmixed05/n001	✗	✗	✗	✗
SK90/2.15	✓	✓	✗	✗	StratRemmixed05/test830	✓	✓	✗	✗
SK90/2.17	✓	✓	✗	✗	TCSR04/Ex15Luc06GM	✗	✗	✗	✗
SK90/2.20	✓	✓	✗	✗	TCSR04/Ex15Luc06iGM	✗	✗	✗	✗
SK90/2.21	✗	✗	✗	✗	TCSR04/Ex15Luc06L	✗	✗	✗	✗
SK90/2.24	✓	✓	✗	✗	TCSR04/Ex18Luc06GM	✗	✗	✗	✗
SK90/2.28	✓	✓	✗	✗	TCSR04/Ex18Luc06iGM	✗	✗	✗	✗
SK90/2.30	✓	✓	✓	✓	TCSR04/Ex18Luc06L	✓	✓	✓	✓
SK90/2.37	✓	✓	✓	✓	TCSR04/Ex12Luc02cGM	✗	✗	✗	✗
SK90/2.40	✗	✗	✗	✗	TCSR04/Ex1GL02aL	✗	✗	✗	✗
SK90/2.49	✓	✓	✗	✗	TCSR04/Ex1GM99GM	✗	✗	✗	✗
SK90/2.51	✓	✓	✗	✗	TCSR04/Ex1Zan97L	✗	✗	✗	✗
SK90/2.56	✓	✓	✓	✓	TCSR04/Ex23Luc06GM	✗	✗	✗	✗
SK90/4.06	✗	✗	✗	✗	TCSR04/Ex23Luc06L	✓	✓	✓	✓
SK90/4.16	✓	✓	✗	✗	TCSR04/Ex24GM04L	✗	✗	✗	✗
SK90/4.17	✗	✗	✗	✗	TCSR04/Ex24Luc06GM	✗	✗	✗	✗
SK90/4.22	✗	✗	✗	✗	TCSR04/Ex25Luc06L	✓	✓	✓	✓
SK90/4.33	✓	✓	✗	✗	TCSR04/Ex44Luc96bGM	✗	✗	✗	✗
SK90/4.35	✓	✓	✓	✓	TCSR04/Ex44Luc96biGM	✗	✗	✗	✗
SK90/4.36	✓	✓	✗	✗	TCSR04/Ex44Luc96bL	✓	✓	✗	✗
SK90/4.37	✓	✓	✗	✗	TCSR04/Ex4715Bor03L	✓	✓	✓	✓
SK90/4.41	✓	✓	✓	✓	TCSR04/Ex4777Bor03GM	✗	✗	✗	✗
SK90/4.44	✓	✓	✓	✓	TCSR04/Ex6GM04C	✗	✗	✗	✗
SK90/4.46	✓	✓	✓	✓	TCSR04/Ex6GM04GM	✓	✓	✓	✓
SK90/4.50	✓	✓	✗	✗	TCSR04/Ex6GM04iGM	✓	✓	✗	✗
SK90/4.56	✓	✓	✓	✓	TCSR04/Ex6GM04L	✓	✓	✓	✓
StratRemAG01/4.14	✗	✗	✗	✗	TCSR04/Ex6Luc98L	✓	✓	✓	✓
StratRemAG01/4.16	✓	✓	✗	✗	TCSR04/Ex9BLR02L	✓	✓	✓	✓
StratRemAG01/4.2	✗	✗	✗	✗	TCSR04/Ex9Luc04GM	✗	✗	✗	✗
StratRemAG01/4.20	✓	✓	✗	✗	TCSR04/Ex9Luc04L	✓	✓	✗	✗
StratRemAG01/4.20a	✓	✓	✗	✗	TCSR04/ExConcZan97GM	✗	✗	✗	✗
StratRemAG01/4.21	✓	✓	✗	✗	TCSR04/ExConcZan97L	✓	✓	✓	✓
StratRemAG01/4.22	✓	✓	✗	✗	TCSR04/LOFLnosorts-noandL	✓	✓	✓	✓
StratRemAG01/4.32	✓	✓	✗	✗	Various04/02	✗	✗	✗	✗
StratRemAG01/4.37	✓	✓	✗	✗	Various04/22	✓	✓	✗	✗
StratRemAG01/4.37a	✓	✓	✗	✗	Various04/25	✓	✓	✓	✓
StratRemAG01/4.7	✓	✓	✗	✗	Various04/27	✓	✓	✓	✓
StratRemCSR05/Ex14AEGL02	✗	✗	✗	✗	Zantema05/z04	✗	✗	✗	✗
StratRemCSR05/Ex1GL02a	✗	✗	✗	✗	Zantema05/z13	✗	✗	✗	✗
StratRemCSR05/Ex1GM99	✓	✓	✗	✗	Zantema05/z14	✗	✗	✗	✗
StratRemCSR05/Ex1Zan97	✗	✗	✗	✗	Zantema05/z15	✗	✗	✗	✗
StratRemCSR05/Ex44Luc96b	✗	✗	✗	✗	Zantema05/z16	✗	✗	✗	✗
StratRemCSR05/Ex6GM04	✗	✗	✗	✗	Zantema05/z17	✓	✗	✗	✗
StratRemCSR05/Ex6Luc98	✗	✗	✗	✗	Zantema05/z18	✗	✗	✗	✗
StratRemCSR05/Ex9BLR02	✗	✗	✗	✗	Zantema05/z23	✗	✗	✗	✗
StratRemCSR05/ExConcZan97	✗	✗	✗	✗	Zantema05/z24	✗	✗	✗	✗
StratRemmixed05/bn111	✗	✗	✗	✗	Zantema05/z27	✗	✗	✗	✗
StratRemmixed05/ex1	✗	✗	✗	✗	Zantema05/z28	✗	✗	✗	✗
StratRemmixed05/ex2	✗	✗	✗	✗	Zantema15/delta	✓	✓	✗	✗
StratRemmixed05/ex3	✗	✗	✗	✗	Zantema15/ex14	✓	✓	✗	✗
StratRemmixed05/ex4	✗	✗	✗	✗	Legend: ✓= Proof found, ✗= No solution found				

C
DrAGoM Documentation

C.1. Tutorial: How to Use DrAGoM

The following tutorial provides an overview of the basic functionalities in DrAGoM. A detailed user manual, the API documentation, a pre-compiled version of the tool alongside the source code and additional informations can be found on the DrAGoM homepage[1].

System Requirements, Third-Party Libraries and Installation

DrAGoM has been implemented in Java. For the usage of DrAGoM a *Java Runtime Environment* (JRE) of version 1.8 or higher is required. The tool offers a *graphical user interface* (GUI) which allows the user to create and manipulate graph transformation systems and multiply annotated type graphs. Since DrAGoM is written in Java, it can be used on Linux, MacOS and Windows.
DrAGoM depends on the following two third-party libraries:

Z3-Java A Java library of the Z3 theorem prover from *Microsoft Research.*

JGoodies A Java library which offers reliable building blocks for Java applications.

DrAGoM is distributed under the terms of the 3-clause BSD open source license. The third-party library *JGoodies* is distributed under relaxed terms (2-clause) of the BSD license and *Z3-Java* is licensed under the MIT license.

The standard DrAGoM distribution is a pre-compiled `.jar` file that contains all required libraries except for the native library of the Z3 release, since these libraries depend on the user's operating system. DrAGoM calls the external SMT solver Z3 to compute annotations for the rewritable materialization (see also Section 11.2.2). Therefore the user has to download a Z3 release which fits the used operating system. The operating system's search path has to include the folder which contains the *libz3java* dynamic link library.

To compile DrAGoM from source instead, the user needs to configure the project's library dependencies. All required external libraries are provided in the resource folder of the source code archive on the DrAGoM homepage. A detailed description of the dependencies can be found in the DrAGoM program documentation, also available on the homepage.

In most operating systems, double-clicking on the `dragom.jar` file will open DrAGoM in GUI-mode (see Figure C.1). Otherwise, the GUI can be directly started from the command-line via

```
java -jar dragom.jar
```

[1] DrAGoM homepage: https://www.uni-due.de/theoinf/research/tools_dragom.php

Figure C.1.: DrAGoM GUI

In some Linux operating systems, DrAGoM might not be able to immediately locate the folder containing the *libz3java* library. To help the tool find the required library the user should set the global variable named LD_LIBRARY_PATH to point to the corresponding folder path, i.e., DrAGoM is launched by typing

```
LD_LIBRARY_PATH=<PATH_TO_LIBRARY_FOLDER> java -jar dragom.jar
```

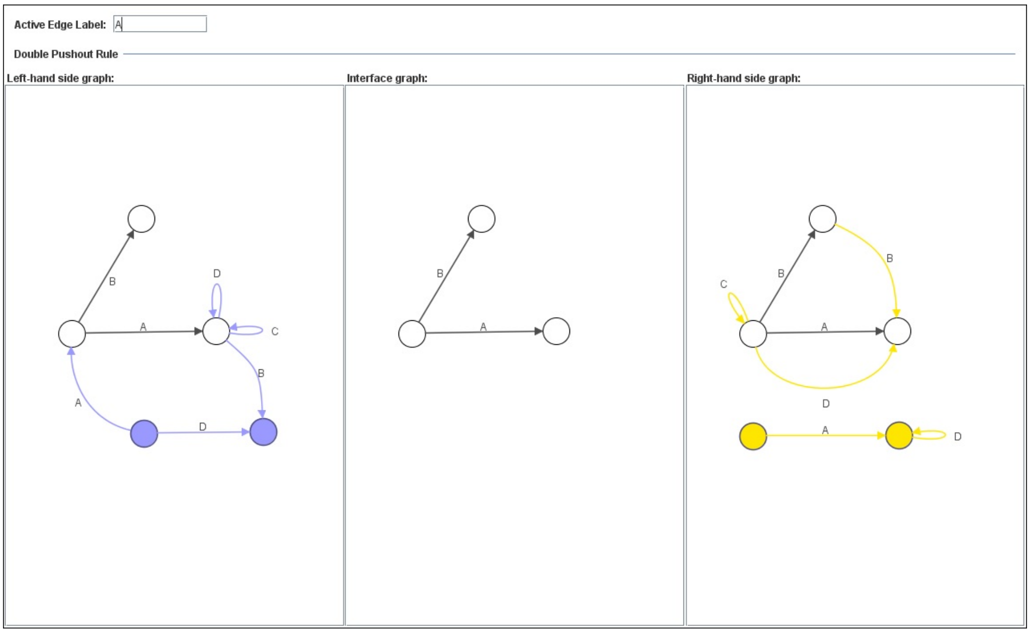

Active Edge Label: A

Figure C.2.: DrAGoM Add-Rule dialog

The Main Window and Main Menu

The main window of DrAGoM, which is shown in Figure C.1, has a simple and clean design. The idea behind this puristic presentation is to not overtax the user's first impression after launching DrAGoM. In the beginning, the main window only allows interaction with the main menu, i.e., the system bar displayed at the top of the screen. DrAGoM's main menu contains the following menus and options:

System Includes commands to generate or load graph transformation systems and multiply annotated type graphs.

- The option *"New..."* offers commands to open the *Add-Rule* dialog (see Figure C.2) or the *Create-Annotated-Type-Graph* dialog (see Figure C.3).
- The options *"Load..."* and *"Save..."* offer commands to load/save graph transformation systems in the SGF format or to load/save multiply annotated type graphs in the GXL format. All commands in the option *"Save..."* become available, once a corresponding data structure has been loaded/created.
- The option *"Exit"* closes DrAGoM.

Algorithms Includes commands to compute rewritable materializations, strongest postconditions or to let DrAGoM perform an invariant check. DrAGoM employs the SMT solver Z3 to compute annotations for the rewritable materialization. If the required native libraries were not found during the booting process, all commands in the *Algorithms* menu are disabled.

- The option *"Compute..."* offers commands to construct a rewritable materialization and the strongest postcondition. To enable the construction for the rewritable materialization, the user has to create/load a graph transformation system and an annotated type graph first. A strongest postcondition can be constructed once the rewritable materialization has been created.
- The option *"Invariant check"* is available if and only if the user created or loaded both, a graph transformation system and an annotated type graph. This option causes DrAGoM to perform an invariant check.

Help Includes commands to provide further information.

- The option *"Visit website..."* redirects the user to the DrAGoM homepage, where the user can find additional information.
- The option *"About..."* opens a dialog with additional information with respect to the current version and the DrAGoM license.

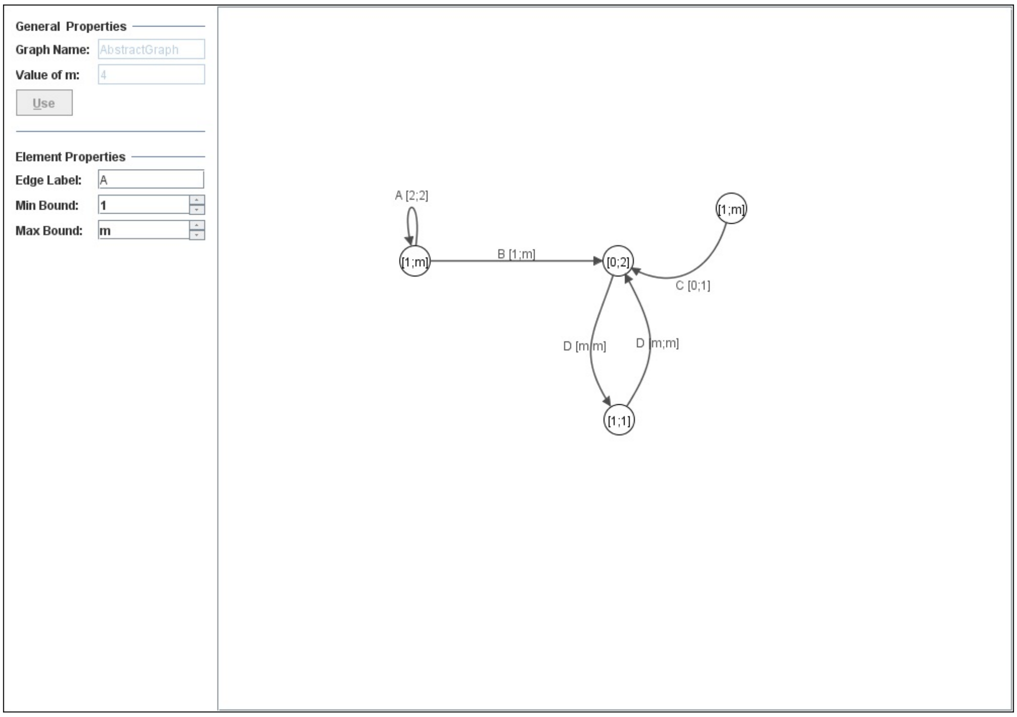
Graph Transformation System Creation

The user can select *System -> New... -> Graph transformation rule* from the main menu, to create a new graph transformation system. The *Add-Rule* dialog appears, which is displayed in Figure C.2. The dialog contains three graph panels which can be used to create the left-hand side, the right-hand side and the interface graph of a double-pushout graph transformation rule.

By double clicking on the interface graph panel, a node is added to the interface graph and two corresponding nodes are added to the left-hand side graph and right-hand side graph respectively. The same holds for loops, which are added via a right click on an existing node, and directed binary edges, which are added via a drag and drop movement from the source node to the target node while pressing the right mouse button. This intuitive interaction works similarly for the left-hand side and right-hand side graph panels. Elements added to either side only belong to the corresponding graph. All interface elements are drawn black and white, elements which only belong to the left-hand side are colored blue and elements which belong only to the right-hand side are colored yellow.

Please note that edge labels are determined during their creation. The user can enter the edge label that one wants to use for the next edge, by entering the label into the text field which can be found at the top of the dialog.

To complete the creation process, the user can hit the *Apply* button located at the bottom of the dialog. The created rule now is displayed in the top part of the main window and it generates a new graph transformation system which consists of the single rule. A right click on the rule panel shows a pop up menu where the user has the option to either add a new rule, delete the currently displayed rule or switch between the displayed rules of our graph transformation system. Furthermore, the option to save the graph transformation system is now enabled in the main menu.



General Properties

Graph Name: AbstractGraph

Value of m: 4

Element Properties

Edge Label: A

Min Bound: 1

Max Bound: m

Figure C.3.: DrAGoM Create-Annotated-Type-Graph dialog

Multiply Annotated Type Graph Creation

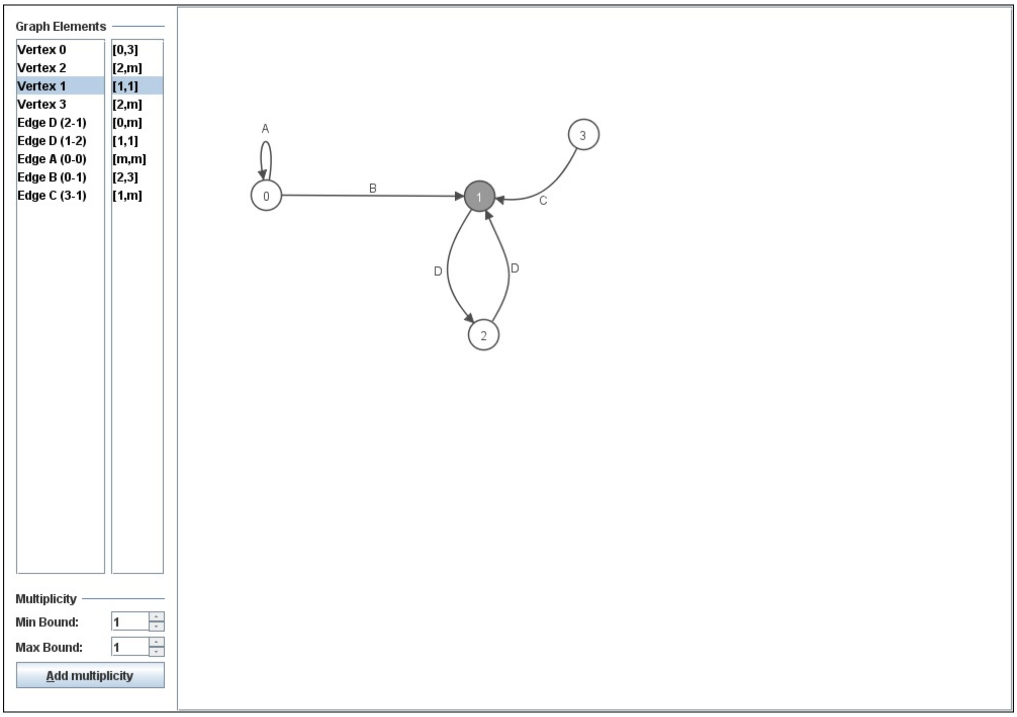
Next, we explain how one can create an annotated type graph. For this purpose the user selects *System -> New... -> Annotated type graph* from the main menu. The *Create-Annotated-Type-Graph* dialog appears, which is displayed in Figure C.3.

Before the user can create the graph structure, one needs to define a name for the new annotated type graph and choose a value m which represents the value $*$ ($m$any) of the ordered monoid $\mathcal{M}_n$ (see also Example 8.3 and Definition 8.6). Subsequently, the user fills out the corresponding text fields which are located in the *General Properties* section on the top left of the dialog. Once the user filled out both text fields, a click on the *Use* button confirms the choice. Now the graph panel on the right side of the dialog gets enabled such that the user can create the annotated type graph.

The graph creation works similar to the creation of graphs for the graph transformation rule. Please note that multiplicities for graph elements are determined at the moment they are added to the graph. The values are taken from the respective combo boxes which are located in the *Element Properties* section on the left side of the dialog.

As soon as the user hits the *Apply* button, the annotated type graph is displayed in the bottom left corner of the main window. The option to save multiply annotated type graphs is now enabled in the system bar. The user can double click on the graph panel to maximize its view or right click on it to open a pop up menu.

The first option in the pop up menu is to add a multiplicity to the annotated type graph, i.e. to turn the annotated type graph into a multiply annotated type graph. If the user chooses this option the *Add-Multiplicity* dialog appears, which is shown in Figure C.4.



Graph Elements

Vertex 0	[0,3]
Vertex 2	[2,m]
Vertex 1	[1,1]
Vertex 3	[2,m]
Edge D (2-1)	[0,m]
Edge D (1-2)	[1,1]
Edge A (0-0)	[m,m]
Edge B (0-1)	[2,3]
Edge C (3-1)	[1,m]

Multiplicity

Min Bound: 1

Max Bound: 1

Figure C.4.: DrAGoM Add-Multiplicity dialog

To edit multiplicities for graph elements, the user can select the elements in the graph panel of this dialog. Every time that the user selects an element, two entries in the lists, which are located in the *Graph Elements* section on the left side of the dialog, are highlighted. The first list shows the element's corresponding identifier while the second list shows the current multiplicity to be added to the element. The default value for all elements is $[0, m]$ which corresponds to the annotation bounds $[0, *]$.

To change the multiplicity of an element one can select the desired bounds in the combo boxes in the *Multiplicity* section located at the bottom left of the dialog. Afterwards, the user can hit the *Add multiplicity* button to overwrite the selected elements annotation bounds with the ones selected in the combo boxes.

Once the user is done, the dialog can be closed by hitting the *Apply* button. DrAGoM will check if the new multiplicity was already covered by an existing one in the set of annotations for this multiply annotated type graph. If it was not covered yet, the created multiplicity is added to the set, otherwise it is discarded and we get a notification.

The two remaining options in the pop up menu of the annotated type graph panel, allows the user to delete existing multiplicities from the set of available annotations or to switch the currently displayed multiplicity.

Rewritable Materialization Construction

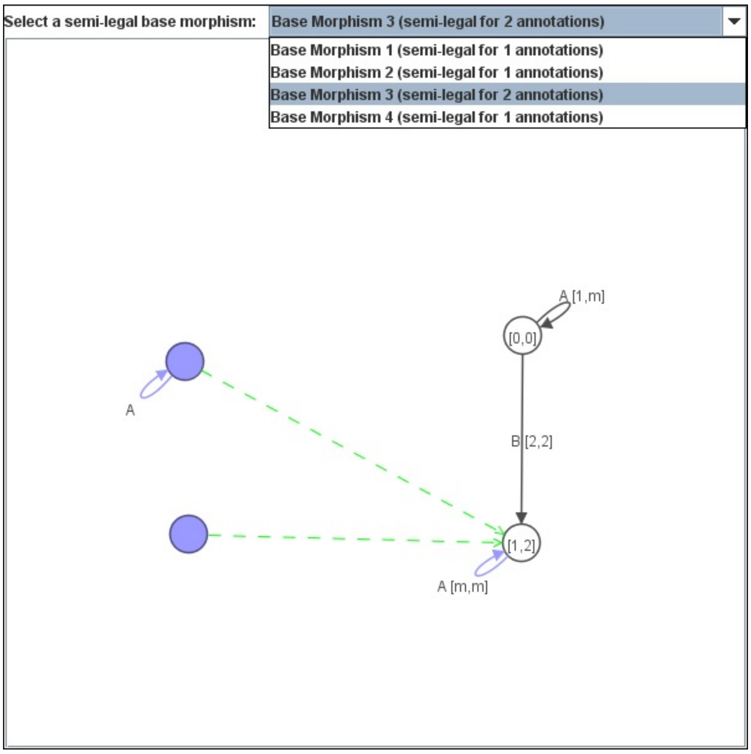
Now, that the user has created both, a graph transformation system and a multiply annotated type graph, the option to compute rewritable materializations gets enabled in the menu bar (*Algorithms -> Compute... -> Materialization*). The graph transformation rule and the multiply annotated type graph displayed in the main window are being used for the computation. To proceed the user needs to choose a semi-legal base morphism φ from the left-hand side graph of the rule to the annotated type graph. The *Select-Base-Morphism* dialog appears (see Figure C.5) which displays the set of available morphisms. DrAGoM filters the set of morphisms to only contain semi-legal ones. If there does not exist a semi-legal morphism, no graph contained in the graph language can be rewritten since none of them contain a concrete instance of the left-hand side.

The user can select the semi-legal base morphism by choosing a corresponding entry in the combo box, located at the top of the dialog. Afterwards, the user can press the *Apply* button, which initializes the materialization algorithm explained in Section 11.2.1. The rewritable materialization is displayed in the main window and the option to compute the strongest postcondition is enabled.

A right click on the materialization graph panel shows a pop up menu with options to either switch the currently displayed multiplicity for the rewritable materialization or to simplify its visualization. The simplification hides all elements annotated $[0, 0]$ and all edges with a hidden source or target node.

Invariant Checking

To perform an invariant check the user chooses (*Algorithms -> Invariant check*). DrAGoM checks for all rules in the graph transformation system and for all semi-legal base morphisms if the language of the constructed postcondition is included in the initial graph language specified by the multiply annotated type graph. For the inclusion check, DrAGoM uses the sufficient condition from Proposition 8.20. Whenever DrAGoM finds legal morphisms between the corresponding multiply annotated type graphs, we can infer the inclusion. However, if DrAGoM fails to find a legal morphism the inclusion could still hold. In this case, the rule, the rewritable

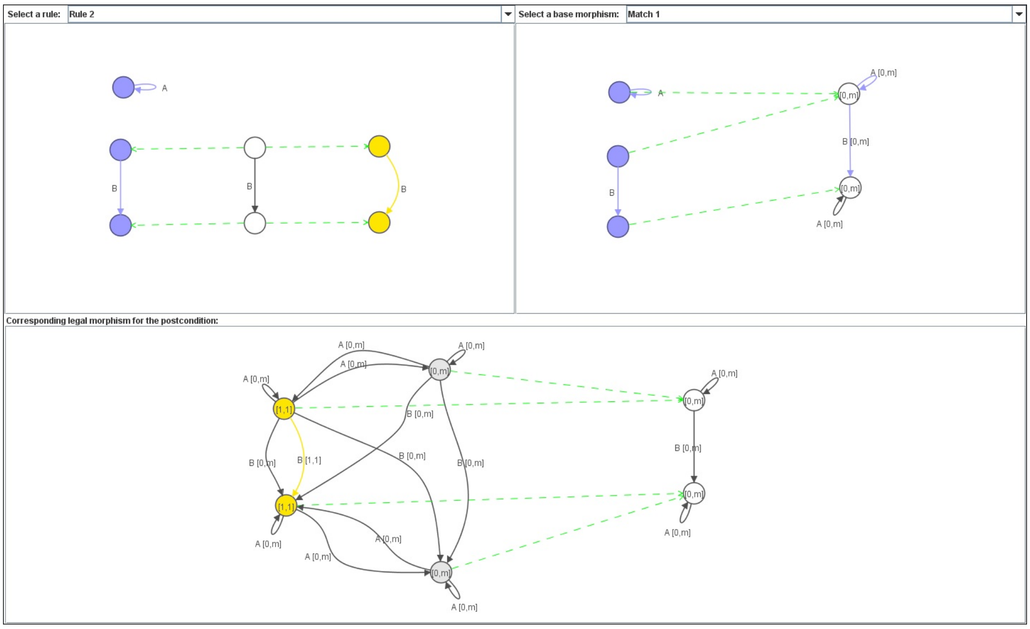


Select a semi-legal base morphism: Base Morphism 3 (semi-legal for 2 annotations)

Base Morphism 1 (semi-legal for 1 annotations)
Base Morphism 2 (semi-legal for 1 annotations)
Base Morphism 3 (semi-legal for 2 annotations)
Base Morphism 4 (semi-legal for 1 annotations)

Figure C.5.: DrAGoM Select-Base-Morphism dialog

materialization and the strongest postcondition graph with the multiplicity for which the check fails are displayed in the main window. Otherwise, in case of a successful check, the *Invariant-Proof* dialog appears (see Figure C.6) in which the user can see every found legal morphism by choosing a rule and base morphism via the combo boxes located at the top of the dialog.



Select a rule: Rule 2

Select a base morphism: Match 1

Corresponding legal morphism for the postcondition:

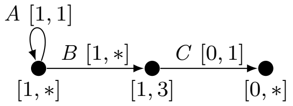
Figure C.6.: DrAGoM Invariant-Proof dialog

C.2. The GXL Format for Multiply Annotated Type Graphs

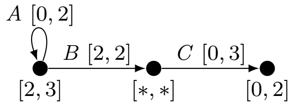
In this section we give an example for the GXL encoding of a multiply annotated type graph. The annotated graphs are stored into `.gxl`-files. A detailed description of the GXL format can be found on the website at http://www.gupro.de/GXL/.

Let the following multiply annotated type graph $T[M]$ be given, with the set of double multiplicities $M = \{(\ell_1, u_1), (\ell_2, u_2)\}$ over the annotation functor $\mathcal{B}^3$, i.e. the multiplicity of each graph element is indicated by an element of $\mathcal{M}_3 = \{0, 1, 2, 3, *\}$:

$T[\ell_1, u_1] =$



$T[\ell_2, u_2] =$



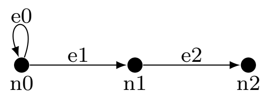
The multiply annotated type graph $T[M]$ is encoded in the following GXL format:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE gxl SYSTEM "http://www.gupro.de/GXL/gxl-1.0.dtd">
<gxl>
  <graph id="AbstractGraph">

    <node id="limit">
      <attr name="limit">
        <int>4</int>
      </attr>
    </node>

    <node id="n0">
      <attr name="min">
        <seq>
          <int>1</int>
          <int>2</int>
        </seq>
      </attr>
      <attr name="max">
        <seq>
          <int>4</int>
          <int>3</int>
        </seq>
      </attr>
    </node>

    <node id="n1">
      <attr name="min">
        <seq>
          <int>1</int>
          <int>4</int>
        </seq>
      </attr>
      <attr name="max">
```

```
        <seq>
          <int>3</int>
          <int>4</int>
        </seq>
      </attr>
    </node>

    <node id="n2">
      <attr name="min">
        <seq>
          <int>0</int>
          <int>0</int>
        </seq>
      </attr>
      <attr name="max">
        <seq>
          <int>4</int>
          <int>2</int>
        </seq>
      </attr>
    </node>

    <edge id="e0" from="n0" to="n0">
      <attr name="label">
        <string>A</string>
      </attr>
      <attr name="min">
        <seq>
          <int>1</int>
          <int>0</int>
        </seq>
      </attr>
      <attr name="max">
        <seq>
          <int>1</int>
          <int>2</int>
        </seq>
      </attr>
    </edge>

    <edge id="e1" from="n0" to="n1">
      <attr name="label">
        <string>B</string>
      </attr>
      <attr name="min">
        <seq>
          <int>1</int>
          <int>2</int>
        </seq>
      </attr>
      <attr name="max">
        <seq>
          <int>4</int>
          <int>2</int>
        </seq>
      </attr>
    </edge>

    <edge id="e2" from="n1" to="n2">
      <attr name="label">
        <string>C</string>
      </attr>
      <attr name="min">
        <seq>
          <int>0</int>
          <int>0</int>
        </seq>
      </attr>
      <attr name="max">
        <seq>
          <int>1</int>
          <int>3</int>
        </seq>
      </attr>
    </edge>

  </graph>
</gxl>
```

As evident from the encoding shown above, the GXL format uses identifiers to encode the structure of the type graph. For completeness sake, we depict the identifier reference for each element of the encoded type graph $T[M]$:

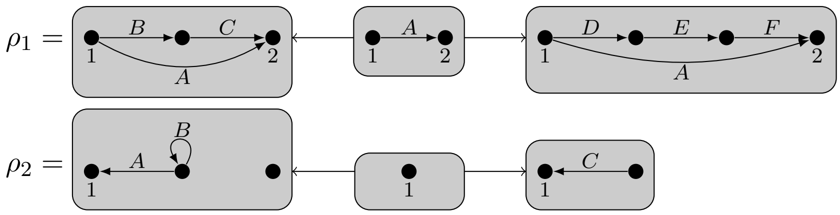
$T =$



C.3. The SGF Format for Graph Transformation Systems

In this section we give an example for the Sgf encoding of a graph transformation system. The graph transformation systems are stored into `.sgf`-files, where one file in the Sgf format can contain several graph transformation systems at the same time. A formal description of the Sgf format is given in [Bru15].

Let the following graph transformation system $\mathcal{R} = \{\rho_1, \rho_2\}$ be given, where ρ_1 and ρ_2 are depicted below:



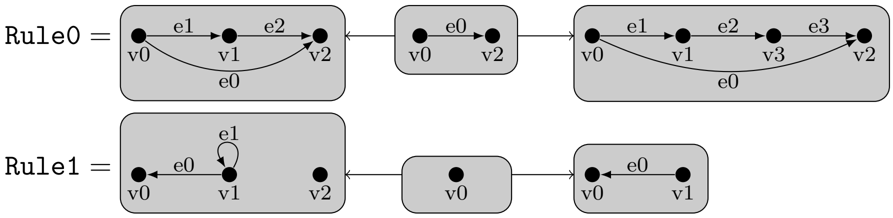
The Sgf format represents a double-pushout rule using a corresponding morphism from the left-hand side graph to the right-hand side graph instead of saving two rule morphisms and the interface graph. Please note that the Sgf format only supports injective rule morphisms. In the encoding, nodes are introduced implicitly via edge definitions, in contrast to distinct nodes which need to be added explicitly in the respective graph encoding.

The graph transformation system $\mathcal{R}$ is encoded in the following Sgf format:

```
leftGraph0 = graph {
  e0:A(v0,v2);
  e1:B(v0,v1);
  e2:C(v1,v2);
};

rightGraph0 = graph {
  e0:A(v0,v2);
  e1:D(v0,v1);
  e2:E(v1,v3);
  e3:F(v3,v2);
};

Morphism0 = morphism from leftGraph0 to rightGraph0 {
  v0 => v0;
  v2 => v2;
  e0 => e0;
};

leftGraph1 = graph {
  e0:A(v1,v0);
  e1:B(v1,v1);
  node v2;
};
```

```
rightGraph1 = graph {
  e0:C(v1,v0);
};

Morphism1 = morphism from leftGraph1 to rightGraph1 {
  v0 => v0;
};

Rule0 = rule {
  left = leftGraph0;
  right = rightGraph0;
  morphism = Morphism0;
};

Rule1 = rule {
  left = leftGraph1;
  right = rightGraph1;
  morphism = Morphism1;
};

result = gts {
  rules = [
    Rule1,
    Rule0
  ];
};
```

For the sake of completeness, we depict the identifier reference for each element of the encoded graph transformation system `result` $= \{\texttt{Rule0}, \texttt{Rule1}\}$:



References

Publications of Dennis Nolte

[CH+19] A. Corradini, T. Heindel, B. König, D. Nolte, and A. Rensink. “Rewriting Abstract Structures: Materialization Explained Categorically”. In: *Foundations of Software Science and Computation Structures.* Ed. by M. Bojańczyk and A. Simpson. Cham: Springer International Publishing, 2019, pp. 169–188. DOI: 10.1007/978-3-030-17127-8_10. arXiv: 1902.04809 [cs.LO].

[CKN19] A. Corradini, B. König, and D. Nolte. “Specifying Graph Languages with Type Graphs”. In: *Journal of Logical and Algebraic Methods in Programming* Vol. 104 (2019), pp. 176–200. DOI: 10.1016/j.jlamp.2019.01.005.

[KNN19] B. König, M. Nederkorn, and D. Nolte. “CoReS: A Tool for Computing Core Graphs via SAT/SMT Solvers”. In: *Journal of Logical and Algebraic Methods in Programming* (2019). Submitted.

[KNN18] B. König, M. Nederkorn, and D. Nolte. “CoReS: A Tool for Computing Core Graphs via SAT/SMT Solvers (Tool Presentation Paper)”. In: *Proc. of ICGT ’18 (International Conference on Graph Transformation).* LNCS 10887. Springer, 2018, pp. 37–42. DOI: 10.1007/978-3-319-92991-0_3.

[KN+18] B. König, D. Nolte, J. Padberg, and A. Rensink. “A Tutorial on Graph Transformation”. In: *Graph Transformation, Specifications, and Nets – In Memory of Hartmut Ehrig.* Ed. by R. Heckel and G. Taentzer. LNCS 10800. Springer, 2018, pp. 1–22. DOI: 10.1007/978-3-319-75396-6_5.

[CKN17] A. Corradini, B. König, and D. Nolte. “Specifying Graph Languages with Type Graphs”. In: *Proc. of ICGT ’17 (International Conference on Graph Transformation).* LNCS 10373. Springer, 2017, pp. 73–89. DOI: 10.1007/978-3-319-61470-0_5. arXiv: 1704.05263 [cs.FL].

[Nol17] D. Nolte. “Analysis and Abstraction of Graph Transformation Systems via Type Graphs”. In: *STAF 2017 Doctoral Symposium.* Vol. 1955. CEUR Workshop Proceedings. 2017.

[ZNK16] H. Zantema, D. Nolte, and B. König. “Termination of Term Graph Rewriting”. In: *Proc. of WST ’16 (Workshop on Termination).* 2016.

[BK+15] H. J. S. Bruggink, B. König, D. Nolte, and H. Zantema. “Proving Termination of Graph Transformation Systems Using Weighted Type Graphs over Semirings”. In: *Proc. of ICGT ’15 (International Conference on Graph Transformation).* 2015, pp. 52–68. DOI: 10.1007/978-3-319-21145-9_4. arXiv: 1505.01695 [cs.LO].

All References

[AH+13] P. A. Abdulla, L. Holík, B. Jonsson, O. Lengál, C.Q. Trinh, and T. Vojnar. "Verification of Heap Manipulating Programs with Ordered Data by Extended Forest Automata". In: *Proc. of ATVA '13*. LNCS 8172. 2013, pp. 224–239 (cit. on p. 102).

[AHS09] J. Adamek, H. Herrlich, and G.E. Strecker. *Abstract and Concrete Categories: The Joy of Cats*. Dover books on mathematics. Dover Publications, 2009 (cit. on p. 21).

[Bac15] P. Backes. "Cluster Abstraction of Graph Transformation Systems". PhD thesis. Saarland University, 2015 (cit. on pp. 105, 127).

[BR15a] P. Backes and J. Reineke. "Analysis of Infinite-State Graph Transformation Systems by Cluster Abstraction". In: ed. by Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen. Vol. 8931. LNCS. Springer Berlin Heidelberg, 2015, pp. 135–152 (cit. on pp. 105, 127, 139).

[BR15b] P. Backes and J. Reineke. "ASTRA: A Tool for Abstract Interpretation of Graph Transformation Systems". In: ed. by Bernd Fischer and Jaco Geldenhuys. Vol. 9232. LNCS. Springer International Publishing, 2015, pp. 13–19 (cit. on p. 139).

[BCK01] P. Baldan, A. Corradini, and B. König. "A Static Analysis Technique for Graph Transformation Systems". In: vol. 2154. LNCS. Springer-Verlag, 2001, pp. 381–395 (cit. on p. 138).

[BST10] C. Barrett, A. Stump, and C. Tinelli. "The SMT-LIB Standard – Version 2.0". In: *Proc. of the 8th International Workshop on Satisfiability Modulo Theories (SMT '10)*. Edinburgh, Scotland. July 2010 (cit. on pp. 54, 136).

[Bau06] J. Bauer. "Analysis of Communication Topologies by Partner Abstraction". PhD thesis. Saarland University, 2006 (cit. on pp. 127, 139).

[BB+08] J. Bauer, I. Boneva, M. E. Kurbán, and A. Rensink. "A Modal-Logic Based Graph Abstraction". In: *Proc. of ICGT '08*. LNCS 5214. Springer, 2008, pp. 321–335 (cit. on p. 92).

[BW07] J. Bauer and R. Wilhelm. "Static Analysis of Dynamic Communication Systems by Partner Abstraction". In: *Proc. of SAS '07*. LNCS 4634. Springer, 2007, pp. 249–264 (cit. on pp. 105, 127).

[Blu14] C. Blume. "Graph Automata and Their Application to the Verification of Dynamic Systems". PhD thesis. University of Duisburg-Essen, 2014 (cit. on pp. 5, 98, 102, 138, 142, 146).

[BB+12] C. Blume, H. J. S. Bruggink, D. Engelke, and B. König. "Efficient Symbolic Implementation of Graph Automata with Applications to Invariant Checking". In: *Proc. of ICGT '12*. LNCS 7562. Springer, 2012, pp. 264–278 (cit. on pp. 98, 102).

[BB+13] C. Blume, H. J. S. Bruggink, M. Friedrich, and B. König. "Treewidth, Pathwidth and Cospan Decompositions with Applications to Graph-Accepting Tree Automata". In: *Journal of Visual Languages & Computing* 24.3 (2013), pp. 192–206 (cit. on p. 95).

[BL+07] I. Bogudlo, T. Lev-Ami, T. Reps, and M. Sagiv. “Revamping TVLA: Making Parametric Shape Analysis Competitive”. In: *Computer Aided Verification*. Ed. by Werner Damm and Holger Hermanns. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 221–225 (cit. on p. 139).

[BH+05] P. Bottoni, K. Hoffman, F. Parisi Presicce, and G. Taentzer. “High-Level Replacement Units and their Termination Properties”. In: *Journal of Visual Languages and Computing* 16.6 (2005), pp. 485–507 (cit. on p. 71).

[BJ+00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. “Regular Model Checking”. In: *Computer Aided Verification*. Springer Berlin Heidelberg, 2000, pp. 403–418 (cit. on p. 1).

[Bru15] H. J. S. Bruggink. *Grez User Manual*. www.ti.inf.uni-due.de/research/tools/grez. 2015 (cit. on pp. 54, 134, 213).

[BK08] H. J. S. Bruggink and B. König. “On the Recognizability of Arrow and Graph Languages”. In: *Proc. of ICGT ’08*. LNCS 5214. Springer, 2008, pp. 336–350 (cit. on pp. 95, 96).

[BK+15] H. J. S. Bruggink, B. König, D. Nolte, and H. Zantema. “Proving Termination of Graph Transformation Systems Using Weighted Type Graphs over Semirings”. In: *Proc. of ICGT ’15 (International Conference on Graph Transformation)*. 2015, pp. 52–68. DOI: 10.1007/978-3-319-21145-9_4. arXiv: 1505.01695 [cs.LO] (cit. on p. ix).

[BKZ14] H. J. S. Bruggink, B. König, and H. Zantema. “Termination Analysis of Graph Transformation Systems”. In: *Proc. of TCS 2014*. Vol. 8705. LNCS. Springer, 2014 (cit. on pp. ix, 39, 47, 49, 50, 54, 61, 71).

[CD+11] C. Calcagno, D. Distefano, P. W. O’Hearn, and H. Yang. “Compositional Shape Analysis by Means of Bi-Abduction”. In: *Journal of the ACM* 58.6 (2011), 26:1–26:66 (cit. on p. 105).

[CM77] A. K. Chandra and P. M. Merlin. “Optimal Implementation of Conjunctive Queries in Relational Data Bases”. In: *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*. STOC ’77. Boulder, Colorado, USA: ACM, 1977, pp. 77–90 (cit. on p. 77).

[CR08] B. E. Chang and X. Rival. “Relational inductive shape analysis”. In: *Proc. of POPL ’08*. ACM, 2008, pp. 247–260 (cit. on p. 105).

[CG+03] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. “Counterexample-guided abstraction refinement for symbolic model checking”. In: *Journal of the ACM* 50.5 (2003), pp. 752–794 (cit. on pp. 1, 75, 138).

[CL03] J. R. B. Cockett and S. Lack. “Restriction categories II: partial map classification”. In: *TCS* 294.1–2 (2003), pp. 61–102 (cit. on p. 108).

[CD11] A. Corradini and F. Drewes. “Term graph rewriting and parallel term rewriting”. In: *Proceedings of the 6th International Workshop on Computing with Terms and Graphs (Termgraph)*. Vol. 48. Electronic Proceedings in Theoretical Computer Science. 2011, pp. 3–18 (cit. on pp. 40, 58, 59, 61, 64).

[CH+06] A. Corradini, T. Heindel, F. Hermann, and B. König. “Sesqui-pushout rewriting”. In: *Proc. of ICGT ’06 (International Conference on Graph Transformation)*. LNCS 4178. Springer, 2006, pp. 30–45 (cit. on pp. 105, 110, 179).

[CH+19] A. Corradini, T. Heindel, B. König, D. Nolte, and A. Rensink. “Rewriting Abstract Structures: Materialization Explained Categorically”. In: *Foundations of Software Science and Computation Structures*. Ed. by M. Bojańczyk and A. Simpson. Cham: Springer International Publishing, 2019, pp. 169–188. DOI: 10.1007/978-3-030-17127-8_10. arXiv: 1902.04809 [cs.LO] (cit. on p. x).

[CKN17] A. Corradini, B. König, and D. Nolte. “Specifying Graph Languages with Type Graphs”. In: *Proc. of ICGT ’17 (International Conference on Graph Transformation)*. LNCS 10373. Springer, 2017, pp. 73–89. DOI: 10.1007/978-3-319-61470-0_5. arXiv: 1704.05263 [cs.FL] (cit. on p. x).

[CKN19] A. Corradini, B. König, and D. Nolte. “Specifying Graph Languages with Type Graphs”. In: *Journal of Logical and Algebraic Methods in Programming* Vol. 104 (2019), pp. 176–200. DOI: 10.1016/j.jlamp.2019.01.005 (cit. on p. x).

[CMR96] A. Corradini, U. Montanari, and F. Rossi. “Graph Processes”. In: *Fundamenta Informaticae* 26.3/4 (1996), pp. 241–265 (cit. on p. 33).

[CM+97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. “Algebraic Approaches to Graph Transformation—Part I: Basic Concepts and Double Pushout Approach”. In: *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*. Ed. by G. Rozenberg. World Scientific, 1997. Chap. 3 (cit. on pp. 2, 21, 25, 31).

[CR93] A. Corradini and F. Rossi. “Hyperedge replacement jungle rewriting for term-rewriting systems and logic programming”. In: *Theoretical Computer Science* 109 (1993), pp. 7–48 (cit. on p. 58).

[Cou90] B. Courcelle. “The Monadic Second-Order Logic of Graphs I. Recognizable Sets of Finite Graphs”. In: *Information and Computation* 85 (1990), pp. 12–75 (cit. on p. 101).

[CE12] B. Courcelle and J. Engelfriet. *Graph Structure and Monadic Second-Order Logic, A Language-Theoretic Approach*. Cambridge University Press, June 2012 (cit. on pp. 75, 95, 101).

[Cou96] P. Cousot. “Abstract Interpretation”. In: *ACM Computing Surveys* 28.2 (1996) (cit. on p. 105).

[DF+12] V. Danos, J. Feret, W. Fontana, R. Harmer, J. Hayman, J. Krivine, C. D. Thompson-Walsh, and G. Winskel. “Graphs, Rewriting and Pathway Reconstruction for Rule-Based Models”. In: *Proc. of FSTTCS ’12*. Vol. 18. LIPIcs. Schloss Dagstuhl – Leibniz Center for Informatics, 2012 (cit. on p. 25).

[DJ90] N. Dershowitz and J.-P. Jouannaud. “Rewrite Systems”. In: *Formal Models and Semantics, Handbook of Theoretical Computer Science*. Ed. by Jan van Leeuwen. Vol. B. Elsevier, 1990. Chap. 6, pp. 243–320 (cit. on pp. 23, 55).

[DOY06] D. Distefano, P. W. O’Hearn, and H. Yang. “A Local Shape Analysis based on Separation Logic”. In: *Proc. of TACAS ’06*. LNCS 3920. Springer, 2006, pp. 287–302 (cit. on p. 102).

[DG17] J. Dyck and H. Giese. “K-Inductive Invariant Checking for Graph Transformation Systems”. In: *Graph Transformation*. Ed. by Juan de Lara and Detlef Plump. Vol. 10373. LNCS. Cham: Springer, 2017, pp. 142–158 (cit. on p. 144).

[DT87] R. Dyckhoff and W. Tholen. “Exponentiable morphisms, partial products and pullback complements”. In: *Journal of Pure and Applied Algebra* 49.1-2 (1987), pp. 103–116 (cit. on pp. 5, 105, 110, 177).

[Ehr79] H. Ehrig. “Introduction to the Algebraic Theory of Graph Grammars (A Survey)”. In: *Graph-Grammars and Their Application to Computer Science and Biology*. Vol. 73. LNCS. Springer, 1979, pp. 1–69 (cit. on pp. 2, 28, 31).

[EE+05] H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay. “Termination Criteria for Model Transformation”. In: *Proc. of FASE 2005*. Vol. 3442. LNCS. Springer, 2005 (cit. on p. 71).

[EE+06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. Springer, 2006 (cit. on pp. 30, 31, 163).

[EE+99] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, eds. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.2: Applications, Languages and Tools*. World Scientific, 1999 (cit. on pp. 23, 25, 31).

[EE+15] H. Ehrig, C. Ermel, U. Golas, and F. Hermann. *Graph and Model Transformation – General Framework and Applications*. Monographs in Theoretical Computer Science. Springer, 2015 (cit. on p. 23).

[EGH+13] H. Ehrig, U. Golas, F. Hermann, et al. “Categorical Frameworks for Graph Transformation and HLR Systems Based on the DPO Approach”. In: *Bulletin of EATCS* 3.102 (2013), pp. 111–121 (cit. on p. 2).

[EH+04] H. Ehrig, A. Habel, J. Padberg, and U. Prange. “Adhesive High-Level Replacement Categories and Systems”. In: *Proc. of ICGT ’04 (International Conference on Graph Transformation)*. LNCS 3256. 2004, pp. 144–160 (cit. on p. 2).

[EK+99] H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, eds. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.3: Concurrency, Parallellism, and Distribution*. World Scientific, 1999 (cit. on p. 31).

[EPS73] H. Ehrig, M. Pfender, and H. Schneider. “Graph grammars: An algebraic approach”. In: *Proc. 14th IEEE Symp. on Switching and Automata Theory.* 1973, pp. 167–180 (cit. on pp. 2, 25).

[EWZ08] J. Endrullis, J. Waldmann, and H. Zantema. “Matrix Interpretations for Proving Termination of Term Rewriting”. In: *Journal of Automated Reasoning* 40.2–3 (2008), pp. 195–220 (cit. on pp. 4, 39, 41).

[EZ15] J. Endrullis and H. Zantema. “Proving non-termination by finite automata”. In: *RTA ’15.* Vol. 36. LIPIcs. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 160–176 (cit. on pp. 1, 3, 75).

[Fre72] P. Freyd. “Aspects of Topoi”. In: *Bulletin of the Australian Mathematical Society* 7.1 (1972), pp. 1–76 (cit. on p. 113).

[FO97] L. Fribourg and H. Olsén. “Reachability sets of parameterized rings as regular languages”. In: *Proceedings of Infinity ’97.* Vol. 9. Electronic Notes in Theoretical Computer Science. Elsevier, 1997 (cit. on p. 1).

[Ges90] A. Geser. “Relative Termination”. PhD thesis. Universität Passau, 1990 (cit. on p. 48).

[GHW04] A. Geser, D. Hofbauer, and J. Waldmann. “Match-bounded string rewriting”. In: *Applicable Algebra in Engineering, Communication and Computing* 15.3–4 (2004), pp. 149–171 (cit. on p. 1).

[Hab92] A. Habel. *Hyperedge Replacement: Grammars and Languages.* LNCS 643. Springer, 1992 (cit. on p. 102).

[HKP88] A. Habel, H.-J. Kreowski, and D. Plump. “Jungle Evaluation”. In: *Proc. Recent Trends in Data Type Specification.* Vol. 332. LNCS. Springer, 1988, pp. 92–112 (cit. on p. 58).

[HP05] A. Habel and K.-H. Pennemann. “Nested Constraints and Application Conditions for High-Level Structures”. In: *Formal Methods in Software and Systems Modeling. Essays Dedicated to Hartmut Ehrig, on the Occasion of His 60th Birthday.* LNCS 3393. Springer, 2005, pp. 294–308 (cit. on pp. 75, 102, 127).

[HW95] R. Heckel and A. Wagner. “Ensuring consistency of conditional graph rewriting – a constructive approach”. In: *Proc. of the Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation.* Vol. 2. ENTCS. 1995 (cit. on p. 79).

[Hei09] T. Heindel. “A Category Theoretical Approach to the Concurrent Semantics of Rewriting – Adhesive Categories and Related Concepts”. PhD thesis. Universität Duisburg-Essen, Sept. 2009 (cit. on p. 164).

[HJ+15] J. Heinen, C. Jansen, J.-P. Katoen, and T. Noll. “Verifying pointer programs using graph grammars”. In: *Science of Computer Programming* 97 (2015), pp. 157–162 (cit. on p. 139).

[HJ+04] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. “Abstractions from Proofs”. In: *Proc. of POPL ’04.* ACM, 2004, pp. 232–244 (cit. on p. 3).

[HW06] D. Hofbauer and J. Waldmann. “Termination of String Rewriting with Matrix Interpretations”. In: *Proc. of RTA ’06.* LNCS 4098. 2006, pp. 328–342 (cit. on pp. 39, 41–43).

[HS+06] R. C. Holt, A. Schürr, S. E. Sim, and A. Winter. “GXL: A graph-based standard exchange format for reengineering”. In: *Science of Computer Programming* 60.2 (2006), pp. 149–170 (cit. on p. 132).

[Jac99] B. Jacobs. *Categorical Logic and Type Theory.* Vol. 141. Studies in Logic and the Foundation of Mathematics. Elsevier, 1999 (cit. on pp. 118, 119).

[Joh02] P. T. Johnstone. *Sketches of an Elephant: A Topos Theory Compendium.* Vol. 1. Clarendon Press, 2002 (cit. on p. 108).

[KS+05] R. Kennaway, P. Severi, R. Sleep, and F.-J. de Vries. “Infinitary Rewriting: From Syntax to Semantics”. In: *Processes, Terms and Cycles: Steps on the Road to Infinity:Essays Dedicated to Jan Willem Klop on the Occasion of His 60th Birthday.* Vol. 3838. LNCS. Springer, 2005 (cit. on p. 72).

[KV03] R. Kennaway and F.-J. de Vries. “Infinitary rewriting”. In: *Term Rewriting Systems, by Terese.* Cambridge University Press, 2003, pp. 668–711 (cit. on p. 72).

[KV05] J. W. Klop and R. C. de Vrijer. “Infinitary Normalization”. In: *We Will Show Them! Essays in Honour of Dov Gabbay.* Vol. 2. College Publications, 2005, pp. 169–192 (cit. on p. 72).

[Kön99] B. König. “Description and Verification of Mobile Processes with Graph Rewriting Techniques”. PhD thesis. Technische Universität München, 1999 (cit. on p. 86).

[Kön00] B. König. “A general framework for types in graph rewriting”. In: *Proc. of FST TCS ’00.* LNCS 1974. Springer-Verlag, 2000, pp. 373–384 (cit. on p. 117).

[KK06] B. König and V. Kozioura. “Counterexample-guided Abstraction Refinement for the Analysis of Graph Transformation Systems”. In: vol. 3920. LNCS. Springer, 2006, pp. 197–211 (cit. on p. 138).

[KNN18] B. König, M. Nederkorn, and D. Nolte. “CoReS: A Tool for Computing Core Graphs via SAT/SMT Solvers (Tool Presentation Paper)”. In: *Proc. of ICGT ’18 (International Conference on Graph Transformation).* LNCS 10887. Springer, 2018, pp. 37–42. DOI: 10.1007/978-3-319-92991-0_3 (cit. on pp. 78, 154).

[KNN19] B. König, M. Nederkorn, and D. Nolte. “CoReS: A Tool for Computing Core Graphs via SAT/SMT Solvers”. In: *Journal of Logical and Algebraic Methods in Programming* (2019). Submitted.

[KN+18] B. König, D. Nolte, J. Padberg, and A. Rensink. “A Tutorial on Graph Transformation”. In: *Graph Transformation, Specifications, and Nets – In Memory of Hartmut Ehrig.* Ed. by R. Heckel and G. Taentzer. LNCS 10800. Springer, 2018, pp. 1–22. DOI: 10.1007/978-3-319-75396-6_5 (cit. on p. ix).

[KW08] A. Koprowski and J. Waldmann. “Arctic Termination ... Below Zero”. In: *Proceedings of the 19th Conference on Rewriting Techniques and Applications (RTA).* Ed. by A. Voronkov. Vol. 5117. LNCS. Springer, 2008, pp. 202–216 (cit. on p. 39).

[KRW02] B. Kullbach, V. Riediger, and A. Winter. “An Overview of the GXL Graph Exchange Language”. In: *Software Visualization.* Ed. by Stephan Diehl. Vol. 2269. Lecture Notes in Computer Science. Springer, 2002, pp. 324–336 (cit. on p. 132).

[LS05] S. Lack and P. Sobociński. “Adhesive and Quasiadhesive Categories”. In: *RAIRO – Theoretical Informatics and Applications* 39.3 (2005) (cit. on pp. 2, 21, 30, 164, 181).

[LS06] S. Lack and P. Sobociński. “Toposes are adhesive”. In: *International conference on graph transformation, ICGT ’06.* Vol. 4178. Springer, 2006, pp. 184–198 (cit. on p. 21).

[LO14] L. Lambers and F. Orejas. “Tableau-Based Reasoning for Graph Properties”. In: *Proc. of ICGT ’14.* LNCS 8571. Springer, 2014, pp. 17–32 (cit. on p. 127).

[Law70] F.W. Lawvere. “Quantifiers and sheaves”. In: *Actes du Congrès International des Mathématiciene.* 1970, pp. 329–334 (cit. on p. 108).

[LRC15] H. Li, X. Rival, and B. E. Chang. “Shape Analysis for Unstructured Sharing”. In: *Proc. of SAS ’15.* LNCS 9291. Springer, 2015, pp. 90–108 (cit. on p. 105).

[Löw93] M. Löwe. “Algebraic approach to single-pushout graph transformation”. In: *Theoretical Computer Science* 109 (1993), pp. 181–224 (cit. on pp. 30, 31).

[Löw10] M. Löwe. “Graph Rewriting in Span-Categories”. In: *Proc. of ICGT ’10.* LNCS 6372. Springer, 2010, pp. 218–233 (cit. on pp. 105, 113, 178).

[Mac71] S. Mac Lane. *Categories for the Working Mathematician.* Graduate Texts in Mathematics, Vol. 5. New York: Springer-Verlag, 1971, pp. ix+262 (cit. on p. 21).

[MM94] S. Mac Lane and I. Moerdijk. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory.* Universitext. Springer New York, 1994 (cit. on pp. 109, 110).

[Mar94] M. Marchiori. “Logic programs as term rewriting systems”. In: *Algebraic and Logic Programming.* Ed. by Giorgio Levi and Mario Rodríguez-Artalejo. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 223–241 (cit. on p. 55).

[NT00] J. Nešetřil and C. Tardif. “Duality Theorems for Finite Structures (Characterising Gaps and Good Characterisations)”. In: *Journal of Combinatorial Theory, Series B* 80 (2000), pp. 80–97 (cit. on pp. 77, 78, 81, 164).

[Nol17] D. Nolte. “Analysis and Abstraction of Graph Transformation Systems via Type Graphs”. In: *STAF 2017 Doctoral Symposium.* Vol. 1955. CEUR Workshop Proceedings. 2017.

[OHe07] P. W. O'Hearn. "Resources, Concurrency and Local Reasoning". In: *Theoretical Computer Science* 375.1–3 (May 2007). Reynolds Festschrift, pp. 271–307 (cit. on p. 102).

[OHe12] P. W. O'Hearn. "A Primer on Separation Logic (and Automatic Program Verification and Analysis)". In: *Software Safety and Security: Tools for Analysis and Verification.* Vol. 33. NATO Science for Peace and Security Series. 2012, pp. 286–318 (cit. on p. 105).

[OEP08] F. Orejas, H. Ehrig, and U. Prange. "A Logic of Graph Constraints". In: *Proc. of FASE '08.* LNCS 4961. Springer, 2008, pp. 179–198 (cit. on p. 81).

[Pen09] K.-H. Pennemann. "Development of Correct Graph Transformation Systems". PhD thesis. Universität Oldenburg, May 2009 (cit. on p. 102).

[Pie88] B. C. Pierce. *A Taste of Category Theory for Computer Scientists.* CMU-CS. Carnegie Mellon University, Computer Science Department, 1988 (cit. on p. 21).

[PE93] M. J. Plasmeijer and M. C. J. D. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, 1993 (cit. on p. 58).

[Plu95] D. Plump. "On termination of graph rewriting". In: *Graph-Theoretic Concepts in Computer Science.* LNCS 1017. Springer, 1995, pp. 88–100 (cit. on p. 72).

[Plu98] D. Plump. "Termination of graph rewriting is undecidable". In: *Fundamenta Informaticae* 33.2 (1998), pp. 201–209 (cit. on pp. 4, 42).

[Plu99] D. Plump. "Term Graph Rewriting". In: *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools.* Ed. by G. Rozenberg. World Scientific, 1999 (cit. on p. 58).

[Plu18] D. Plump. "Modular Termination of Graph Transformation". In: *Graph Transformation, Specifications, and Nets - In Memory of Hartmut Ehrig.* Ed. by Reiko Heckel and Gabriele Taentzer. Vol. 10800. Lecture Notes in Computer Science. Springer, 2018, pp. 231–244 (cit. on p. 72).

[Ren03] A. Rensink. "The GROOVE Simulator: A Tool for State Space Generation". In: *Applications of Graph Transformations with Industrial Relevance.* Ed. by John L. Pfaltz, Manfred Nagl, and Boris Böhlen. Vol. 3062. Lecture Notes in Computer Science. Springer, 2003, pp. 479–485 (cit. on p. 138).

[Ren04a] A. Rensink. "Canonical Graph Shapes". In: *Proc. of ESOP '04.* LNCS 2986. Springer, 2004, pp. 401–415 (cit. on pp. 33, 75, 85, 92, 105, 127, 139).

[Ren04b] A. Rensink. "Representing First-Order Logic using Graphs". In: *Proc. of ICGT '04.* LNCS 3256. Springer, 2004, pp. 319–335 (cit. on pp. 75, 102).

[RZ10] A. Rensink and E. Zambon. “Neighbourhood Abstraction in GROOVE”. In: *Proc. of GraBaTs ’10 (Workshop on Graph-Based Tools)*. Vol. 32. Electronic Communications of the EASST. 2010 (cit. on pp. 105, 127).

[RS86] N. Robertson and P. D Seymour. “Graph minors. II. Algorithmic aspects of tree-width”. In: *Journal of Algorithms* 7.3 (1986), pp. 309–322 (cit. on p. 94).

[RS83] N. Robertson and P. D. Seymour. “Graph minors. I. Excluding a forest”. In: *Journal of Combinatorial Theory, Series B* 35.1 (1983), pp. 39–61 (cit. on p. 94).

[Roz97] G. Rozenberg, ed. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations*. World Scientific, 1997 (cit. on pp. 23, 31, 33, 105).

[RR+09] A. Rutle, A. Rossini, Y. Lamo, and U. Wolter. “A Diagrammatic Formalisation of MOF-Based Modelling Languages”. In: *Proc. of TOOLS EUROPE ’09*. LNBIP 33. Springer, 2009, pp. 37–56 (cit. on p. 102).

[SZ15] D. Sabel and H. Zantema. “Transforming Cycle Rewriting into String Rewriting”. In: *26th International Conference on Rewriting Techniques and Applications (RTA’15)*. Ed. by Maribel Fernandez. Vol. 36. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 285–300 (cit. on pp. 39, 41, 43, 50, 61, 72).

[SRW02] M. Sagiv, T. Reps, and R. Wilhelm. “Parametric Shape Analysis via 3-Valued Logic”. In: *TOPLAS (ACM Transactions on Programming Languages and Systems)* 24.3 (2002), pp. 217–298 (cit. on pp. 5, 33, 101, 102, 105, 127, 139).

[SWW10] D. Steenken, H. Wehrheim, and D. Wonisch. “Towards A Shape Analysis for Graph Transformation Systems”. In: *CoRR* abs/1010.4423 (2010) (cit. on p. 139).

[SWW11] D. Steenken, H. Wehrheim, and D. Wonisch. “Sound and Complete Abstract Graph Transformation”. In: *Proc. of SBMF ’11*. LNCS 7021. Springer, 2011, pp. 92–107 (cit. on pp. 2, 33, 75, 105, 127, 139).

[SW11] D. Steenken and D. Wonisch. “Using shape analysis to verify graph transformations in model driven design”. In: Aug. 2011, pp. 457–462 (cit. on p. 139).

[Ter03] Terese. *Term Rewriting Systems*. Vol. 55. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003 (cit. on pp. 39, 55).

[Toy87] Y. Toyama. “Counterexamples to Termination for the Direct Sum of Term Rewriting Systems”. In: *Information Processing Letters* 25 (1987), pp. 141–143 (cit. on p. 63).

[VV+06] D. Varró, S. Varró-Gyapay, H. Ehrig, U. Prange, and G. Taentzer. “Termination Analysis of Model Transformations by Petri Nets”. In: *Proc. of ICGT 2006*. Vol. 4178. LNCS. Springer, 2006 (cit. on p. 71).

[Win02] A. Winter. “Exchanging Graphs with GXL”. In: *Graph Drawing*. Ed. by Petra Mutzel, Michael Jünger, and Sebastian Leipert. Vol. 2265. Lecture Notes in Computer Science. Springer, 2002, pp. 485–500 (cit. on p. 132).

[Zan95] H. Zantema. “Termination of Term Rewriting by Semantic Labelling”. In: *Fundamenta Informaticae* 24.1/2 (1995), pp. 89–105 (cit. on p. 39).

[Zan03] H. Zantema. “Termination”. In: *Term Rewriting Systems*. Ed. by Marc Bezem, Jan Willem Klop, and Roel de Vrijer. Cambridge University Press, 2003. Chap. 6 (cit. on p. 48).

[Zan08] H. Zantema. “Normalization of Infinite Terms”. In: *Proceedings of the 19th Conference on Rewriting Techniques and Applications (RTA)*. Ed. by A. Voronkov. Vol. 5117. LNCS. Springer, 2008, pp. 441–455 (cit. on p. 72).

[ZBK14] H. Zantema, H. J. S. Bruggink, and B. König. “Termination of cycle rewriting”. In: *Proceedings of Joint International Conference, RTA-TLCA 2014*. Ed. by G. Dowek. Vol. 8560. LNCS. Springer, 2014, pp. 476–490 (cit. on pp. 39, 61, 63, 72).

[ZNK16] H. Zantema, D. Nolte, and B. König. “Termination of Term Graph Rewriting”. In: *Proc. of WST '16 (Workshop on Termination)*. 2016 (cit. on p. ix).

Nomenclature

Category Theory Symbols

$\mathbf{C}$	An arbitrary category, page 16
$\mathbf{C} \downarrow A$	Slice category of a category $\mathbf{C}$ over an object A, page 109
$\mathbf{Set}$	Category of sets and functions, page 16
$\mathbf{Rel}$	Category of sets and relations, page 16
$\mathbf{Mon}$	Category of ordered monoids and monoid morphisms, page 85
$\mathbf{Mon}^{-}$	Category of ordered monoids with subtraction, page 117
$\mathbf{Mat}_{\varphi}$	Materialization category over a mono φ, page 112
$\mathbf{Mat}_{\varphi}^{\varphi_L}$	Materialization subcategory of rewritable objects, page 114
$\mathbf{Graph}_{\Lambda}$	Category of Λ-labeled graphs and graph morphisms, page 30
$\lvert\mathbf{Graph}_{\Lambda}\rvert$	Set of objects from $\mathbf{Graph}_{\Lambda}$, page 30
$Cospan_m(\mathbf{Graph}_{\Lambda})$	Category of discrete graphs and cospans, page 94
$\mathcal{O}$	A class $\mathcal{O}$ of objects, page 16
$\mathcal{M}$	A class $\mathcal{M}$ of arrows (or morphisms), page 16
id	A class id of identity arrows, page 16
$dom(f)$	Domain of an arrow f, page 16
$cod(f)$	Codomoain of a arrow f, page 16
$f \circ g$	Arrow composition, page 16
$A \rightarrowtail B$	A monomorphism from A to B, page 17
$A \twoheadrightarrow B$	An epimorphism from A to B, page 17
$A \xrightarrow{\sim} B$	An isomorphism between A and B, page 17
$A \rightharpoonup B$	A partial map from A to B, page 108
$A \dot{\rightarrow} B$	A family of arrows from A to B, page 108
$A \dashrightarrow B$	Denoting existence of an arrow from A to B, page 111
$A \times B$	Product object of two objects A and B, page 17
$A \oplus B$	Coproduct object of two objects A and B, page 18
$\mathbf{1}$	Terminal object in a category, page 18
$\mathbf{0}$	Initial object in a category, page 18

$!_A$	Unique morphism from an object A to $\mathbf{1}$, page 18
$?_A$	Unique morphism from $\mathbf{0}$ to an object A, page 18
Ω	Truth value object in a category, page 107
$\mathtt{true}$	A subobject classifier, page 107
$\mathtt{true}_A$	A subobject classifier in a slice category over A, page 109
$\mathcal{F}\colon \mathbf{C} \to \mathbf{D}$	A functor $\mathcal{F}$ from a category $\mathbf{C}$ to a category $\mathbf{D}$, page 18
$Id_{\mathbf{C}}$	An identity functor from $\mathbf{C}$ to $\mathbf{C}$, page 18
$\mathcal{C}$	An automaton functor $\mathcal{C}$, page 96
$\mathcal{C}_{T[M]}$	A counting cospan automaton functor $\mathcal{C}_{T[M]}$, page 97
η	A natural transformation between functors, page 108
η_A	A component of a natural transformation η on A, page 108
$B \xleftarrow{f} A \xrightarrow{g} C$	A span, i.e. pair of arrows $f\colon A \to B, g\colon A \to C$, page 19
$B \xrightarrow{g} D \xleftarrow{f} C$	A cospan, i.e. pair of arrows $f\colon C \to D, g\colon B \to D$, page 20
$p\colon L \leftarrow I \to R$	A production (or rule) p in a category, page 21
$X \overset{p,m}{\Longrightarrow} Y$	Rewriting step from X to Y via rule p at match m, page 21
$X \overset{p,m}{\Longrightarrow}$	Denoting that X can be rewritten via p at m, page 115
$\rightsquigarrow$	Abstract rewriting relation, page 122
$\hookrightarrow$	Modified abstract rewriting relation, page 123

Graph Symbols

Λ	Set of edge labels, page 24
V_G	Finite set of nodes of a graph G, page 24
E_G	Finite set of edges of a graph G, page 24
src_G	Source function mapping an edge to its source node, page 24
tgt_G	Target function mapping an edge to its target node, page 24
lab_G	Labeling function for the edges of G, page 24
$\varnothing$	The empty graph, page 24
φ	Graph morphism with components φ_V and φ_E, page 24
$\varphi \circ \psi$	Graph morphism composition $[(\varphi{\circ}\psi)(x) = \varphi(\psi(x))]$, page 24
$G \cong H$	Denoting that the graphs G, H are isomorphic, page 24
$G \ncong H$	Denoting that the graphs G, H are not isomorphic, page 24

$G +_I H$	Gluing of G and H over a common interface I, page 26
$\to$	Homomorphism preorder, page 30
$G \to H$	Denoting existence of a morphism from G to H, page 30
$G \not\to H$	Denoting non-existence of a morphism from G to H, page 30
$G \sim H$	Homomorphically equivalent graphs G and H, page 30
$\mathit{core}(G)$	A core graph of G, page 77
(R, T)	A duality pair, page 81
T_G	Tree decomposition of a graph G, page 94
P_G	Path decomposition of a graph G, page 94
$w(\mathsf{T}_G)$	Width of a tree decomposition $\mathcal{T}_G$, page 94
$tw(G)$	Treewidth of a graph G, page 94
$pw(G)$	Pathwidth of a graph G, page 94
$c\colon J \circ\!\!\rightarrow K$	Short cospan representation of $c\colon J \to G \leftarrow K$, page 95
$c_1 ; c_2$	Composition of cospans via pushouts, page 95

Graph Transformation Symbols

ρ	A graph transformation rule, page 27
$L \leftarrow\!\varphi_L\!- I -\!\varphi_R\!\rightarrow R$	A double-pushout graph transformation rule, page 27
$G \Rightarrow_\rho H$	Graph transformation step from G to H via rule ρ, page 27
C	The context graph C in a DPO rewriting step, page 27
$\mathcal{R}$	A graph transformation system, i.e., a set of rules ρ, page 29

Termination Analysis Symbols

w_T	Weight function of a type graph T, page 45
✻_T	Flower node of type graph T, page 45
$\mathit{fl}_T(G)$	Flower morphism from graph G to type graph T, page 45
$w_t(\varphi)$	Weight of a morphism φ wrt. a morphism t, page 47
$R^{<}$	Set of decreasing rules, page 48
$R^{=}$	Set of non-increasing rules, page 48
$\mathcal{X}$	Countable set of variables, page 56
$\mathcal{F}$	Signature, page 56
$\mathcal{T}(\mathcal{F}, \mathcal{X})$	Set of terms over the signature $\mathcal{F}$, page 56

$\mathcal{V}\text{ar}(t)$	Set of variables occuring in a term t, page 56
$C[t]$	Context around a subterm t, page 56
$\square$	Empty space symbol, page 56
σ	A substitution of variables to terms, page 57
$\bar{\sigma}$	Homomorphically extended substitution, page 57
$\mathcal{R}$	A term rewriting system, page 58
$\mathcal{R}^b$	A term graph rewriting system (basic version), page 62
$\mathcal{R}^e$	A term graph rewriting system (extended version), page 62
$\rightarrow_{\mathcal{R}}$	Rewrite relation of term rewriting system $\mathcal{R}$, page 58
$\mathsf{TG}(t)$	Term graph of a term t, page 59
$p = L \leftarrow_{\ell} I \rightarrow_{r} R$	A term graph production, page 60
$\mathsf{SN}(\mathcal{R})$	Strong normalization of term rewriting system $\mathcal{R}$, page 62
$\mathsf{SN}^b(\mathcal{R})$	Strong normalization of $\mathcal{R}^b$, page 62
$\mathsf{SN}^e(\mathcal{R})$	Strong normalization of $\mathcal{R}^e$, page 62

Graph Language and Object Language Symbols

T	A type graph, page 33
$T^{\Lambda}_{\ast}$	The flower graph over the label set Λ, page 34
$\mathcal{L}(T)$	A type graph language specified by T, page 33
$\mathcal{L}(T[a])$	Type graph language by an annotated graph $T[a]$, page 88
$\mathcal{L}(T[M])$	Language by a multiply annotated graph $T[M]$, page 89
$\mathcal{L}(T[M])^{\leq k}$	k-bounded language of $\mathcal{L}(T[M])$, page 94
R	A restriction graph, page 78
$\mathcal{L}_{\mathtt{R}}(R)$	A restriction graph language specified by R, page 78
$\mathcal{L}(T_1) \subseteq \mathcal{L}(T_2)$	Language inclusion of type graph languages, page 79
$\mathcal{L}(T_1) \cup \mathcal{L}(T_2)$	Union of type graph languages, page 82
$\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$	Intersection of type graph languages, page 82
$\mathcal{L}_{\mathcal{C}}$	Language of an automaton functor $\mathcal{C}$, page 96
$\mathcal{L}_{\mathcal{C}_{T[M]}}$	Language of counting cospan automaton functor, page 97
$\mathcal{L}(A)$	Object language of an abstract object A, page 111
$\mathcal{L}(\varphi)$	Mono language of a monomorphism φ, page 111

Materialization Symbols

$L \rightarrowtail X \to A$	Factorization in the materialization category, page 112
$L \rightarrowtail \langle\varphi\rangle \to A$	Materialization of $\varphi\colon L \to A$, page 112
$\langle\varphi\rangle$	Short representation of $L \rightarrowtail \langle\varphi\rangle \to A$, page 112
$\langle\!\langle\varphi, \varphi_L\rangle\!\rangle$	Rewritable materialization of $\varphi\colon L \to A, \varphi_L\colon I \to L$, page 114

Annotation Symbols

$\mathcal{A}$	Generic annotation functor, page 86
$\mathcal{A}_\varphi$	Alternative representation denoting $\mathcal{A}(\varphi)$, page 86
s_G	A standard annotation for a graph G, page 86
$\mathcal{M}_n$	Natural number monoid (up to n), page 86
$\mathcal{B}^n$	Multiplicities over ordered monoid $\mathcal{M}_n$, page 87
$\mathcal{S}^n$	Node out-degree annotation over ordered monoid $\mathcal{M}_n$, page 118
$\mathcal{P}_G$	Path monoid of the graph G, page 86
$\mathcal{T}$	Path annotation functor over ordered monoid $\mathcal{P}_G$, page 87
$G[M]$	A multiply annotated graph G, page 89

Logical Symbols

$\wedge$	Conjunction, page 13
$\vee$	Disjunction, page 13
$\Longrightarrow$	Implication, page 13
$\Longleftrightarrow$	Bi-implication, page 13
$\exists$	Existential quantification, page 13
$\forall$	Universal quantification, page 13

Other Symbols

$\in, \notin$	Membership relation and its negation, page 14
$=, \neq$	Equality relation and its negation, page 14
$\subseteq, \nsubseteq$	Subset relation and its negation, page 14
$\subset, \not\subset$	Strict subset relation and its negation, page 14
$\emptyset$	The empty set, page 14
$\mathcal{P}(X)$	The powerset of a set X, page 14
$\mathbb{N}_0$	Set of natural numbers with zero, i.e. $\{0, 1, 2, \ldots\}$, page 14

$\mathbb{N}$	Set of natural numbers without zero, i.e. $\{1, 2, \ldots\}$, page 14
$X \setminus Y$	Relative complement, i.e. $\{x \in X \mid x \notin Y\}$, page 14
$X \cup Y$	Union, i.e. $\{z \mid z \in X \vee z \in Y\}$, page 14
$X \cap Y$	Intersection, i.e. $\{z \mid z \in X \wedge z \in Y\}$, page 14
$X \times Y$	Cartesian product, i.e. $\{(x, y) \mid x \in X \wedge y \in Y\}$, page 14
X^n	n-ary cartesian product, i.e. n-times $X \times \ldots \times X$, page 14
X^Y	Exponentiation, i.e. $\{f \colon Y \to X\}$, page 86
$X_1 \biguplus X_2$	Disjoint union, i.e. $\bigcup\limits_{i \in I} \{(x, i) \mid x \in X_i\}$, page 14
R, R^{-1}	Binary relation R and its inverse R^{-1}, page 14
R^+	Transitive closure R^+ of a relation R, page 14
$\leq$	Order relation, page 14
$<$	Strict subrelation of $\leq$, page 14
$\equiv$	Equivalence relation, page 14
$X/\equiv$	Quotient set of all equivalence classes of X by $\equiv$, page 14
$[x]_\equiv$	Equivalence class of an element x w.r.t. $\equiv$, page 14
$f \colon X \to Y$	Function with domain X and codomain Y, page 14
$f(x)$	Image of the input value x, page 14
$f\vert_Z$	Function f with a domain restriction to a set Z, page 15
$\bigvee X$	Least upper bound (or join, or supremum) of X, page 15
$\bigwedge X$	Greatest lower bound (or meet, or infimum) of X, page 15
$\top$	Maximal element (also called top element), page 15
$\bot$	Minimal element (also called bottom element), page 15
$\oplus$	Abstract addition-operator, page 43
$\otimes$	Abstract multiplication-operator, page 43
Σ	Generalised sum of $\oplus$, page 44
$\prod$	Generalised product of $\otimes$, page 44

Index