Faculty of Electrical Engineering, Mathematics and Computer Science,
Formal Methods and Tools,
University of Twente

Master's Thesis

Type Inference for Graph Transformation Systems

by
Frank J. van Es

July, 2008

Abstract

Graphs have proved to be a powerful formalism to represent structures in models and meta-models of software systems. In this context, dynamic changes to graphs are described by rule-based graph transformations and the use of type graphs to classify graph structures has emerged as a valuable principle.
While most studies towards the integration of type graphs in graph transformation systems rely on the existence of manually created type graphs, this project aims at automatically computing these type graphs for graph tranformation systems. As a logical extension, we also consider type graphs with inheritance and verify our findings with an implementation of a type inference algorithm into the GROOVE tool set.
The main results of this study show that automatic type graph reconstruction in graph transformation systems is possible and offers valuable insights to graph typings.

Preface

With this thesis I conclude my final project, which has been my main occupation for the past nine months. This thesis is also the end of my study at the University of Twente, which I started almost seven years ago. In these seven years I have not only learnt a lot, I also consider my student time as a joyful period of my life.

Acknowledgements

The first person I would like to thank is my main supervisor, Arend Rensink. I thank him for guiding me through this project and for keeping me on the right track when I tended to get lost. He helped me to stay motivated during the whole project, particularly when I had the tendency to lose faith in making some of the proofs.

Next, I would not have improved as much as I did in writing proofs without the support given by my second supervisor, Iovka Boneva. I would like to thank her for giving me all the help I needed during this project.

I want to thank my third supervisor, Rom Langerak, for his great involvement during my project. The questions he asked during the earliest meetings of this project helped me to better understand the theories myself; his enthusiasm for my project and for formal methods in general have been very encouraging for me.

During this project, my friends in the project room were very important for me. Because of them I can back at the period in which I conducted my graduation project as a time full of fun and laughter. I would like to thank Alfons, David, Erwin, Jan-Willem K, Jan-Willem V, Jorge, Mark, Matthijs, Niek, Paul, Riemer, Steven, Viet-Yen, and Wouter for the great time we had together in the project room.

Finally, I also want to thank my family and close friends for mentally supporting me during this project.

Frank van Es
Enschede, July 2008

Table of Contents

CHAPTER 1

Introduction

In recent years, software systems have become more and more complex; therefore modelling and abstraction play an increasingly important role in current software development. Although modelling languages such as the Unified Modelling Language (UML) are common heritage and widely used in the design of many systems, implementation of those systems generally is still performed using traditional programming languages, which are unaware of the model. Techniques such as object-oriented programming and aspect-oriented programming are useful for implementing systems in a modular and clarifying way that approaches the model, but the main drawback of this approach is that there is no synchronisation between the models and the implementation, a fact which may lead to inconsistencies between the specification and the implementation of systems when these systems evolve.

In order to overcome this problem, model-based software development, in which static structure is expressed in models and where model transformations describe dynamic changes to these models, has been developed. Models are not merely used for documentation using this approach, but fully describe the behaviour and functionality of programs and act as a direct input for code generators.

In this thesis we use graphs and graph transformations to describe models and dynamic changes to these models respectively. Graph transformations have been succesfully applied to modelling, meta-modelling, and model-driven architecture [EEPT06] and offer a simple and intuitive, but at the same time rigorous formal, way to describe manipulations to models [Kön05]. An additional advantage of using graph transformations is that they describe dynamic behaviour of object-oriented systems through rules working directly on the models, rather than using some intermediate modelling language [Ren08].

An important property of many frameworks concerned with model transformation, such as the Model Driven Architecture (MDA) [MM03], is the use of meta-models to which the models being transformed must conform. Analogously, research concerning graph transformations has evolved towards the use of type graphs, which are used to classify graph nodes and edges [BEdLT04, dLBE$^+$07].

The basic idea of using type graphs in combination with graph transformations is that type graphs describe and constrain the static structure of the instance graphs that are being transformed, i.e. they are used for correctness and documentation of graph grammars.

Although the integration of type graphs into graph grammars has been thoroughly investigated (see e.g. [BEdLT04], [dLBE$^+$07], or [Kön05]), most studies on this subject so far have always assumed the existence of a predefined type graph. In this thesis, however, we take an alternative approach: we aim at automatically computing type graphs from existing graph grammars.

Because a type graph constrains the structure of the graphs that are being transformed, a basic requirement for the type graphs that we automatically compute is that they impose no constraints on the graphs and transformations of the grammar given. Concretely, we aim at answering the following question during this work:

> **Problem statement:** *Develop a method to automatically compute a type graph from a given graph grammar, such that it imposes no restriction on the derivations of this graph grammar.*

This defines the main challenge of this project; however, in order to get a more complete research project, we also ask ourselves the following sub-questions:

1. How does type inheritance fit within this method?
2. How can the theory we gathered during this project be applied in the GROOVE tool set?

In the remainder of this thesis we will answer these questions. The results regarding these questions will be presented in Chapters 3, 4, and 5 respectively.

1.1 Motivation

Most publications on type graphs in combination with graph grammars—often referred to as typed graph grammars—are based on manually created type graphs. On the contrary, existing type inference algorithms such as the Hindley-Milner algorithm [Hin69, Mil78] have proved to be powerful in their application, particularly in functional programming languages.

We think that automatic type reconstruction is also possible in graph transformation systems; moreover, we think that it might be a powerful extension to the current theories on typed graph grammars. Not only does it save time when implemented, it may also lead to important new insights with respect to graph typings.

As an example to this, note that existing theories on type graphs define type graphs just as graphs, possibly extended with notions such as inheritance edges and multiplicities, that constrain their instance graphs. In this study on type inference for graph transformation systems, however, we also define constraints on the type graphs themselves.

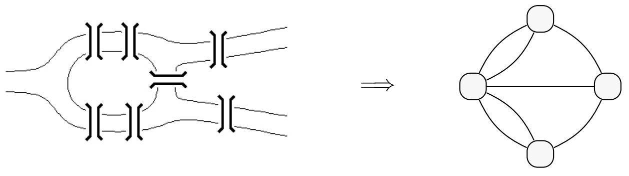


Figure 1.1: The *"Seven bridges of Königsberg"* problem; on the left displayed in a schematic way, on the right modelled as a graph. Nodes in this graph represent the islands of Königsberg; the edges denote the bridges between these islands.

1.2 Background

This section gives an intuitive introduction to the concepts used in this thesis and is used to offer some basic understanding of these topics to the readers. More detailed and particularly more formal descriptions of the concepts are given in later chapters.

1.2.1 Graphs and Graph Transformations

Graphs are mathematical structures consisting of a set of nodes and a set of edges that connect these nodes to each other. Because graphs have an intuitive graphical representation, where nodes are represented by boxes and edges as lines between these nodes, they can be used to model virtually any possible structure.

Example 1.1. *A well-known application of graphs to solve problems is the prlblem of the seven bridges of Königsberg, that was published in 1736 by the Swiss mathematician Leonhard Euler. This problem, which is illustrated in Figure 1.1, states that it is impossible to cross all seven bridges that are illustrated in this figure without crossing at least one bridge more than once, and resulted in some of the fundamental concepts of graph theory [Gri98].*

Nowadays graphs are used in many applications, including computer networks, wireless/mobile networks, and car navigation systems; in this thesis, however, the primary use for graphs involves modelling and meta-modelling of software systems, where graphs are used to define the static structure of such systems.

Dynamic changes of these static structures, on the other hand, are described using graph manipulations. In this thesis, we define such manipulations on graphs using graph transformations, which provide a formal yet intuitive way to specify the manipulation of graphs based on rules [CMR$^+$97].

The rich theory of graph transformations has developed in the past 30 years [dLBE$^+$07]; standard works on the subject include [Roz97] and [EEPT06]. Graph transformations have been subject of many international conferences (e.g. ICGT) and workshops (e.g. GT-VMT, AGTIVE, SETRA, TERMGRAPH) and have been succesfully applied to many areas in software engineering, such as model and program transformation, visual programming, (meta-) modelling,

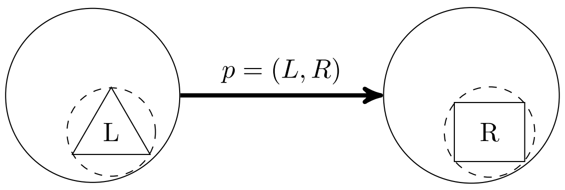


Figure 1.2: Schematic overview of a graph transformation (source: [EEPT06]).

and model-driven architecture [EEPT06]; this clearly indicates that the topic is thoroughly investigated in the last decades and still is an active area of research.

The main idea of graph transformations is rule-based manipulation of graphs [EEPT06]. Graphs are manipulated using so-called production rules, $p = (L, R)$, which are pairs of graphs consisting of a left hand side L and a right hand side R. These production rules describe how certain graphs (viz. those to which the rule is applicable) are transformed into others [Ren05].

Application of a production rule is illustrated in Figure 1.2. Applying a rule $p = (L, R)$ to a graph G means finding an occurrence of L and replacing it by R, leading to a resulting graph H. These graphs G and H are often called the *host* graph and *product* graph of a graph transformation, respectively.

This describes the general structure of a graph transformation from a host graph G to a product graph H, usually written $G \Rightarrow H$. There exist several different approaches to graph transformation; these approaches describe how exactly the ocurrence of L is deleted and how R is connected to the resulting graph afterwards. For a list of different approaches, see e.g. [Roz97] or [EEPT06].

The approach we use in this thesis is the algebraic approach, specifically the single-pushout variant of this approach. This approach is based on relations between graphs that are called *graph morphisms* and on so-called *pushout constructions*, and is introduced by Löwe [Löw93].

Example 1.2. *Figure 1.3 illustrates a simple example of a graph transformation in the single pushout approach. Graphs L and R in this figure represent the graphs of a production rule p and graphs G and H represent the host graph and product graph of the application of p on G, respectively. The numbering of the nodes indicates how the nodes of the graphs are mapped by graph morphisms; the edge mappings are not explicitly shown but follow uniquely from the node mappings.*

1.2.2 Type Systems

The theory of types originates from the early 1900s when Bertrand Russell wrote his Principles of Mathematics, containing what is currently well known as *Russel's paradox.* This paradox states that a set that contains all sets that do not contain themselves, should and should not contain itself at the same time and therefore can never exist; a solution to this problem constituted the first theory on types.

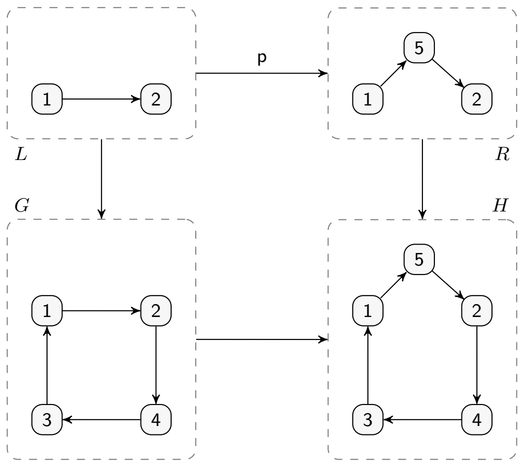


Figure 1.3: Example of a single pushout graph transformation.

Many definition for type systems have been introduced throughout history. Russell, for instance, defined a type as *"a range of significance for certain propositional functions"* [Rus03], and Constable et al as *"a collection of objects having similar structure"* [CSB$^+$86]. In modern software engineering, however, type systems are one of numerous *formal methods* that help software engineers to enforce systems to behave correctly according to their specifications [Pie02]. The main purpose of type systems is to prevent the occurrence of certain errors during program execution [Car04].

Example 1.3. *Consider the following Java method, which simply returs the length of a given string:*

```
public static int strLen(String s) {
    return s.length();
}
```

Now suppose that this method is called as follows:

```
strLen(new Boolean(true));
```

Then, the type checker raises an error, because type `String` *and* `Boolean` *are incompatible.*

This illustrates a simple example in which type checking allows early detection of an error. In larger examples type checking is even more helpful because then the mistake might be less ovbious.

Thus type systems are useful for the early detection of errors, which is a desirable feature. A second function of type systems is *documentation*: types offer useful help for reading programs [Pie02]. This is easily shown in the following code fragment, which contains the interface of the method `replaceAll` in Java class String. Regardless of the implementation of this function, its interface shows that it requires two parameters of type `String` and that it also returns an object of type `String`.

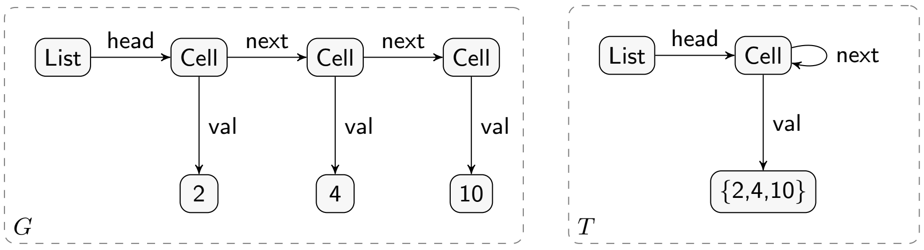


Figure 1.4: An instance graph (left) and the corresponding type graph (right)

```
public String replaceAll(String regex, String replacement);
```

The main advantage of the form of documentation given by types is that it is always up-to-date, since it is checked every time the code is compiled [Pie02].

An important mechanism that is present in most current type systems, particularly for *object-oriented* languages, is *inheritance*, or *subtyping*. The basic idea of inheritance is that new types can be defined using already existing types; this new type is then a *subtype* of the already existing type and this existing type is a *supertype* of the new one.

A subtype is a specialisation of its supertype and is considered more concrete. Reason for this is that the subtype *extends* the more abstract supertype and inherits its functionality.

Inherent to inheritance is what Pierce calls the *principle of safe substitution*. According to this principle, any object of type S can be used safely in contexts where objects of type T are expected if S is a subtype of T [Pie02].

Type Graphs

In the setting of typed graph grammars, type graphs are used to classify nodes and edges of instance graphs, i.e. they put restrictions on the possible nodes and edges that their instances may have. A graph is an instance of a type graph if it has a typing into the type graph, which intuitively means that the type graph contains node types for all nodes of the instance graph and edge types for all edges in the instance graph.

Example 1.4. *Figure 1.4 illustrates an example of a type graph and one of its possible instances. It is easy to see that the type graph contains node types and edge types for all nodes and edges of the instance graph. The exact typing of the instance graph into the type graph will be explained in Example 3.2 on page 20.*

Several extensions to type graphs are proposed throughout the literature. Without doubt the most often used is *inheritance*, which is introduced in graph transformation systems by Bardohl et al. [BEdLT04]; it comprises the use of a special type of edges, called *inheritance* edges, to specify that certain node types are subtypes or supertypes of others. Some papers, however, also introduce a notion of node- and edge multiplicities into type graphs; rather than specifying what nodes and edges may be present in instance graphs, type graphs with multiplicities specify how many of these nodes and edges may be present. Although we will use type inheritance in this thesis (see Chapter 4), we will not elaborate on multiplicities. For more information on multiplicities in type graphs see, e.g. [EKTW06] or [TR05].

Type Inference

Most programming languages depend on type systems that use explicit type annotations [Pie02]. This means that all variables in these programming languages must be explicitly annotated with their type. For instance, consider the following Java code fragment:

```
public int aMethod(int a) {
    this.a = a;
}
```

In this example, the input and output variables of function `aMethod` need to be explicitly annotated with their types (**`int`** and **`int`**).

A technique present in some programming languages that makes explicit type annotations unnecessary is called *type inference*. A definition that gives a good idea of what type inference is, is given by Mitchell: he defines type inference as “the process of finding types for untyped expressions” [Mit84]. In line with this definition, type inference algorithms are capable of calculating types for variables that are not explicitly annotated with type information.

A well-known type inference algorithms is the *Hindley-Milner* algorithm [Hin69, Mil78], which is used by many functional programming languages such as *Haskell* or *Miranda*. To illustrate the functioning of the algorithm, consider the following Miranda implementation of a function `length`, which computes the length of a given list:

```
length []     = 0
length (x:xs) = 1 + length xs
```

Although the two parameters of the function are not explicitly annotated with their types, the type inference algorithm is capable to compute a type for the function (which is `[A] -> num`).

1.2.3 Groove

During this project, we will implement a type inference algorithm into GROOVE, which stands for GRaphs for Object-Oriented VErification [RBKS]. GROOVE is a graph transformation based model checker created at the University of Twente.

GROOVE is a tool that uses graphs and graph transformations to facilitate the modelling and verification of object-oriented systems and model transformations. Although GROOVE offers many features with respect to automated verification, such as model checking, we are particularly interested in the design of object-oriented systems where types play an important role.

Currently, Groove does not support typed graph grammars. We believe that type graphs give—next to a strong verification mechanism for graphs—many insights to complex graph problems and therefore we trust that the addition of a type inheritance algorithm to Groove will lead to many opportunities for the future development of the tool and associated research.

1.3 Organisation of this Thesis

We start by introducing the formalisms used throughout this thesis in Chapter 2. The chapter starts with a brief introduction into elementary mathematics and thereafter introduces graphs, graph morphisms, and commutative diagrams; these concepts form the basis of the algebraic approach to graph transformations, which is used throughout this thesis.

Chapter 3 introduces typings and type graphs in graph grammars. After discussing what properties a type graph for a graph grammar should have and what it means for type graphs to be stronger than others, we define a *perfect* type graph for a grammar. We shall see that this perfect type graph can not be computed algorithmically; therefore we introduce two algorithms that compute approximations of this perfect type graph. First, we define a *naive* algorithm, which we thereafter refine; this results in an *improved* algorithm.

Chapter 4 adds the notion of node type inheritance to type graphs. It defines a *naive* type graph with inheritance for a given graph grammar, that contains an over-approximation of inheritance edges and describes an algorithm that is defined in a similar way as this naive type graph with inheritance. The chapter concludes with some contraction scenarios, which are used to decrease the number of unnecessary inheritance edges in the naive type graph.

In order to verify the theory of the preceding chapters, Chapter 5 describes an implementation of the two algorithms we defined in Chapter 3 in Groove. By means of two examples we show that the algorithms compute valid type graphs for given graph grammars and are useful in practice.

Finally, Chapter 6 summarises this work and discusses some possible directions for future work.

CHAPTER 2

Preliminaries

This chapter is an introduction to the theory used in this thesis. Many concepts in this thesis rely on concepts and notations concerned with elementaty mathematics, graph theory, and algebraic graph transformations. Although many of these concepts may be familiar to the reader, we will give a brief overview of those that we consider most important.

In Section 2.1 we introduce relations, functions, and some special relations such as partial orders and equivalence relations, which form the basis for most concepts in this thesis. Thereafter, in Section 2.2, we use many of these concepts in combination with graphs; this section also introduces graph morphisms, which describe relations between graphs and form a fundamental basis for graph transformations. Section 2.3 subsequently introduces the concept of diagrams, which form the underlying concept of graph transformations, which are introduced in Section 2.4.

2.1 Basic Concepts

In this section we shall review some concepts and notations with respect to elementary mathematics. We assume that the user is familiar with elementary set theory and shall therefore not elaborate on this subject. Instead of this, we will introduce relations and functions first, after which we list some common binary relations—such as preorders, partial orders, and equivalence relations—and define closure operations on binary relations.

2.1.1 Relations and Functions

Definition 2.1 (Relation)**.** *A* relation *on a tuple of sets $S_1, S_2, \ldots S_n$ is a subset of the* cartesian product *of these sets, defined as*

$$S_1 \times S_2 \times \ldots \times S_n = \{(s_1, s_2, \ldots, s_n) \mid s_i \in S_i,\ \textit{for } i = 1, 2, \ldots, n\}.$$

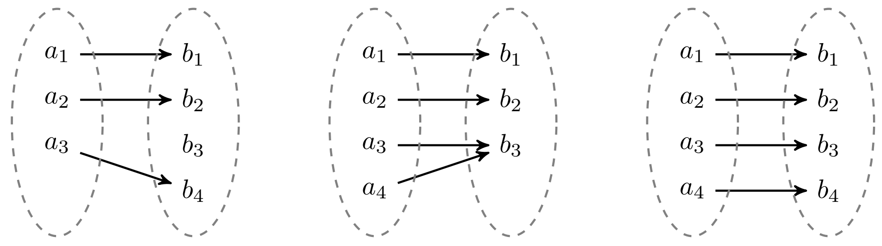
A *binary relation* is a relation on two sets; in this thesis we shall mostly use binary relations. Elements of a binary relation on two sets A and B are ordered pairs (a, b) such that $a \in A$ and $b \in B$. Instead of this notation, we will often use $a\ R\ b$ to denote that (a, b) is in binary relation R.

A special kind of relation is a *function*. A function is a relation for which it holds that for all x there is at most one y such that (x, y) is in the relation.

Definition 2.2 (Function)**.** *A function $f : A \to B$ is a relation on sets A and B such that if $(a, b_1) \in f$ and $(a, b_2) \in f$, then $b_1 = b_2$.*

1. *The* domain *of f, denoted* $\operatorname{dom} f$, *is the set of elements $a \in A$ such that $(a, b) \in f$ for some b. The* image *of f, denoted* $\operatorname{im} f$, *is the set of elements $b \in B$ such that $(a, b) \in f$ for some a.*
2. *We use the standard notation $f(x) = y$ to denote that $(x, y) \in f$.*
3. *f is* injective *if every element in the image of f is mapped by at most one element from the domain, i.e. $\forall a, b \in A$, $f(a) = f(b)$ implies $a = b$.*
4. *f is* surjective *if every element in the image of the function is mapped by at least one element from the domain, i.e. $\forall b \in B$, $\exists a \in A$ such that $f(a) = b$.*
5. *f is* bijective *if it is both injective and surjective, i.e. $\forall b \in B$ there exists exactly one $a \in A$ such that $f(a) = b$.*
6. *The inverse of f is a function $f^{-1} : B \to \wp(A)$, where $\wp(A)$ is the powerset of A. f^{-1} is defined as $f^{-1}(y) = \{x \mid f(x) = y\}$; the set $f^{-1}(y)$ is called the* preimage *of y.*
7. *f is* total *if* $\operatorname{dom} f = A$.
8. *f is* partial *if* $\operatorname{dom} f \subseteq A$.

Figure 2.1 gives an example of an injective, a surjective, and a bijective function.



(a) Injective, not surjective

(b) Surjective, not injective

(c) Bijective

Figure 2.1: Examples of injective, surjective and bijective functions

A concept often used in this thesis is called function composition; it comprises the application of a function to the result of another.

Definition 2.3 (Function composition)**.** *Let $f : A \to B$ and $g : B \to C$ be two functions. The* composition *of f and g is a function $g \circ f : A \to C$ defined by $(g \circ f)(a) = f(g(a))$.*

Definition 2.4 (Restriction of a function)**.** *Let $f : A \to B$ be a function and let S be a subset of A. Then the restriction of f to S is the function $f|_S : S \to B$ defined as $f|_S(s) = f(s)$ for all $s \in S \cap \operatorname{dom} f$.*

2.1.2 Common Binary Relations

Some common binary relations that we often use in this thesis are preorders, partial orders, and equivalence relations.

Definition 2.5 (Preorder). *Let S be a set and $\lesssim$ be a binary relation on S. Then $\lesssim$ is a preorder on S if the following conditions hold:*

– $\forall a \in S \,.\, a \lesssim a$ *(reflexivity)*
– $\forall a, b, c \in S \,.\, a \lesssim b \wedge b \lesssim c \Rightarrow a \lesssim c$ *(transitivity)*

Definition 2.6 (Kernel of a preorder). *Let S be a set and let $\lesssim$ be a preorder on S. Then the* kernel *of $\lesssim$, denoted* $\ker \lesssim$*, is the largest symmetric subrelation of $\lesssim$ and is defined as follows:*

$$\ker \lesssim \, = \{(a, b) \mid a \lesssim b, b \lesssim a\}$$

Definition 2.7 (Partial and total order). *Let S be a set and $\leq$ be a preorder on S. Then $\leq$ is a* partial order *on S if the following property holds:*

– $\forall a, b \in S : a \leq b \wedge b \leq a \Rightarrow a = b$ *(antisymmetry)*

Furthermore, $\leq$ is a total order *if additionally the following property holds:*

– $\forall a, b \in S : a \leq b \vee b \leq a$ *(totality)*

Equivalence Relations

Equivalence relations are used to denote that some elements in a set are *equivalent* in some way. An often used property of equivalence relations is that they are able to partition their underlying set into their equivalence classes.

Definition 2.8 (Equivalence Relation). *Let S be a set and $\simeq$ be a preorder on S. Then $\simeq$ is an equivalence relation on S if the following property holds:*

– $\forall a, b \in S : a \simeq b \Rightarrow b \simeq a$ *(symmetry)*

Definition 2.9 (Equivalence class). *If $\simeq$ is an equivalence relation on a set S and $a \in S$, then the* equivalence class *of a defined by $\simeq$ is the set*

$$[a]_{\simeq} = \{x \in S \mid x \simeq a\}.$$

The set of all equivalence classes of a set is called a quotient set.

Definition 2.10 (Quotient set). *Given a set S and an equivalence relation $\simeq \, \subseteq S \times S$, the set*

$$S/{\simeq} = \{[x]_{\simeq} \mid x \in S\}$$

is called the quotient set *of S over $\simeq$.*

2.1.3 Closures of Relations

We often need relations that satisfy certain properties, i.e. relations that are transitive, reflexive, or symmetric. The closure operators on relations are used to define relations that have these properties, based on arbitrary relations that do not necessarily have them.

Definition 2.11 (Transitive closure)**.** *The* transitive closure R' *of a binary relation R is the minimal transitive relation that contains R, i.e. $a\ R'\ b$ if $a\ R\ b$ or $\exists c_0, \ldots, c_k\,.(a, c_0), (c_0, c_1), \ldots, (c_{k-1}, c_k), (c_k, b) \in R$.*

Definition 2.12 (Reflexive closure)**.** *The* reflexive closure R' *of a binary relation R is the minimal reflexive relation that contains R, i.e. $a\ R'\ b$ if $a\ R\ b$ or $a = b$.*

Definition 2.13 (Symmetric closure)**.** *The* symmetric closure R' *of a binary relation R is the minimal symmetric relation that contains R, i.e. $a\ R'\ b$ if $a\ R\ b$ or $b\ R\ a$.*

Sometimes we use combinations of these closures; for instance the *reflexive-transitive* closure of a relation R is the minimal reflexive and transitive relation that contains R.

2.2 Graphs and Morphisms

Graphs consist of nodes, and edges that connect these nodes to each other. Although there exist many ways to define graphs, this thesis uses *labeled*, *directed* graphs where node labels are represented as self-edges. The classical formal definition of such graphs is as follows.

Definition 2.14 (Simple graph)**.** *Given a fixed set of edge labels* Label*, a simple graph $G = (V_G, E_G)$ is a pair consisting of a set of nodes V_G and a set of edges $E_G \subseteq (V_G \times \mathsf{Label} \times V_G)$.*

- For edges of a graph, three functions are present: $src_G, tgt_G : E_G \to V_G$ map edges to their *source* and *target* nodes respectively; $lbl_G : E_G \to \mathsf{Label}$ maps edges to their labels. These functions are defined as

$$src\Big(v, l, w\Big) = v \qquad lbl\Big(v, l, w\Big) = l \qquad tgt\Big(v, l, w\Big) = w.$$

- For two graphs G and G', if $V_{G'} \subseteq V_G$ and $E_{G'} \subseteq E_G$, then G' is a subgraph of G and G is a supergraph of G', written as $G' \subseteq G$.
- We use graphs and simple graphs as synonyms.

As a further convention, we will use node labels to represent self-edges in all figures of this thesis, unless stated differently.

Relations between graphs are represented using graph morphisms. A graph morphism maps the nodes and edges of one graph to those of another one, such that the source, target, and label of each edge is preserved.

Definition 2.15 (Graph morphism). *Given two graphs G and H, a graph morphism $f : G \rightarrow H$ is a pair of functions $f_V : V_G \rightarrow V_H$ and $f_E : E_G \rightarrow E_H$, such that $f((s, l, t)) = (f_V(s), l, f_V(t))$ for all $(s, l, t) \in \operatorname{dom} f_E$.*

- Graph morphism f is called *total* if both functions f_V and f_E are total, i.e. they contain a mapping for any element in their domain; f is called *partial* if f_V and f_E are partial.
- Graph morphism f is *injective*, *surjective*, or *bijective* if both functions f_V and f_E are injective, surjective, or bijective respectively.
- If morphism f is bijective, it is called an isomorphism. We use $G \cong H$ to denote that there is an isomorphism from graph G to graph H.
- A graph morphism $f : G \rightarrow G$ from a graph into itself is called an *endomorphism.* If f is also an isomorphism it is an *automorphism*; when an automorphism is the identity morphism, we call it *trivial.*
- For morphism $f : G \rightarrow H$, we call G the source of f and H its target, denoted $src(f)$ and $tgt(f)$ respectively.
- The image of f, denoted $\operatorname{im} f$, is a graph (V', E') such that $V' = \operatorname{im} f_V$ and $E' = \operatorname{im} f_E$.

Figure 2.2 illustrates an example of a graph morphism. The edge mapping is not explicitly shown, but follows uniquely from the node mapping. The nodes are indexed in this figure with their node identities.

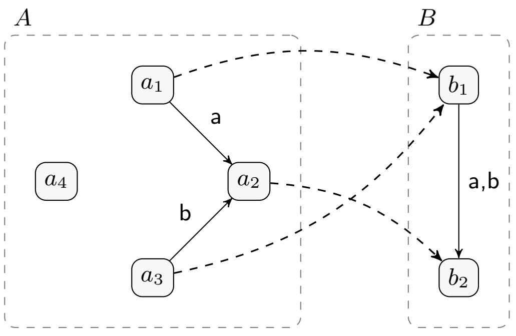


Figure 2.2: Example of a graph morphism

Nodes and edges of different graphs may overlap. Therefore we use the concept of disjoint union to denote the union of graphs.

Definition 2.16 (Disjoint union of two graphs). *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be graphs. Then, if $V_1 \cap V_2 = \varnothing$, $H = (V_1 \cup V_2, E_1 \cup E_2)$ is called a* disjoint union *of G_1 and G_2, denoted $G_1 \uplus G_2$. If, however, $V_1 \cap V_2 \neq \varnothing$, then $G_1 \uplus G_2$ is defined by $H = (V_1 \cup V_2', E_1 \cup E_2')$, where $G_2' = (V_2', E_2')$ is a graph isomorphic to G_2, such that $V_2' \cap V_2 = \varnothing$, and $E_2' \cap E_2 = \varnothing$.*

Furthermore, given a set of graphs $\mathcal{G}$, we denote the disjoint union of all elements in $\mathcal{G}$ as $\uplus\mathcal{G}$.

In some occasions we need to combine graph morphisms, e.g. when two distinct graphs both have a morphism into the same target graph. We denote this combination of graph morphisms as a union of morphisms, which is defined as follows:

Definition 2.17 (Union of morphisms)**.** *If, for graphs A, B, and C where $V_A \cap V_B = \varnothing$, $f : A \to C$ and $g : B \to C$ are two morphisms, $f \cup g : (A \uplus B) \to C$ is defined as:*

$$(f \cup g)(x) = \begin{cases} f(x) & \textit{if } x \in A \\ g(x) & \textit{if } x \in B \end{cases}.$$

Lemma 2.18. *Given graphs A, B, and C and graph morphisms $f : A \to C$ and $g : B \to C$, the union $f \cup g : (A \uplus B) \to C$ of these morphisms is again a graph morphism.*

Proof. This is trivial, because f and g are graph morphisms and $V_A \cap V_B = \varnothing$ (and hence $E_A \cap E_B = \varnothing$). $\square$

2.2.1 Quotient Graphs

As with sets, the quotient construction can be used to partition graph nodes into their equivalence classes. With respect to graphs, the resulting quotient is again a graph in which all nodes of the same equivalence class are merged.

Definition 2.19 (Quotient graph)**.** *Let $G = (V_G, E_G)$ be a graph and let $\simeq \subseteq V_G \times V_G$ be an equivalence relation. Then the graph $H = (V_H, E_H)$ with*

– $V_H = V_G/\simeq$, *and*
– $E_H = \{([w_1]_\simeq, l, [w_2]_\simeq) \mid (w_1, l, w_2) \in E_G\}$

is called the quotient graph *of G over $\simeq$, denoted $G/\simeq$.*

Definition 2.20. *Given a graph G, if $\simeq \subseteq V_G \times V_G$ is an equivalence relation on nodes of G and $x \in V_G \cup E_G$, then the equivalence class of V containing x is defined as:*

$$[x]_\simeq = \begin{cases} [x]_\simeq & \textit{if } x \in V_G \\ \{(n_1, a, n_2) \mid n_1 \simeq src(x), n_2 \simeq tgt(x), a = lbl(x)\} & \textit{if } x \in E_G \end{cases}$$

Lemma 2.21. *Given a graph G and an equivalence relation $\simeq \subseteq V_G \times V_G$, let $H = G \,/\, \simeq$ be the quotient graph of G over $\simeq$. Then there exists a total surjective graph morphism $G \to H$.*

Proof. Let $f = (f_V, f_E)$ be a pair of functions such that $f_V : V_G \to V_H$ and $f_E : E_G \to E_H$ are defined as follows, for all $x \in V_G \cup E_G$:

$$f_{V,E}(x) = [x]_\simeq.$$

Since f_V and f_E are uniquely defined for all $x \in V_G \cup E_G$, f is well-defined and total. From Definition 2.20 it follows that f preserves sources, targets, and labels of all $e \in E_{E_G}$ and hence f is a graph morphism. Finally, surjectivity of f follows from the definition of f.

This proves that $f : G \to H$ is a surjective graph morphism, as required. □

Lemma 2.22. *Let G and H be graphs and let $R \subseteq V_H \times V_H$ be an equivalence relation. Then, if there exists a (total) graph morphism $m : G \to H$, there also exists a (total) graph morphism $m' : G \to H \mathbin{/} R$ such that $m'(x) = [m(x)]_R$ for all $x \in V_G \cup E_G$.*

Proof. We define m' as $m' = f \circ m$ where $f : H \to H \mathbin{/} R$ is a total graph morphism. Existence of f follows from Lemma 2.21: f is defined as $f(x) = [x]_R$ for all $x \in V_H \cup E_H$.

The required property $m'(x) = [m(x)]_R$ is proved as follows:

$$\begin{aligned} m' &= f \circ m \\ m'(x) &= f(m(x)) \\ m'(x) &= [m(x)]_R \end{aligned}$$

for all $x \in V_G \cup E_G$. Finally, since f is total, totality of m' follows directly from the totality of m. □

2.3 Commutative Diagrams

A mathematical formalism often used in combination with graph transformations is *category theory.* Category theory allows to reason about mathematical structures and relationships between these sructures in a uniform and abstract way. A basic introduction to category theory is given in [BW90]. Despite the expressive power of this formalism with respect to graph transformations, we shall not use this theory in this thesis; rather we will use some concepts and proof strategies from this theory.

One of the concepts we use is a *diagram.* We define a diagram as follows.

Definition 2.23 (Diagram)**.** *A diagram is a pair $(\mathcal{G}, \mathcal{M})$ where $\mathcal{G}$ is a set of graphs and where $\mathcal{M}$ is a set of morphisms between these graphs.*

Diagrams offer an intuitive way to reason about objects and relationships between these objects; in this thesis the objects are graphs and the relationships are graph morphisms.

A special kind of diagram is a commutative diagram; a *commutative diagram* is a diagram such that, for any two objects in the diagram, any path between these objects through the morphisms of the diagram yields the same result by composition.

Example 2.24. *Consider the simple diagram illustrated in Figure 2.3, which is a canonical example of a commutative diagram. It consists of three graphs A, B, and C, and three morphisms $f : A \to B$, $g : B \to C$, and $h : A \to C$ between these graphs.*

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[dr, "h"'] & B \arrow[d, "g"] \\ & C \end{tikzcd}$$

Figure 2.3: Example of a diagram

This diagram is commutative *(or* commutes*) if, and only if, h is the composite function $g \circ f$.*

2.4 Graph Grammars

As defined in Section 1.2.1, a graph grammar consists of a set of production rules and a start graph. Production rules are used to define manipulations on graphs and the start graph acts as the host graph for the first applied production rule. The transformation process consists of a sequence of transformation steps in which the current graph is matched against all production rules, after which one of the matching rules is applied. The new graph produced by the rule is used as host graph in succeeding transformation steps.

In this section we shall introduce formal definitions concerned with graph grammars, based on the single-pushout approach to graph tansformation as is specified in [Löw93]. First we define a production rule.

Definition 2.25 (Graph production)**.** *A* graph production rule *$r = L \xrightarrow{p} R$ consists of a partial graph morphism $p : L \to R$ and two graphs L and R, where L is called the* left-hand-side *and R the* right-hand-side *of r.*

Morphism p determines the change from G to H upon application of r. Elements in R that are not in $\operatorname{im} p$ should be added, whereas elements in L that are not in $\operatorname{dom} p$ have to be deleted. Furthermore, elements that are in $\operatorname{dom} p$ that are being mapped to the same element in $\operatorname{im} p$ have to be merged.

According to the single pushout approach, deletion of a node in G automatically causes the deletion of all incident edges and deletion of elements is favored over preservation in case of conflicts [CMR+97]. A constructive definition of a graph transformation in the single-pushout approach is based upon a definition by Rensink [Ren08] and reads as follows.

Definition 2.26 (Graph Transformation)**.** *Let G be a graph and $r : L \xrightarrow{p} R$ be a production rule, such that $V_G \cap V_R = \varnothing$. Then, if there exists a total graph morphism $m : L \to G$, $G \overset{p,m}{\Longrightarrow} H$ is a graph transformation from host graph G into product graph $H = (V_H, E_H)$ such that:*

– $\overline{V} = V_G \cup V_R$.
– $\simeq \subseteq \overline{V} \times \overline{V}$ *is the smallest equivalence relation such that $p(v) \simeq m(v)$ for all $v \in V_L$.*
– $V_H = \{X \in \overline{V} \,/ \simeq \mid m^{-1}(X) \subseteq \operatorname{dom} p\}$

– $E_H = \{(X, l, Y) \mid (v_1, l, v_2) \subseteq E_G \cup E_R, v_1 \in X \in V_H, v_2 \in Y \in V_H\}$

There exist two partial graph morphisms $p^* : G \to H$ and $m^* : R \to H$ such that $p^* \circ m = m^* \circ p$; in other words, such that the pushout diagram given in Figure 2.4 commutes.

$$\begin{tikzcd}
L \arrow[r, "p"] \arrow[d, "m"'] & R \arrow[d, "m^*"] \\
G \arrow[r, "p^*"] & H
\end{tikzcd}$$

Figure 2.4: Pushout diagram.

Example 2.27. *Figure 2.5 demonstrates the application of a production rule $p : L \to R$ on a graph G. Nodes in graphs L, R, and G are indexed with their node identities; nodes in H with the node identities of the nodes they originate from. The rule searches for two nodes with an edge labelled "a" inbetween and subsequently removes this edge and its target node.*

Equivalence relation $\simeq$, as defined in Definition 2.26, is the relation

$$\simeq\, = \{(3,4), (4,3), (3,3), (4,4), (5,5), (6,6)\}$$

and hence all $\simeq$-equivalence classes of $V_G \cup V_R$ are $\{3,4\}$, $\{5\}$, and $\{6\}$. These $\simeq$-equivalence classes are the nodes of graph H, as is depicted in figure 2.5.

Because of the restriction $m^{-1}(x) \subseteq \operatorname{dom} p$, as given in Definition 2.26, equivalence class $\{5\}$ should be omitted: $m^{-1}(5) = 2$ and $2 \notin \operatorname{dom} p$, where 2 and 5 represent nodes with corresponding node identities. Therefore, the correct product graph of the application of rule p to host graph G is the subgraph of H drawn with thick lines.

Definition 2.28 (Graph Grammar). *A* graph grammar *$GG = (G_0, \mathcal{P})$ is a pair consisting of a graph G_0 and a set of production rules $\mathcal{P}$. G_0 is called the start graph of GG.*

A *derivation* of GG, denoted as $G_0 \Rightarrow^* G_n$ is a sequence of graph transformations ($G_0 \overset{p_1,m_1}{\Longrightarrow} G_1 \overset{p_2,m_2}{\Longrightarrow} \ldots \overset{p_n,m_n}{\Longrightarrow} G_n$). All graphs G_n such that $G_0 \Rightarrow^* G_n$ is a derivation of GG are *production graphs* of GG; the *language* $\mathcal{L}$ of GG is the set of all production graphs of GG. Furthermore, rule graphs of GG are all graphs in the set $\{L_r \mid r \in \mathcal{P}\} \uplus \{R_r \mid r \in \mathcal{P}\}$

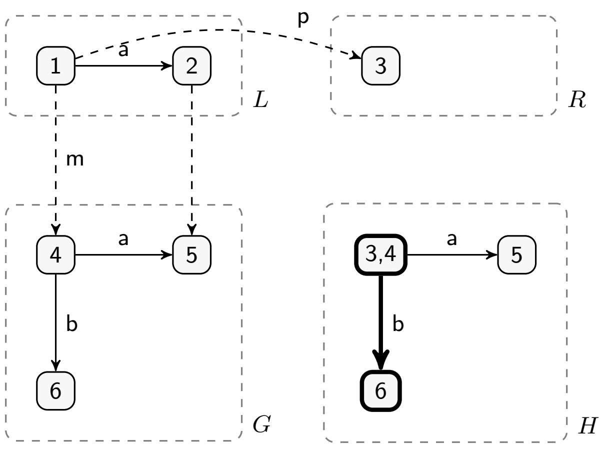


Figure 2.5: Example graph transformation, used in Example 2.27. The subgraph of H drawn with thick lines is the actual result of the transformation.

CHAPTER 3

Type Inference for Graph Grammars

In this chapter we develop an algorithm that computes a type graph for a graph grammar. A type graph is a graph, consisting of node- and edge types, that describes the structure of a set of graphs. All graphs in this set are called *instances* of the type graph and the *typing* of an instance over the type graph is represented by a total graph morphism.

The main issue recalled at the beginning of this chapter, in section 3.1, is to specify what information a type graph should contain to give rise to a certain set of instances. We shall see that a type graph must at least contain node- and edge types for the nodes and edges of all of its instances. But, on the other hand, a type graph should not contain unnecessary information and needs to be a finite structure, even if the set of production graphs of a graph grammar is infinite.

After that, we shall define an ordering among type graphs, expressing which type graphs give a better classification for a set of instance graphs than others. Using this ordering, we shall then define a *perfect type graph* for a graph grammar, which is the type graph that gives the best classification for this graph grammar.

This perfect type graph is the type graph we then try to compute algorithmically in Section 3.2. Unfortunately, this turns out to be impossible in general, as we will see on page 33; therefore, we shall come up with two algorithms computing type graphs that are approximations of the perfect type graph.

The first, naive, algorithm assumes that all production rules eventually become applicable. It adds typing information for all production rules to the type graph. Because it does so, it possibly adds too much information to the type graph, yielding a type graph that contains unnecessary, or *spurious*, elements. The second, improved, algorithm tries to minimise the number of spurious elements by omitting information from some production rules for which it certainly knows that they never become applicable. After introducing this second algorithm, we show that this improved algorithm indeed produces a better type graph than the first.

Finally, in section 3.3, we conclude the chapter with a discussion in which all topics introduced in this chapter are briefly reviewed.

3.1 Type Graphs

In Section 2.2 we defined a graph as a tuple (V, E) containing a set of nodes V and a set of edges $E \subseteq V \times \mathsf{Label} \times V$. As is the case in programming languages, a type can be assigned to each element of a graph [dLBE$^+$07]. In this thesis we do this by defining a *typing*, which is a total graph morphism into a *type graph*. Such type graph describes and constrains the structure of all its instances.

Definition 3.1 (Type graph)**.** *A type graph $T = (V_T, E_T)$ is a graph where V_T represents a set of node types and E_T a set of edge types. A* typing *of a graph G into T is a total graph morphism $\tau_G : G \to T$. We call T a* type *of G and G an* instance *of T.*

We use $\mathcal{I}_T$ to denote the set of instances of a type graph T.

Example 3.2. *Consider Figure 3.1, showing an instance graph G and a type graph T. The labels inside the nodes represent node labels. A typing morphism $\tau_G : G \to T$ would map the nodes labeled "List" and "Cell" of G onto their respective nodes in T, as it would also do for edges labeled "head", "next", and "val". τ_G will map the nodes with labels "2", "4", and "10" of G onto the node labeled "$\{2, 4, 10\}$" in T. Here we use this notation $\{2, 4, 10\}$ to denote that the concerning node has three self-edges, labelled 2, 4, and 10 respectively.*

Note that we have explicitly drawn the self-edge labelled next *in this figure, in order to make a visible distinction between this intended self-edge and the self-edges representing node labels.*

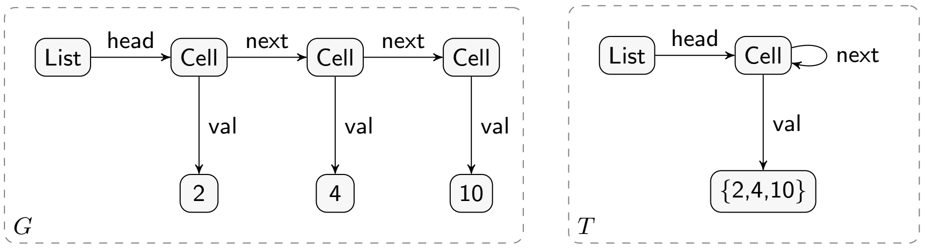


Figure 3.1: A graph G representing a linked list and a type graph T for this graph.

Similar to conventional programming languages, a type graph puts a restriction on the set of legal instances for objects during program execution, i.e. it restricts the legal instances of objects to those that are in some sense correct. In the context of typed graph grammars, graphs are correct if, and only if, they have a typing into the type graph [MvEDJ05]. Analogously, sets of graphs are correct only if all graphs in this set are instances of the type graph.

Definition 3.3 (Type graph for a set of graphs)**.** *Let T be a graph and $\mathcal{G}$ a set of graphs. T is a type graph for $\mathcal{G}$ if $\mathcal{G} \subseteq \mathcal{I}_T$.*

This defines some lower bound for a type graph: a type graph for a set of graphs must at least contain typings for all graphs in this set. On the other hand, type graphs should ideally only contain node- and edge types in the image of the node- and edge mappings of some typing morphism. We shall call these elements *instantiable*, elements that are not instantiable are *spurious*.

This concept of spuriousness of node- and edge types gives rise to a special class of type graphs, which we shall call *genuine*. A genuine type graph is a type graph that contains no spurious elements.

Definition 3.4 (Genuine type graph)**.** *Given a type graph T and a set of instance graphs $\mathcal{G}$, T is a* genuine type graph *for $\mathcal{G}$ if the typing morphisms $\tau_G : G \to T$ for all $G \in \mathcal{G}$ are collectively surjective, meaning that every element $x \in V_T \cup E_T$ is in the image of at least one typing morphism.*

When constructing a type graph, we prefer genuine type graphs over others. Spurious elements do not contain any valuable information and can therefore be safely omitted. We shall now prove that all type graphs containing spurious elements can be reduced to genuine type graphs by removing these non-instantiable elements.

Lemma 3.5. *Every type graph has a subgraph that is a genuine type graph.*

Proof. Let T be a type graph and $\mathcal{G}$ a set of instance graphs. Since all graphs in $\mathcal{G}$ are instances of T, there exist typing morphisms $\tau_G : G \to T$ for all $G \in \mathcal{G}$.

Furthermore let

$$V^* = \{v \in V_T \mid \exists G \in \mathcal{G} \,.\, v \in \operatorname{im} \tau_{G,V}\}$$

be a set of nodes of T and let

$$E^* = \{e \in E_T \mid \exists G \in \mathcal{G} \,.\, e \in \operatorname{im} \tau_{G,E}\}$$

be a set of edges of T, such that all $v \in V^*$ and all $e \in E^*$ are in the image of some typing morphism τ_G. Then the graph (V^*, E^*) is a genuine type graph. □

3.1.1 An Ordering over Type Graphs

In the previous section we have defined a type graph and a typing from instance graphs into this type graph. Without any restrictions, however, any graph or set of graphs has an infinite number of possible type graphs. Although genuinity already poses some restriction on the set of desirable type graphs, we want to be able to select the best type graph out of a set of possible type graphs.

In order to do so, we shall now define an ordering over type graphs, based on their strength. The *strength* of a type graph defines how well it gives a classification for its instances; stronger type graphs are more restrictive than weaker type graphs. In other words, the number of instances of a stronger type graph is smaller than the number of instances of a weaker type graph. Formally, this is expressed as follows.

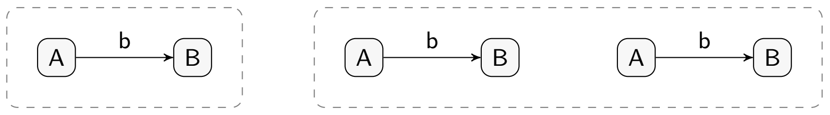


Figure 3.2: Two type graphs that are not isomorphic, but have the same set of instances

Definition 3.6 (Strength of a type graph)**.** *Let T_1 and T_2 be two type graphs. Then T_1 is stronger than T_2 if $\mathcal{I}_{T_1} \subseteq \mathcal{I}_{T_2}$.*

This stronger-than relation defines a preorder over type graphs; this follows directly from the reflexivity- and transitivity properties of $\subseteq$. It is not a partial order, however, since different type graphs can have the same set of instances; this violates the antisymmetry property. A simple example of two type graphs that are not isomorphic but that have the same set of instances is given in Figure 3.2.

The main drawback of this stronger-than relation is that it is not straightforwardedly derivable from the structure of the type graphs associated by this relation. Therefore we shall now define a binary relation $\leq$ over type graphs, purely based on the structure of the type graphs, and thereafter we shall prove that $\leq$ and the stronger-than relations are equivalent.

Definition 3.7. *Given a set of type graphs $\mathcal{T}$, $\leq \subseteq \mathcal{T} \times \mathcal{T}$ is the relation defined as*

$$T_1 \leq T_2 \textit{ if there exists a graph morphism from } T_1 \textit{ into } T_2.$$

Lemma 3.8. *Given a set of type graphs $\mathcal{T}$, $\leq$ forms a preorder over $\mathcal{T}$.*

Proof. Recall that $\leq$ needs to be reflexive and transitive to be a preorder.

For reflexivity, we have to show that for all $T \in \mathcal{T}$ there exists a graph morphism from T into itself. This holds since each graph has an identity morphism $id : T \to T$.

For transitivity, we have to show that for all $T, T', T'' \in \mathcal{T}$ such that there exitst two graph morphisms $\varphi : T \to T'$ and $\psi : T' \to T''$, there also exists a graph morphism $\chi : T \to T''$. This directly follows from morphism composition and the required morphism is defined as $\chi = \psi \circ \varphi$. □

Having now identified two preorders over type graphs, we shall now examine the similarity of these two preorders. Intuitively, given two graphs T_1 and T_2 such that $T_1 \leq T_2$, there are two reasons why the number of instances of T_2 can be larger than the number of instances of T_1, namely

1. If the morphism $T_1 \to T_2$ is non-surjective, T_2 contains node- and edge types that T_1 does not contain. These node- and edge types may give rise to new instances with respect to T_1.

2. If the morphism is non-injective, multiple elements of T_1 are mapped onto the same element in T_2. The following example explains why this may give rise to new instances with respect to T_1.

Example 3.9. *Figure 3.3 illustrates two type graphs, T_1 and T_2, and one instance graph, G. It is clear that there exists a graph morphism from T_1 into T_2, which maps the two nodes labeled B in T_1 onto the same node labeled B in T_2 and leaves the rest of the graph intact.*

Because of this merging, the node labeled B in T_2 has all incident edges of the two nodes labeled B in T_1 combined; hence G has a typing into T_2 although it does not have a typing into T_1.

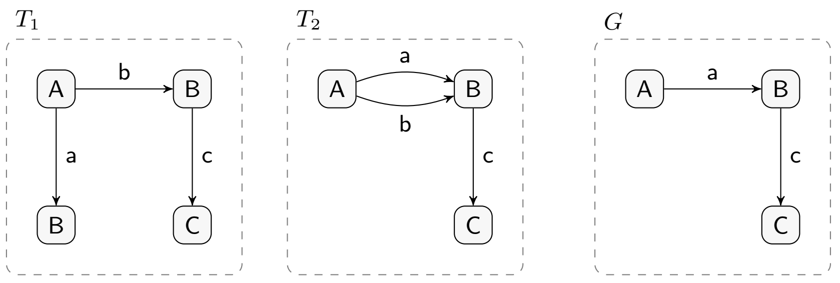


Figure 3.3: Two type graphs T_1, T_2 and one instance graph G. G is an instance of T_2 and is not an instance of T_1.

An interesting observation is that a $\leq$-smaller—that is, smaller with respect to $\leq$—type graph is actually not smaller than a larger type graph in terms of numbers of nodes and edges. As an example consider the example above, where $T_1 \leq T_2$ while T_1 has more nodes and edges than T_2.

Although this may be confusing, a justification for this ordering is the restrictiveness of both type graphs: if $T_1 \leq T_2$, then $\mathcal{I}_{T_1}$ is a subset of $\mathcal{I}_{T_2}$ and hence T_1 has a smaller set of instance graphs.

We shall now prove that the two preorders $\leq$ and the stronger-than relation over type graphs are equivalent.

Lemma 3.10. *Let T_1 and T_2 be type graphs. Then $T_1 \leq T_2 \iff \mathcal{I}_{T_1} \subseteq \mathcal{I}_{T_2}$.*

Proof. We shall prove both directions of this equivalence separately.

$\Longrightarrow$ This follows directly from morphism composition. Let $G \in \mathcal{I}_{T_1}$; we prove that $G \in \mathcal{I}_{T_2}$ under the assumption that $T_1 \leq T_2$.
Let $\tau_{G,1} : G \to T_1$ be a typing morphism and let $f : T_1 \to T_2$ be a graph morphism. f exists since $T_1 \leq T_2$. Then the typing morphism $\tau_{G,2} : G \to T_2$ is defined as $\tau_{G,2} = f \circ \tau_{G,1}$.

$\Longleftarrow$ Because each graph trivially is an instance of itself, it holds that $T_1 \in \mathcal{I}_{T_1}$. Then, since $\mathcal{I}_{T_1} \subseteq \mathcal{I}_{T_2}$, it also holds that $T_1 \in \mathcal{I}_{T_2}$ and hence that there exists a typing morphism $\tau_{T_{1,2}} : T_1 \to T_2$. Since $\tau_{T_{1,2}}$ is a total graph morphism, we have proved that there exists a graph morphism from T_1 into T_2 and hence that $T_1 \leq T_2$. □

Since it is possible for two disjoint type graphs T_1, T_2 that $T_1 \leq T_2$ and $T_2 \leq T_1$ (as shown in Figure 3.2), $\leq$ is not a partial order. With respect to the instances of T_1 and T_2 this means that $\mathcal{I}_{T_1} = \mathcal{I}_{T_2}$. We shall now use this information to introduce an equivalence relation $\simeq$ over type graphs, which allows us to divide type graphs into equivalence classes based on their instance graphs.

Definition 3.11. *Given a set of type graphs $\mathcal{T}$ and preorder $\leq \subseteq \mathcal{T} \times \mathcal{T}$, as defined in definition 3.7, $\simeq \subseteq \mathcal{T} \times \mathcal{T}$ is the relation defined as*

$$T_1 \simeq T_2 \text{ if } T_1 \leq T_2 \text{ and } T_2 \leq T_1.$$

Since $\simeq$ is reflexive, symmetric, and transitive, it is an equivalence relation. Moreover, $\leq$ up to $\simeq$ is a partial order over sets of type graphs. These sets are equivalence classes with respect to $\simeq$; type graphs within the same equivalence class have the same set of instances.

We shall not elaborate on these equivalence classes. Rather, we introduce canonical representatives (up to isomorphism) for every equivalence class of $\simeq$, which we will call *proper type graphs.* A proper type graph is a type graph that does not contain total non-surjective endomorphisms. As defined in section 2.2, an endomorphism is a graph morphism from a graph into itself.

Definition 3.12 (Proper type graph)**.** *Let T be a type graph. T is a* proper type graph *if contains no total non-surjective endomorphisms.*

Type graphs that are not proper contain duplications of node and edge types. This in contrast to non-genuine type graphs. Such type graphs contain node- and edge types that are never instantiated, while non-proper type graph contain duplications of node and edge types.

These duplications give rise to ambiguous typings from instances into such type graphs. Therefore we prefer proper type graphs over type graphs that are not proper.

Example 3.13. *Figure 3.4 illustrates the difference between properness and genuinity of type graphs. Suppose that the graph in Figure 3.4a is a proper and genuine type graph for some set of instance graphs $\mathcal{I}$.*

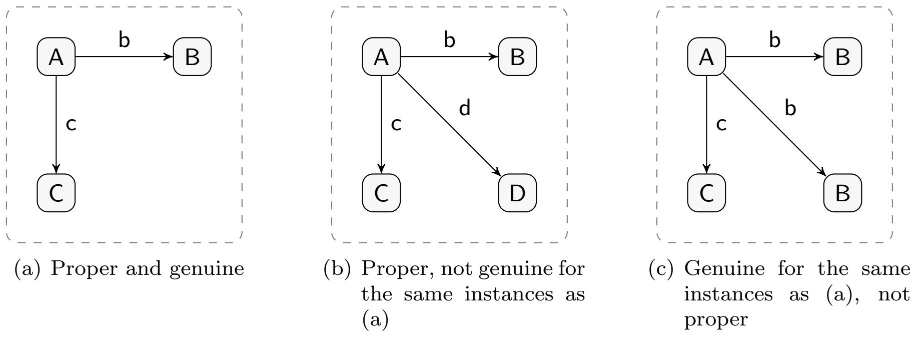
Then it is obvious that the graph in Figure 3.4b is not genuine for $\mathcal{I}$: it contains a spurious node labelled D *and a* d*-labelled edge from node* A *to this node. The graph is proper, however, because it contains no total non-surjective endomorphisms.*

The graph in Figure 3.4c does contain non-surjective endomorphisms, mapping the nodes labelled B *and the edges labelled* b *onto each other; hence, this type graph is not proper. It is genuine for $\mathcal{I}$, however, because it contains no non-instantiable nodes types.*

Proposition 3.14. *Every $\simeq$-equivalence class contains a proper type graph.*

Proof. Let T be an arbitrary type graph. If T is proper, the $\simeq$-equivalence class of T contains a proper type graph: T itself.

If T is not proper, it contains non-surjective endomorphisms. Then there exists a type graph $T' \subset T$, in which all elements that are not in the image of any of



(a) Proper and genuine (b) Proper, not genuine for the same instances as (a) (c) Genuine for the same instances as (a), not proper

Figure 3.4: Difference between proper and genuine type graphs

these non-surjective endomorphisms are removed. $T \simeq T'$ follows directly from $T' \subset T$. $\square$

Moreover, we shall now prove that all $\simeq$-equivalence classes have a unique proper type graph, up to isomorphism.

Lemma 3.15. *Let T_1, T_2 be two proper type graphs such that $T_1 \simeq T_2$. Then $T_1 \cong T_2$.*

Proof. According to Lemma 3.10, it holds that $\mathcal{I}_{T_1} = \mathcal{I}_{T_2}$. Since all type graphs trivially are instances of themselves, this means that T_1 and T_2 are also instances of each other and hence there exist total graph morphisms $m_1 : T_1 \to T_2$ and $m_2 : T_2 \to T_1$, as illustrated below.

$$\begin{tikzcd} T_1 \arrow[r, bend left, "m_1"] & T_2 \arrow[l, bend left, "m_2"] \end{tikzcd}$$

Morphism composition gives two extra morphisms: $m_2 \circ m_1 : T_1 \to T_1$ and $m_1 \circ m_2 : T_2 \to T_2$. Both $m_1 \circ m_2$ and $m_2 \circ m_2$ are total; this follows directly from the totality of both m_1 and m_2.

Surjectivity of $m_1 \circ m_2$ and $m_2 \circ m_1$ follows directly from Definition 3.12: if one of these morphisms is not surjective, either T_1 or T_2 has a non-surjective endomorphism and therefore is not proper.

For injectivity suppose that m_1 is non-injective. Then $|V_{T_2}| < |V_{T_1}|$, which means that T_2 has less nodes than T_1. Since m_2 is surjective, $|V_{T_1}| \leq |V_{T_2}|$ holds, which results in $|V_{T_2}| < |V_{T_2}|$. Since T_2 is finite, this is a contradiction and therefore m_1 must be injective, as is also the case with m_2.

Thus we have proved that there exist graph isomorphisms from T_1 into T_2 and vice versa. Therefore T_1 and T_2 are isomorphic. $\square$

Thus we have defined a preorder $\leq$ over type graphs which, in combination with the equivalence relation $\simeq$, forms a partial order over equivalence classes of type graphs. Each of these equivalence classes contains a proper type graph, which is unique up to isomorphism according to Lemma 3.15 and is preferred over non-proper type graphs. In other words, the preferred type graph for a $\simeq$-equivalence class is a proper type graph and $\leq$ is a partial ordering over isomorphism classes of proper type graphs.

3.1.2 The Smallest Type Graph

Having defined an ordering among type graphs, purely based upon their structure, we shall now examine the boundaries of this ordering, particularly the smallest element because this is the most restrictive, and therefore the desired, type graph. Since this graph is the smallest type graph with respect to $\leq$, we shall call this type graph the *smallest type graph*.

Many type systens identify a weakest type. For instance, in an object oriented setting, the weakest type corresponds to the type Object, see e.g. [Pie02]. The weakest type is the most general (every possible object is an instance of this type); the strongest the most restrictive. In the setting of typed graph grammars, however, we are not interested in finding the weakest type. Actually, it is easy to define the weakest type graph: a graph consisting of one node and self-edges for labels in Label, which is a finite set.

We are rather interested in finding the strongest type graph, the type graph that gives the best classification for its instances. As stated in section 3.1.1, this element is the smallest element of $\leq$.

We shall now prove that such a *smallest type graph* always exists, but first we introduce a related concept, namely a *minimal type graph*. A minimal type graph is a type graph such that there exist no smaller type graphs.

Definition 3.16 (Minimal type graph)**.** *Let $\mathcal{T}$ be a set of type graphs. Then $T \in \mathcal{T}$ is a minimal type graph with respect to $\leq$ if $T' \leq T$ implies $T \cong T'$ for all $T' \in \mathcal{T}$.*

Besides being minimal, the type graph we try to find is the smallest type graph with respect to $\leq$. Although the difference between a minimal type graph and the smallest type graph is subtle, the smallest type graph is a minimal type graph with the extra requirement that it is stronger than any other type graph. This is defined as follows.

Definition 3.17 (Smallest type graph)**.** *Let $\mathcal{T}$ be a set of type graphs. Then $T \in \mathcal{T}$ is the smallest type graph with respect to $\leq$ if $T \leq T'$ for all $T' \in \mathcal{T}$.*

Example 3.18. *Figure 3.5 is a graphical representation of two partially ordered sets, which we shall call set A for the set represented by Figure 3.5a and set B for the set represented by Figure 3.5b. The partial order on the elements of the sets is represented by lines between the elements, with the convention that these lines go from x to y in upward direction if $x < y$, i.e. in set A we have $d < a$, $d < b$, and $c < b$.*

Both sets have minimal elements. In set A these are c and d; set B has one minimal element, e, which is also its smallest element. Set A has no smallest element, since the preorder over this set does not relate c and d to each other.

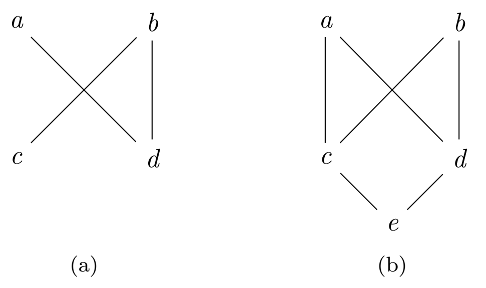


Figure 3.5: Graphical representation of two partially ordered sets. (b) has a smallest element while (a) does not.

Although the smallest object of an ordering usually is unique, this is not the case with the smallest type graph. Since $\leq$ is a preorder, it gives rise to a set of smallest type graphs. According to Lemma 3.15, however, the smallest type graph is unique up to $\simeq$.

Proposition 3.19. *The smallest type graph for a set of instance graphs $\mathcal{G}$ is genuine with respect to $\mathcal{G}$.*

This follows directly from Definition 3.4.

3.1.3 A Type Graph for a Diagram

In this section we shall define a type graph for a diagram. As specified in Section 2.3, a diagram is a pair $(\mathcal{G}, \mathcal{M})$ consisting of a set of graphs $\mathcal{G}$ and a set of morphisms $\mathcal{M}$ between these graphs. We use diagrams as an intermediate step towards type graphs for graph grammars, because diagrams allow us to make proofs in a uniform way, which thereafter can be easily reused for graph grammars.

A type graph for a diagram must satisfy at least two properties. First, it must have a typing for all graphs in the diagram; second, if two nodes are mapped onto each other by any morphism or any composition of morphisms, the typing morphisms must map them onto the same node type in the type graph. This last property comes down to the commutativity property of the typing morphisms.

Definition 3.20 (Type graph for a diagram)**.** *Let $(\mathcal{G}, \mathcal{M})$ be a diagram. A graph T is a* type graph for $(\mathcal{G}, \mathcal{M})$ *if T is a type graph for $\mathcal{G}$, with typing morphisms τ_G for all $G \in \mathcal{G}$, such that*

$$\forall m \in \mathcal{M} \,.\, m : G \to H \implies \tau_H \circ m = \tau_G.$$

This commutativity property is depicted in the following diagram:

$$\begin{tikzcd}
G \arrow[rr, "m"] \arrow[dr, "\tau_G"'] & & H \arrow[dl, "\tau_H"] \\
& T &
\end{tikzcd}$$

Moreover, from morphism composition it directly follows that, for a diagram $(\mathcal{G}, \mathcal{M})$, if there exists a sequence of morphisms $m_1, m_2, \ldots, m_n \in \mathcal{M}$ such that $H = m_1(m_2(\ldots(m_n(G))))$ for $G, H \in \mathcal{G}$, then

$$\tau_H \circ m_n \circ \ldots \circ m_2 \circ m_1 = \tau_G$$

as illustrated below:

$$\begin{tikzcd}
G \arrow[r, "m_1"] \arrow[drr, "\tau_G"'] & G' \arrow[r, "m_2"] \arrow[dr, "\tau_{G'}"] & \cdots \arrow[r, "m_n"] \arrow[d] & H \arrow[dl, "\tau_H"] \\
& & T &
\end{tikzcd}$$

We shall now define an equivalence relation $\sim$ over nodes of the graphs in a diagram $(\mathcal{G}, \mathcal{M})$. This equivalence relation is defined such that $n_1 \sim n_2$ for any $n_1, n_2 \in \uplus\mathcal{G}$, if it holds that n_2 is the image of n_1 under some morphism $m \in \mathcal{M}$ or some composition of such morphisms.

Definition 3.21. *Given a diagram $(\mathcal{G}, \mathcal{M})$, $\sim' \subseteq V_{\uplus\mathcal{G}} \times V_{\uplus\mathcal{G}}$ is the relation defined as*

$$n_1 \sim' n_2 \text{ if } n_1 = n_2 \vee \exists m \in \mathcal{M} \;.\; m(n_1) = n_2 \vee m(n_2) = n_1$$

Then $\sim$ is the transitive closure of $\sim'$.

Since $\sim$ is reflexive, symmetric, and transitive, it is an equivalence relation. Using this equivalence relation, we can now define a particular type graph for a diagram, which we shall call the *perfect type graph.* First we prove that this graph is indeed a valid type graph for $(\mathcal{G}, \mathcal{M})$.

Lemma 3.22. *The graph $\uplus\mathcal{G} \;/ \sim$ is a type graph for $(\mathcal{G}, \mathcal{M})$.*

Proof. In order for $\uplus\mathcal{G} \;/ \sim$ to be a type graph for $(\mathcal{G}, \mathcal{M})$, a typing morphism from every $G \in \mathcal{G}$ should exist and all morphisms in $\mathcal{M}$ should commute. The first property directly follows from the Lemma 2.21; the typing morphisms τ_G for all $G \in \mathcal{G}$ are defined as $\forall x \in V_G \cup E_G \,.\, \tau_G(x) = [x]_\sim$.

Next, we prove commutativity of the morphisms. Let $x \in V_G \cup E_G$ and let $m : s \to t \in \mathcal{M}$ be a morphism such that $x \in \operatorname{dom} m$. Then, according to Definition 3.21, we have $x \sim m(x)$ and hence

$$[m(x)]_\sim = [x]_\sim.$$

Rewriting gives

$$[x]_\sim \overset{def}{=} \tau_s(x) \text{ and } [m(x)]_\sim \overset{def}{=} \tau_t(m(x))$$

and hence the required property

$$\begin{aligned} \tau_t(m(x)) &= \tau_s(x) \\ m \circ \tau_t(x) &= \tau_s(x). \end{aligned}$$ □

Definition 3.23 (Perfect type graph for a diagram)**.** *Let $(\mathcal{G}, \mathcal{M})$ be a diagram. Then, the graph $\uplus\mathcal{G} \mathbin{/} \sim$ is called the* perfect type graph for $(\mathcal{G}, \mathcal{M})$.

According to this definition, the nodes of the perfect type graph for a diagram are $\sim$-equivalence classes of $\uplus\mathcal{G}$.

Proposition 3.24. *The perfect type graph for a diagram $(\mathcal{G}, \mathcal{M})$ is genuine with respect to $\mathcal{G}$.*

This follows directly from Definition 3.4. Although the perfect type graph for a diagram is genuine, it is not proper in general. It is, however, the smallest type graph for a diagram.

Theorem 3.25. *The perfect type graph for a diagram is the* smallest *type graph for this diagram, with respect to $\leq$.*

In order to prove this theorem, we need to show that there exists a graph morphism from the perfect type graph for a diagram $(\mathcal{G}, \mathcal{M})$ into any other type graph for $(\mathcal{G}, \mathcal{M})$. Before we can do so, we shall first define the structure of an arbitrary type graph for $(\mathcal{G}, \mathcal{M})$. We shall, without loss of generality, only use *genuine* type graphs in these proofs, because Lemma 3.5 states that all type graphs can be reduced to genuine type graphs and because if a type graph T is reduced to a genuine type graph T', it follows directly from Definition 3.4 that $T' \leq T$.

Lemma 3.26. *Every genuine type graph for a diagram $(\mathcal{G}, \mathcal{M})$ is isomorphic to $\uplus\mathcal{G} \mathbin{/} R$, for some equivalence relation R such that*

$$\forall n_1, n_2 \in V_{\uplus\mathcal{G}} \,.\, n_1 \sim n_2 \Longrightarrow n_1 \; R \; n_2. \tag{3.1}$$

Proof. Let T be a genuine type graph for $(\mathcal{G}, \mathcal{M})$. Then there exists a typing $\tau_G : G \to T$ for all $G \in \mathcal{G}$, such that these typing morphisms and the morphisms in $\mathcal{M}$ commute.

Let $f : \uplus\mathcal{G} \to T$ be the union of the typing morphisms τ_G, as defined in Definition 2.17. From Lemma 2.18 it follows that f is a graph morphism; f is surjective because T is genuine.

Because f is a surjection, it can be represented as a quotient $\uplus\mathcal{G} \mathbin{/} R$, where R is the smallest equivalence relation such that $v_1 \; R \; v_2$ if $f(v_1) = f(v_2)$, for all $v_1, v_2 \in V_{\uplus\mathcal{G}}$.

To prove (3.1), let n_1, n_2 be two arbitrary nodes, let G_1, G_2 be two graphs such that $n_1 \in V_{G_1}$ and $n_2 \in V_{G_2}$, and let $\tau_{G_1} : G_1 \to T$ and $\tau_{G_2} : G_2 \to T$ be two typings. Then, if $n_1 \sim n_2$ holds but $n_1 \; R \; n_2$ does not, this means that $\tau_{G_1}(n_1) \neq \tau_{G_2}(n_2)$ although $n_1 \sim n_2$. This means that there exists a morphism $m : G \to H \in \mathcal{M}$ such that $\tau_H \circ m \neq \tau_G$, which is a violation of the commutativity property of definition 3.20. □

Conversely, we can also prove the following.

Lemma 3.27. *Let $(\mathcal{G}, \mathcal{M})$ be a diagram and let R be an equivalence relation such that*

$$\forall n_1, n_2 \in V_{\uplus\mathcal{G}} \,.\, n_1 \sim n_2 \Longrightarrow n_1 \; R \; n_2. \tag{3.2}$$

Then, every graph that is isomorphic to $\uplus\mathcal{G} \;/\; R$ is a genuine type graph for $(\mathcal{G}, \mathcal{M})$.

Proof. Let $T = \uplus\mathcal{G} \;/\; R$ be a graph and let R be an equivalence relation such that (3.2) holds.

We define the typing morphisms $\tau_G : G \to T$ as $\tau_G(x) = [x]_R$, for all $G \in \mathcal{G}$ and for all $x \in G$. Then T is a valid type graph for $(\mathcal{G}, \mathcal{M})$ if for all morphisms $m : s \to t$ in $\mathcal{M}$ it holds that $\tau_t \circ m = \tau_s$. This holds if $[x]_R = [m(x)]_R$ for all x in $V_{\uplus\mathcal{G}} \cup E_{\uplus\mathcal{G}}$. From (3.2) it follows that, for all $x \in V_{\uplus\mathcal{G}}$,

$$x_1 \sim x_2 \Longrightarrow [x_1]_R = [x_2]_R$$

and $x \sim m(x)$ follows from the definition of $\sim$. Therefore, $[x]_R = [m(x)]_R$ for all $x \in V_{\uplus\mathcal{G}} \cup E_{\uplus\mathcal{G}}$, as is required for the proof. □

From this we can now give a proof for Theorem 3.25.

Proof of Theorem 3.25. Let T_p be the perfect type graph for $(\mathcal{G}, \mathcal{M})$. Furthermore, let T be an arbitrary type graph for $(\mathcal{G}, \mathcal{M})$ and let $T' \subseteq T$ be a genuine type graph defined as $\uplus\mathcal{G} \;/\; R$, for some equivalence relation R such that

$$\forall n_1, n_2 \in V_{\uplus\mathcal{G}} \,.\, n_1 \sim n_2 \Longrightarrow n_1 \; R \; n_2.$$

The existence of such T' follows from Lemma 3.26. Because $T' \subseteq T$, it trivially holds that if there exists a graph morphism $f : T_p \to T'$, there exists also a graph morphism $g : T_p \to T$.

Furthermore, let $r : \uplus\mathcal{G} \to T'$ be the morphism defined as $r(x) = [x]_R$. Then we can define $f : T_p \to T'$ as $f([x]_\sim) = r(x)$ for all x in $\uplus\mathcal{G}$. This morphism is well-defined because the characterisation of R states that $[x]_\sim \subseteq [x]_R$.

Since this morphism exists, there also exists a graph morphism from T_p into T, as stated before. From Definition 3.7 it then follows that T_p is stronger than T and since this holds for every T, the perfect type graph is the smallest type graph for a diagram. □

In the following example, the perfect type graph for a diagram consisting of two graphs and one morphism between these graphs is given.

Example 3.28. *Consider two graphs, G and H and a graph morphism m, as depicted in figure 3.6a. Furthermore, let $(\mathcal{G}, \mathcal{M}) = (\{G, H\}, \{m\})$. According to Theorem 3.25, the graph $\uplus\mathcal{G} \;/\; \sim$, as depicted in figure 3.6b, is the smallest type graph for $(\mathcal{G}, \mathcal{M})$. This type graph is illustrated in Figure 3.6b.*

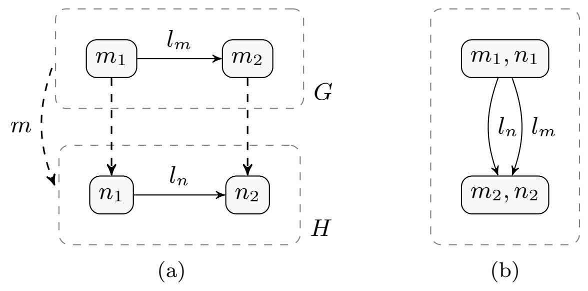


Figure 3.6: (a): Graphs G, H and morphism m. (b): The perfect type graph for $(\{G, H\}, \{m\})$. The text inside the nodes represents their node identities.

3.1.4 A Type Graph for a Graph Grammar

In the last section we defined a type graph for a diagram and claimed that all proofs we created for diagrams can be easily reused for graph grammars. This is possible due to the fact that a graph grammar can be seen as a diagram with some additional properties. In this section we shall therefore start with defining a transformation from graph grammars into diagrams; thereafter we define the perfect type graph for a graph grammar based on Definition 3.23.

In Section 2.4 we defined a graph grammar as a pair $GG = (G_0, \mathcal{P})$, consisting of a start graph G_0 and a set of production rules $\mathcal{P}$. Furthermore, we defined the language of GG as the set of production graphs of GG. We define a diagram for a graph grammar as follows:

Definition 3.29 (Diagram of a graph grammar)**.** *Let $GG = (G_0, \mathcal{P})$ be a graph grammar. Then, the diagram of GG is a pair $(\mathcal{G}, \mathcal{M})$ where $\mathcal{G}$ and $\mathcal{M}$ are the smallest sets such that:*

- $G_0 \in \mathcal{G}$
- *If $G \in \mathcal{G}$ and there exists a matching of a production rule $L \xrightarrow{p} R$, i.e. $\exists L \xrightarrow{p} R \in \mathcal{P} \,.\, m : L \rightarrow G$, then there exists G' such that $G \overset{m,p}{\Longrightarrow} G'$ is a graph transformation, and*
  - $p, m, p^*, m^* \in \mathcal{M}$
  - $L, R, G' \in \mathcal{G}$

Having now defined a diagram of a graph grammar, we can use the concepts from the previous section in order to define the perfect type graph for a graph grammar. As mentioned in Chapter 1, a type graph for a graph grammar should be defined such that it poses no restrictions on any derivation of the graph grammar. In this thesis we make sure that this property holds by requiring the following:

1. The left-hand side and right-hand side graphs of all production rules of the grammar must have a unique typing into this type graph.

2. All production graphs of the grammar must have a typing into the type graph.
3. The typing morphisms must commute with the morphisms p, m, p^*, and m^* in any of the graph transformations.

From this characterisation it follows that a graph T is a type graph for a graph grammar GG if, and only if, T is a type graph for the diagram of GG: this diagram contains exactly those graphs and morphisms as given in the characterisation above. Using this information, a type graph for a graph grammar can be formulated as follows:

Definition 3.30 (Type graph for a graph grammar)**.** *Let GG be a graph grammar and let $(\mathcal{G}, \mathcal{M})$ be its diagram. A type graph T is called a* type graph for *GG if T is a type graph for the diagram of GG.*

The same applies to the perfect type graph; we define the perfect type graph for a graph grammar based on the definition of a perfect type graph for a diagram.

Definition 3.31 (Perfect type graph for a graph grammar)**.** *Let GG be a graph grammar and let $(\mathcal{G}, \mathcal{M})$ be its diagram. Then the perfect type graph for GG is defined to be the perfect type graph for $(\mathcal{G}, \mathcal{M})$.*

We have now defined the perfect type graph for a graph grammar. This type graph is the most restrictive for the graph grammar, without imposing constraints on any of the derivations of this graph grammar; this follows directly from the proofs in the preceding section. We shall now define an algorithm that, given a graph grammar, computes a type graph that approaches this perfect type graph as much as possible.

3.2 Type Graph Inference

In the preceding section, we defined a perfect type graph for a graph grammar. The main problem that arises when trying to find an algorithm that finds this type graph is that the perfect type graph is defined using the entire set of production graphs of a graph grammar. Since this is an infinite set, we can not iterate over all of its elements and create the type graph accordingly.

Rather, we need an approach to find a type graph for a graph grammar based upon a finite set of elements. This is what this section aims at. We shall propose an algorithm that computes a type graph that approximates the perfect type graph for a graph grammar, but is based only on the start graph and rule graphs of this grammar, which constitute a finite set.

Unfortunately, it turns out to be impossible to find the perfect type graph for a graph grammar entirely based on its start graph and rule graphs, since the applicability of production rules is undecidable. Let us make this more precise.

Applicability of production rules is defined with respect to graphs. If a production rule is applicable with respect to a graph, it has a matching into this graph. In this thesis we shall also refer to applicability of a production rule in a graph grammar, meaning that this production rule is applicable in some production graph of the graph grammar. In order to prove that the application of production rules in graph grammars is undecidable, we shall now prove that the halting problem for Turing machines can be reduced to the applicability of production rules in a graph grammar.

Theorem 3.32. *There is no algorithm that determines whether an arbitrary pruduction rule is applicable in a graph grammar.*

Proof. Consider a Turing machine and reduce this Turing machine to a graph grammar such that all state-symbol combinations are represented by a production rule in this graph grammar. It is well-known that graph grammars are Turing complete; therefore such reduction is possible. We will not give it here in order to avoid unnecessary technical details. Furthermore, introduce production rules for all state-symbol combinations that are not in the transition table of the Turing machine.

Now suppose that one of these extra production rules becomes applicable in any derivation of the graph grammar. This would mean that the Turing machine enters a state-symbol combination for which it has no entry in its transition table and therefore halts. This has been proven undecidable and therefore the applicability of the production rule is also undecidable. □

Having proved this, we can now prove that computing the perfect type graph in an algorithmic way is impossible. Intuitively this is because the perfect type graph contains a typing for exactly all graphs that can be produced by the graph grammar. Since the applicability of production rules in a graph grammar is undecidable, as proved in Theorem 3.32, computing this exact set of producible graphs is impossible in a finite amount of time and therefore the perfect type graph can not be computed by any algorithm in general.

Lemma 3.33. *Computing the perfect type graph for a graph grammar is impossible in general.*

Proof. Let $GG = (G_0, \mathcal{P})$ be an arbitrary graph grammar and let $GG' = (G_0, \mathcal{P}')$ be an extension of GG, such that a disconnected node with a uniquely labelled self-edge is added to the right-hand side graphs of all production rules (we call this label u_p for production rule p). This extendsion yields a unique identification for all applied production rules while the applicability of the rules is not modified in any way.

Let $p \in \mathcal{P}'$ be an arbitrary production rule of GG' and let T be the perfect type graph for GG'. Then T either must or must not contain an edge labelled u_p, depending on whether p is applicable in GG; this is undecidable according to Theorem 3.32.

This means that if T contains a an edge labelled u_p while p is not applicable in GG, T is not genuine—since it contains a spurious type—and hence is not the pefect type graph, according to Lemma 3.24. On the other hand, if T does not contain any edge labelled u_p while p is applicable in GG, the type graph does not contain a typing for all elements in $\mathcal{L}_{GG'}$ and hence is not a valid type graph for GG', according to Definition 3.3.

Algorithm 3.1: Naive type inference algorithm

```
   Input: $GG = (G_0, \mathcal{P})$
   Output: A type graph for $GG$
1  $T \leftarrow G_0 \uplus \{L_r \mid r \in \mathcal{P}\} \uplus \{R_r \mid r \in \mathcal{P}\}$;
2  $Equiv \leftarrow \{(v, v') \mid \exists L \xrightarrow{p} R \in \mathcal{P} \,.\, p(v) = v'\}$;
3  $T \leftarrow T \,/\, Equiv$;
4  $T' = \varnothing$;
5  repeat
6      $T' \leftarrow T$;
7      $Equiv \leftarrow \varnothing$;
8      forall $L \xrightarrow{p} R \in \mathcal{P}$ do
9          $\mathcal{M} \leftarrow \{m \mid m : L \to T\}$;
10         forall $v, v' \in V_T$ do
11             if $\exists m \in \mathcal{M} \,.\, m(v) = v'$ then
12                 $Equiv \leftarrow Equiv \cup \{(\tau_L(v), v')\}$;
13             end
14         end
15     end
16     $T \leftarrow T \,/\, Equiv$;
17 until $|T| = |T'|$ ;
18 return $T$;
```

This proves that it is impossible to compute the perfect type graph for GG'. Because the reduction does not affect the applicability of production rules, this also proves that computing the perfect type graph for GG is impossible, as required. □

Although the perfect type graph can not be computed by any algorithm, the following sections will present two algorithms that compute type graphs given a graph grammar, approximating the perfect type graph. The first algorithm we present, which we shall call the *naive algorithm*, assumes that all production rules are eventually applicable and uses information from all production rules plus the start graph of a graph grammar to compute a type graph. The main drawback of this algorithm is that it may contain many spurious elements if some of the production rules in the grammar are not applicable in the grammar. Therefore, we introduce an *improved algorithm* in which some production rules that are certainly not applicable are omitted, yielding a type graph that is at least as small as the type graph computed by the naive algorithm.

3.2.1 Naive Algorithm

The naive algorithm is outlined in Algorithm 3.1. It accepts a graph grammar as its input and produces a type graph for this grammar. As specified at line 1 of the algorithm, it starts with an initial type graph that is the disjoint union of the start graph and all rule graphs of the provided graph grammar. Then it merges the left hand side and right hand side graphs based on the rule morphisms on lines 2 and 3. Thereafter, it repeatedly merges nodes of this type graph using the quotient construction for graphs, until no more merges are necessary; the type graph that has been computed by the algorithm until this point is the resulting type graph of the algorithm.

The merging of nodes is executed by three nested loops. The outermost loop, starting at line 5 of the algorithm, continually checks whether any of the nodes of the current type graph need to be merged. If no more merges are necessary, the loop terminates, and so does the algorithm.

In one iteration of this outer loop, first a binary relation, called *Equiv*, is created. This happens in the middle loop, starting at line 8 of the algorithm. This loop iterates over all production rules and computes all matchings from these rules into the current type graph. When a rule has matchings into the current type graph, the innermost loop, ranging from lines 10 until 14, adds information about what nodes need to be merged to *Equiv*.

After the middle loop has finished, the algorithm closes *Equiv* reflexively and transitively and then uses it for merging nodes using the quotient construction for graphs, as specified at line 16 of the algorithm. The resulting type graph is then used as the current type graph in the next iteration of the outer loop.

In order to omit unnecessary details, we do not explicitly define τ_L in the algorithm, although it is used at line 12. We implicitly assume that $\tau_L(v)$ maps a node $v \in V_L$ onto the node in the current type graph T that contains v as an element. However, because T is the product of multiple invocations of the quotient operation, this is not simply the equivalence class of v in T. Actually it is the equivalence class of v in some T', where T' is a flattened version of T.

Evaluation

Having identified an algorithm that computes a type graph for a graph grammar, we shall now evaluate whether it is *correct* with respect to the type graph it computes and how it computes this type graph.

First we examine whether the algorithm itself behaves correctly, i.e. if the algorithm is confluent and eventually terminates, for any input; thereafter we shall prove that the algorithm computes a valid type graph for a given graph grammar.

Confluence is a concept often used with term rewriting systems, as discussed in, e.g. [Klo92]. If a term rewriting system is confluent, this means that elements in such a rewriting system can be rewritten in more than one way, yielding the same result. Termination, on the other hand, means that there exist no infinite rewriting sequences.

Confluence of Algorithm 3.1 is straightforward, since the construction of *Equiv* and the quotient of T over the transitive-reflexive closure of *Equiv* are executed independently and because the ordering of the production rules only affects in what order the tuples of nodes to be merged are added to *Equiv*, the order in which production rules are processed does not affect the resulting type graph. Therefore, regardless of the order in which the production rules of GG are processed, the algorithm always computes the same type graph for a given graph grammar.

Furthermore, it is also straightforward that the algorithm always terminates because the algorithm starts with initialising the type graph based on a finite amount of production rules and subsequently only merges nodes using the quotient construction. Since the number of nodes is finite, it is straightforward that this merging eventually terminates: in every iteration of the loop ranging from line 5 until 17 the number of nodes decreases; therefore the algorithm will either eventually end up with one node and then terminate, or it will terminate earlier.

Now we shall prove that the algorithm computes a valid type graph from a graph grammar. According to Definition 3.20 the produced type graph should fulfill two properties: first, all production graphs of the grammar must be instances of this type graph and all typing morphisms from these instances into the type graph must commute with all morphisms between the graphs in the grammar.

We shall inductively prove these properties, but first we prove that given a host graph G and a production rule $L \xrightarrow{r} R$ such that there exist typing morphisms $\tau_G : G \to T$ and $\tau_L : L \to T$ into some type graph T, then there also exists a typing morphism $\tau_H : H \to T$, where H is the product of graph transformation $G \overset{r,m}{\Longrightarrow} H$.

Lemma 3.34. *Given a graph G and a production rule $L \xrightarrow{p} R$, let H be the result of the graph transformation $G \overset{r,m}{\Longrightarrow} H$ for matching $m : L \to G$, and let $p^* : G \to H$ and $m^* : R \to H$ be the morphisms in the pushout, as specified in Definition 2.26.*

Furthermore, let T be a type graph and let $\tau_G : G \to T$ and $\tau_R : R \to T$ be two typings such that $\tau_G \circ m = \tau_R \circ p$. Then, there exists a unique typing morphism $\tau_H : H \to T$ such that $\tau_G = \tau_H \circ p^*$ and $\tau_R = \tau_H \circ m^*$; in other words, such that the following diagram commutes:

$$\begin{tikzcd}
L \arrow[r, "p"] \arrow[d, "m"'] & R \arrow[d, "m^*"'] \arrow[ddr, bend left, "\tau_R"] & \\
G \arrow[r, "p^*"] \arrow[drr, bend right, "\tau_G"'] & H \arrow[dr, "\tau_H"] & \\
 & & T
\end{tikzcd}$$

This is precisely the pushout property, which has been proved in many works on algebraic graph transformations (see, e.g. [EEPT06]). We shall give a proof though, because in Chapter 4 we will use a slightly different property.

Proof. We define τ_H as follows:

– $\tau_H(v) = \begin{cases} \tau_G(v') & \textit{if } v = p^*(v') \\ \tau_R(v') & \textit{if } v = m^*(v') \end{cases}$ for nodes of H, and

– $\tau_H(e) = \begin{cases} \tau_G(e') & \textit{if } e = p^*(e') \\ \tau_R(e') & \textit{if } e = m^*(e') \end{cases}$ for edges of H.

First we prove that τ_H is well-defined, i.e. functions τ_H, V and τ_E must be well-defined. This holds if $\tau_G(x') = \tau_R(x'')$, for all $x \in V_H \cup E_H$ such that $x = p^*(x') = m^*(x'')$.

This holds because x' and x'' have the same preimage in L, meaning that there exists an $l \in L$ such that $x' = m(l)$ and $x'' = p(l)$. Then

$$\tau_G(x') = \tau_G(m(e)) = \tau_R(p(e)) = \tau_R(x'').$$

Second, we prove that τ_H is a graph morphism. As specified in Definition 2.15 this holds if, for all $e \in E_H$, the following holds:

- $\tau_H(src(e)) = src(\tau_H(e))$,
- $\tau_H(tgt(e)) = tgt(\tau_H(e))$, and
- $\tau_H(lbl(e)) = lbl(e)$.

The third property, $\tau_H(lbl(e)) = lbl(e)$, follows trivially from the definition of τ_H. For $src(e)$ and $tgt(e)$ two distinctive cases can be identified:

1. Let $e \in \operatorname{im} p^*$ and $src(e) \notin \operatorname{im} m^*$. Then there exists a unique typing for e and $src(e)$, defined as $\tau_H(e) = \tau_G(e')$ and $\tau_H(src(e)) = \tau_G(src(e'))$. Because τ_G is a graph morphism, we prove the required property as follows:
$$\begin{aligned}\tau_H(src(e)) &= \tau_G(src(e')) \\ &= src(\tau_G(e')) \\ &= src(\tau_H(e))\end{aligned}$$
2. Let $e \in \operatorname{im} p^*$ and $src(e) \in \operatorname{im} m^*$. Then there exists a $v \in \operatorname{dom} m^*$ and a $v' \in \operatorname{dom} p^*$ such that $m^*(v) = p^*(v')$. In this case, v and v' have the same preimage in L. From $\tau_G \circ m = \tau_R \circ p$ it then follows that $\tau_G(v') = \tau_R(v) = \tau_H(src(e))$. Because τ_H is a graph morphism, the required property holds in this case.
3. If $e \in \operatorname{im} m^*$, the proofs are analogous to those for $\operatorname{im} p^*$, as given in (1) and (2).

Since the proof for $tgt(e)$ is analogous to the proof for $src(e)$, this proves that there exists a typing morphism $\tau_H : M \to T$ for any graph H resulting from graph transformation $G \overset{r,m}{\Longrightarrow} H$.

Finally we prove $\tau_G = \tau_H \circ p^*$ and $\tau_R = \tau_H \circ m^*$. This follows directly from the definition of τ_H:

$$\begin{aligned}\tau_H(p^*(e')) &= \tau_G(e')|_{\operatorname{dom} p^*} \\ \tau_H(m^*(e')) &= \tau_R(e')|_{\operatorname{dom} m^*}.\end{aligned}$$ □

Intuitively, all nodes and edges of H, the result graph of the graph transformation, either originate from G, R, or both. Commutativity of the pushout diagram, as used in graph transformations, ensures that all nodes and edges, regardless where these originate from, always have the same node- and edge types.

Using the information in Lemma 3.34 we shall now prove that Algorithm 3.1 produces a type graph for a given graph grammar.

Theorem 3.35. *Let $GG = (G_0, \mathcal{P})$ be a graph grammar and let $(\mathcal{G}, \mathcal{M})$ be the diagram of GG. Then, the type graph induced from GG by Algorithm 3.1 is a type graph for GG.*

Proof. We prove that the type graph T at line 24 of the algorithm is a type graph for $(\mathcal{G}, \mathcal{M})$.

Let $D_0, \dots, D_i, \dots$ be an infinite sequence of diagrams, inductively defined as follows:

- $D_0 = (\mathcal{G}_0, \mathcal{M}_0)$
  - $\mathcal{G}_0 = \{G_0\} \cup \{L_r \mid r \in \mathcal{P}\} \cup \{R_r \mid r \in \mathcal{P}\}$
  - $\mathcal{M}_0 = \{p \mid L \xrightarrow{p} R \in \mathcal{P}\}$
- $D_i = (\mathcal{G}_i, \mathcal{M}_i)$
  - $\mathcal{G}_i = \mathcal{G}_{i-1} \cup \{H \mid \exists r \in \mathcal{P} \,.\, \exists G \in \mathcal{G}_{i-1} \,.\, G \overset{r,m}{\Longrightarrow} H\}$
  - $\mathcal{M}_i = \mathcal{M}_{i-1} \cup$

$$\left\{ m, m^*, p^* \mid \exists r = L \xrightarrow{p} R \in \mathcal{P}, \exists G \in \mathcal{G}_{i-1} \,.\, \begin{tikzcd} L \arrow[r, "p"] \arrow[d, "m"] & R \arrow[d, "m^*"] \\ G \arrow[r, "p^*"] & H \end{tikzcd} \right\}$$

Let $T_0, \dots, T_i, \dots$ be an infinite sequence of type graphs, where T_0 is the intermediate type graph at line 10 of Algorithm 3.1 before the first iteration of the loop ranging from line 5 until 17, and T_i $(i > 0)$ is the intermediate type graph at line 17 of Algorithm 3.1 after the i^{th} iteration of this loop.

We shall now inductively prove that T_i is a type graph for D_i, for all $i \geq 0$.

Basis step:

We prove that T_0 is a type graph for D_0, i.e.

(a) $\forall G \in \mathcal{G}_0 \,.\, \exists \tau_G : G \to T_0$
(b) $\forall m : G \to H \in \mathcal{M}_0 \,.\, \tau_H \circ m = \tau_G$

To prove (a), we prove that all $G \in \{G_0\} \uplus \{L_r \mid r \in \mathcal{P}\} \uplus \{R_r \mid r \in \mathcal{P}\}$ have a typing in T_0. Since the algorithm initialises the type graph to the disjoint union of exact this set of graphs, and the only changes to T are performed at line 3 of the algorithm using the quotient construction, it follows from Lemma 2.22 that (a) holds.

To prove (b), we prove that $\tau_R \circ p = \tau_L$ for all $p = \{p \mid L \xrightarrow{p} R \in \mathcal{P}\}$. This follows directly from lines 2 and 3 of the algorithm: for all these morphisms it merges all elements in $\operatorname{dom} p$ with their images under p.

Induction hypothesis:

T_i is a type graph for D_i, i.e.

(a) $\forall G \in \mathcal{G}_i, \,.\, \exists \tau_G : G \to T_i$
(b) $\forall m : G \to H \in \mathcal{M}_i \,.\, \tau_H \circ m = \tau_G$

Induction step:

We prove that T_{i+1} is a type graph for D_{i+1}, i.e.

(a) $\forall G \in \mathcal{G}_{i+1} \,.\, \exists \tau'_G : G \to T_{i+1}$
(b) $\forall m : G \to H \in \mathcal{M}_{i+1} \,.\, \tau'_H \circ m = \tau'_G$

Let $\alpha : T_i \to T_{i+1}$ be a total graph morphism. Existence of such α follows directly from Lemma 2.21.
First we prove (a). Let $G \in \mathcal{G}_{i+1}$ be a graph. If $G \in \mathcal{G}_i$, there exists $\tau'_G : G \to T_{i+1}$, defined as $\tau'_G = \alpha \circ \tau_G$.
If $G \notin \mathcal{G}_i$, there exists $G' \in \mathcal{G}_i$ such that $G' \overset{r,m}{\Longrightarrow} G$, for some $r = (L \xrightarrow{p} R) \in \mathcal{P}$ and some $m : L \to G'$. According to Lemma 3.34, τ'_G exists in this case if there exist $\tau'_{G'} : G' \to T_{i+1}$ and $\tau'_R : R \to T_{i+1}$ such that $\tau'_{G'} \circ m = \tau'_R \circ p$.
Since $G', R \in \mathcal{G}_i$, $\tau'_{G'}$ and τ'_R exist; these morphisms are defined as $\tau'_{G'} = \alpha \circ \tau_{G'}$ and $\tau'_R = \alpha \circ \tau_R$. Since $\tau_{G'} \circ m = \tau_R \circ p$ holds—this follows from the induction hypothesis—we have $\alpha \circ \tau_{G'} \circ m = \alpha \circ \tau_R \circ p$ and hence the required property $\tau'_{G'} \circ m = \tau'_R \circ p$. (a) then follows directly from Lemma 3.34.
Next we prove (b). Let $m : G \to H \in \mathcal{M}_{i+1}$ be a graph morphism. If $m \in \mathcal{M}_i$, then $\tau_H \circ m = \tau_G$ follows from the induction hypothesis.
If $m \notin \mathcal{M}_i$, then m is a match, co-match, or co-production that gave rise to the construction of some $G \in \mathcal{G}_i \setminus \mathcal{G}_{i-1}$. For the match we have proved the required property in (a). For the co-match and co-production this property follows directly from Lemma 3.34.

This proves that T_i is a type graph for D_i, for all $i > 0$. The algorithm terminates when $T_{i+1} = T_i$. Because of this, T_i is a type graph for D_{i+1} and hence for all D_i for $i \to \infty$. Since D_i converges to $(\mathcal{G}, \mathcal{M})$ if $i \to \infty$, T_i is a type graph for $(\mathcal{G}, \mathcal{M})$, as required. □

3.2.2 Improved algorithm

So far, we have defined a perfect type graph and have outlined an algorithm that calculates a type graph for a graph grammar. As stated before, the main drawback of this algorithm is that it uses all production rules of a graph grammar for creating a type graph instead of those that are applicable; this may result in a type graph with many spurious elements.

In this section we shall give an improved type inference algorithm in which some of the non-applicable production rules are omitted, in order to reduce this number of spurious elements. Although applicability of production rules in a graph grammar is undecidable in general, it is possible to determine some production rules that certainly will never become applicable. A small example, where the non-applicability of a production rule is obvious, is as follows.

Example 3.36. *Figure 3.7 represents a grammar consisting of a start graph (G_0, see Fig. 3.7a) and one production rule (p, see Fig. 3.7b). Nodes in this figure are identified using their node labels. It is obvious that p is never applicable in this graph grammar and hence the language of the grammar consist only of graph G_0. Therefore, the perfect type graph for the grammar is the type graph illustrated in Figure 3.8a; the type graph computed by Algorithm 3.1 is illustrated in Figure 3.8b.*

In general it holds that production rules for which the left hand side graphs do not have a typing into the type graph for a graph grammar will not be

Algorithm 3.2: Improved type inference algorithm

```
Input: $GG = (G_0, \mathcal{P})$
Output: A type graph for $GG$
1  $Processed \leftarrow \varnothing$;
2  $T \leftarrow G_0$;
3  $T' \leftarrow \varnothing$;
4  repeat
5      $T' \leftarrow T$;
6      $Equiv \leftarrow \varnothing$;
7      forall $L \xrightarrow{p} R \in \mathcal{P} . \exists m : L \rightarrow T$ do
8          if $p \notin Processed$ then
9              $Equiv' \leftarrow \{(v, v') \mid \exists L \xrightarrow{p} R \in \mathcal{P} . p(v) = v'\}$;
10             $T \leftarrow T \uplus ((L \uplus R) \,/\, Equiv')$;
11             $Processed \leftarrow Processed \cup \{p\}$;
12         end
13         $\mathcal{M} \leftarrow \{m \mid m : L \rightarrow T\}$;
14         forall $v, v' \in V_T$ do
15             if $\exists m \in \mathcal{M} . m(v) = v'$ then
16                 $Equiv \leftarrow Equiv \cup \{(\tau_L(v), v')\}$;
17             end
18         end
19     end
20     $T \leftarrow T \,/\, Equiv$;
21 until $|T| = |T'|$ ;
22 return $T$;
```

applicable in any derivation of the graph grammar. This follows directly from Lemma 2.22: since both typing morphisms and matchings are total morphisms, it follows that, given a production rule r, a type graph T, and a graph $G \in \mathcal{I}_T$, if there exists a matching $m : L_r \rightarrow G$, there also exists a typing $\tau_L : L_r \rightarrow T$.

We shall use this information in this section and shall define an algorithm that ignores production rules that do not have a typing into the type graph produced up to that point. An algorithm for finding this improved type graph is given in Algorithm 3.2.

This algorithm works analogously to Algorithm 3.1, with the exception that

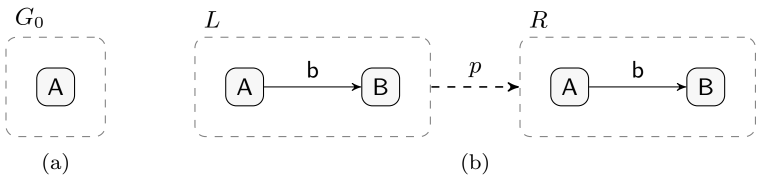


Figure 3.7: Example graph grammar consisting of a start graph (a) and one production rule that will not be applicable (b).

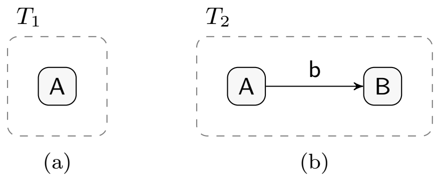


Figure 3.8: Type graphs for the graph grammar illustrated in Figure 3.7. (a) illustrates the perfect type graph for this grammar; (b) is the type graph for this grammar, as computed by Algorithm 3.1.

the improved algorithm starts with an initial type graph that contains typing information for only the start graph of the grammar, as specified in line 2 of the algorithm. Information from the right-hand side graph of a production rule is now added only if the left-hand side graph of the concerning production rule has a matching into the current type graph. This is specified at lines 8 until 12 of the improved algorithm.

Because the information from the right-hand side of a production rule only needs to be added once to the type graph, the algorithm introduces a set of production rules, called *Processed* for which this information is already added to the type graph.

Except this alternate strategy of when to add information from right-hand side graphs to the type graph, the algorithm is identical to the naive algorithm. Therefore, most proofs used for this naive algorithm can be reused to prove properties of the improved algorithm, as we shall do in the next section.

Evaluation

As we did for the naive algorithm in Section 3.2.1, we shall now prove that the improved algorithm behaves correctly and whether it computes a valid type graph given a graph grammar.

We first verify whether Algorithm 3.2 is confluent and whether it terminates. For confluence we can use a reasoning analogous to the one used for the naive algorithm. Each iteration of the outer loop that starts at line 4 ranges until line 21 of the improved algorithm, basically does two things.

First, it iterates over all production rules and constructs a binary relation *Equiv* based on which nodes of the current type graph will later be merged. This happens in the middle loop, specified at lines 7 until 19 of the algorithm. If a rule that has a matching into the current type graph has not been processed yet, the right-hand side graph of the rule is added to the current type graph. Because this middle loop only adds elements to the type graph and constructs relation *Equiv* without merging any nodes yet, the order in which production rules are processed does not affect the resulting type graph.

After this middle loop is executed, the algorithm computes the transitive-reflexive closure of *Equiv* and merges nodes of the current type graph using the quotient construction for graphs at line 20. Therefore, regardless of the order in which the production rules of the given graph grammar are processed, the algorithm always computes the same type graph for a given graph grammar.

Termination of Algorithm 3.2 is also straightforward since the algorithm basically does two things: it adds the right-hand side graphs of a finite amount of production rules to the type graph and thereafter only merges nodes using the quotient construction for graphs. Since the number of nodes is finite, it is straightforward that this merging eventually terminates: in every iteration of the loop ranging from line 4 until 21 the number of nodes decreases; therefore the algorithm will either eventually end up with one node and then terminate, or it will terminate earlier.

We shall now inductively prove the validity of the type graph computed by the algorithm. Because of the similarity of Algorithm 3.1 and Algorithm 3.2, we can reuse the proof of Theorem 3.35 for proving correctness of the improved algorithm.

Theorem 3.37. *Let $GG = (G_0, \mathcal{P})$ be a graph grammar and let $(\mathcal{G}, \mathcal{M})$ be the diagram of GG. Then, the type graph induced from GG by Algorithm 3.2 is a valid type graph for GG.*

Proof. We prove that the type graph T at line 24 of the algorithm is a type graph for $(\mathcal{G}, \mathcal{M})$.

Let $D_0, \ldots, D_i, \ldots$ be an infinite sequence of diagrams, inductively defined as follows:

- $D_0 = (\mathcal{G}_0, \mathcal{M}_0)$
  - $\mathcal{G}_0 = \{G_0\}$
  - $\mathcal{M}_0 = \varnothing$
- $D_i = (\mathcal{G}_i, \mathcal{M}_i)$
  - $\mathcal{G}_i = \mathcal{G}_{i-1} \cup \{H, L_r, R_r \mid \exists r \in \mathcal{P} \,.\, \exists G \in \mathcal{G}_{i-1} \,.\, G \overset{r,m}{\Longrightarrow} H\}$
  - $\mathcal{M}_i = \mathcal{M}_{i-1} \cup$

$$\left\{ m, p, m^*, p^* \mid \exists r = L \xrightarrow{p} R \in \mathcal{P}, \exists G \in \mathcal{G}_{i-1} \,.\, \begin{tikzcd} L \arrow[r, "p"] \arrow[d, "m"] & R \arrow[d, "m^*"] \\ G \arrow[r, "p^*"] & H \end{tikzcd} \right\}$$

Let $T_0, \ldots, T_i, \ldots$ be an infinite sequence of type graphs, where T_0 is the intermediate type graph at line 4 of Algorithm 3.2 before the first iteration of the loop ranging from line 4 until 21, and T_i (for $i > 0$) is the intermediate type graph at line 21 of Algorithm 3.2 after the i^{th} iteration of this loop.

We shall now inductively prove that T_i is a type graph for D_i, for all $i \geq 0$.

Basis step:

We prove that T_0 is a type graph for D_0, i.e.

(a) $\forall G \in \mathcal{G}_0 \,.\, \exists \tau_G : G \to T_0$

(b) $\forall m : G \to H \in \mathcal{M}_0 \,.\, \tau_H \circ m = \tau_G$

To prove (a), we prove that G_0 has a typing in T_0. Since the algorithm initialises the type graph to exactly this graph and the type graph remains invariant until line 4, we may conclude that (a) holds.
(b) is trivial, because $\mathcal{M}_0 = \varnothing$.

Induction hypothesis:

T_i is a type graph for D_i, i.e.

(a) $\forall G \in \mathcal{G}_i, . \exists \tau_G : G \to T_i$
(b) $\forall m : G \to H \in \mathcal{M}_i \,.\, \tau_H \circ m = \tau_G$

Induction step:

We prove that T_{i+1} is a type graph for D_{i+1}, i.e.

(a) $\forall G \in \mathcal{G}_{i+1} \,.\, \exists \tau'_G : G \to T_{i+1}$
(b) $\forall m : G \to H \in \mathcal{M}_{i+1} \,.\, \tau'_H \circ m = \tau'_G$

Let $\alpha : T_i \to T_{i+1}$ be a total graph morphism. Existence of such α follows directly from Lemma 2.21.
First we prove (a). Let $G \in \mathcal{G}_{i+1}$ be a graph. If $G \in \mathcal{G}_i$, there exists $\tau'_G : G \to T_{i+1}$, defined as $\tau'_G = \alpha \circ \tau_G$.
If $G \notin \mathcal{G}_i$, then $G \in \{H, L_r, R_r \mid \exists r \in \mathcal{P} \,.\, \exists G \in \mathcal{G}_{i-1} \,.\, G \overset{r,m}{\Longrightarrow} H\}$. For the existence of τ'_G in this case, we first need to prove that $r \in$ *Processed* at line 21 of the algorithm, after the $i+1^{th}$ iteration of the loop; *Processed* is the set *Processed* used in the algorithm. Since the defintion of G states that r gives rise to graph transformation $G \overset{r,m}{\Longrightarrow} H$, r has a matching in some $G' \in \mathcal{G}_{i-1}$. From Lemma 2.22 it then follows that r also has a matching m_r into T_{i-1}; the matching m'_r from r into T_i is then defined as $\alpha \circ m_r$. Because m'_r exists, r is added to *Processed* at line 11 of the algorithm and hence $r \in$ *Processed* at line 21 of the algorithm. The proof that G has a typing in T_{i+1} in this case is given in the proof of Theorem 3.35 and is therefore omitted here.
Next we prove (b). Let $m : G \to H \in \mathcal{M}_{i+1}$ be a graph morphism. If $m \in \mathcal{M}_i$, then $\tau_H \circ m = \tau_G$ follows from the induction hypothesis.
If $m \notin \mathcal{M}_i$, then m is a match, production (rule morphism), co-match, or co-production (m, p, m^*, p^* resp.) that gave rise to the construction of some $G \in \mathcal{G}_i \backslash \mathcal{G}_{i-1}$. For the match we have proved the required property in (a). For the rule morphism this follows from lines 10–12 of the algorithm: all nodes $v, v' \in V_{T_i}$ such that $p(v) = v'$ are added to *Equiv* and hence merged at line 23 of the algorithm and hence $\tau_L(v) = \tau_R(p(v))$ holds for all these nodes. For the co-match and co-production, the required property property follows directly from Lemma 3.34.

Furthermore, we need to prove that the set *Processed* eventually contains all production rules that are applicable in *GG*. To prove this, suppose that the algorithm terminates after n iterations of the loop and that there exists a production rule $r \notin$ *Processed* that would have been added after $n + j$ iterations, for some $j > 0$.

Because the algorithm terminates, there exist no $p \in$ *Processed* that yield changes to T_n. According to Lemma 2.22, this means that there also exist no $q \in$ *Processed* that influences the applicability of any production rule in $\mathcal{P}$. This means that r will not become applicable and hence will not be added to *Processed*, which is a contradiction.

This proves that T_i is a type graph for D_i, for all $i > 0$. The algorithm terminates when $T_{i+1} = T_i$. Because of this, T_i is a type graph for D_{i+1} and hence for all D_i for $i \to \infty$. Since D_i converges to $(\mathcal{G}, \mathcal{M})$ if $i \to \infty$, T_i is a type graph for $(\mathcal{G}, \mathcal{M})$, as required. □

This concludes our evaluation of the improved algorithm. First we have shown that the algorithm is both confluent and terminating; thereafter we proved that the algorithm computes a valid type graph for a given graph grammar. The next section will give a comparison of both algorithms, illustrating why the improved algorithm computes a smaller type graph than the naive algorithm.

3.2.3 A comparison of the algorithms

An interesting question that arises is whether the type graph computed by Algorithm 3.2 is always smaller than the type graph computed by algorithm 3.1. Although we suppose that this is always the case, but because of the limited amount of time for this project, we did not manage to formally prove this; instead we shall give a possible proof strategy and give an intuitive explanation for why the improved algorithm computes smaller type graphs than the naive algorithm for the same graph grammars.

To improve readability we shall denote algorithm 3.1 as algorithm A and Algorithm 3.2 as algorithm B. Likewise, we shall denote the type graphs computed by algorithm A and B as T_A and T_B, respectively.

As we have seen in Theorem 3.35 and Theorem 3.37, both algorithms A and B compute an infinite sequence of intermediate type graphs, which we shall call $T_A^0, T_A^1, \ldots, T_A^i, \ldots$ and $T_B^0, T_B^1, \ldots, T_B^i, \ldots$ respectively. We define T_A^i as follows, for all $i > 0$:

- $T_A^0 = G_0 \uplus \{L_r \mid r \in \mathcal{P}\} \uplus \{R_r \mid r \in \mathcal{P}\}$
- $T_B^i = T_A^0 \;/\; R_i^*$, where R_i^* is the cumulative relation *Equiv*, i.e. the union of all relations *Equiv* in the algorithm, until the i^{th} iteration of the algorithm.

Likewise, we define T_B^i as follows, for all $i > 0$:

- $T_B^0 = G_0$
- $T_B^i = (T_B^{i-1} \uplus \{L_r, R_r \mid r \in \mathcal{P}, \exists m : L_r \to T_B^{i-1}\}) \;/\; Q_i$ for some equivalence relation Q_i.

We think that it is possible to write an inductive proof that proves that $T_B \leq T_A$, for all $i > 0$. The main reason for this intuition is that algorithm A has a typing for all production rules in $\mathcal{P}$, while algorithm B only adds the production rules that are in *Processed*. Furthermore, it is intuitive to see that all elements that are merged by algorithm B in step i are also merged by algorithm A in the same step or in earlier steps. Finally, note that all left-hand side graphs and right-hand side graphs that are added in T_B^{i+1} also have a typing in T_A^{i+1}; this follows directly from the proof of Theorem 3.35.

Thus we have given some ingredients for a possible proof and we think the required property can be proved using this strategy, but as we already stated we did not manage to write a complete proof in the amount of time we had during this project.

3.2.4 Application Conditions

Production rules are often accompanied by application conditions. Such conditions restrict the possible matchings for a production rule in the host graph in a graph transformation. The general idea is that a production rule that is accompanied by one or more application conditions matches into a host graph if, and only if, there exists a matching from the left-hand side of the rule into the host graph that satisfies all application conditions.

This illustrates the often negative nature of application conditions. Although both positive and negative application conditions exist, only negative application conditions make the graph transformations formalism more powerful [Wag95]. Therefore we shall only consider negative application conditions in this thesis, which are defined as follows.

Definition 3.38 (Negative application condition)**.** *A* negative application condition *N is a graph such that $n : L \to N$ is a graph morphism. A graph morphism $m : L \to G$ satisfies N, denoted $m \vDash N$, if there exists no graph morphism $f : N \to G$ such that $f \circ n = m$.*

As stated before, a production rule that is accompanied a set of negative application conditions $\mathcal{N}$ is only applicable if none of the conditions in $\mathcal{N}$ are satisfied; in other words, if there exist no graph morphisms $N_i \to G$ for any $N_i \in \mathcal{N}$, such that the diagram given in Figure 3.9 commutes.

$$\begin{tikzcd}
N_1 \arrow[rrdd] & & \\
N_2 \arrow[rrd] & & \\
\cdots & L \arrow[luu] \arrow[lu] \arrow[ld] \arrow[r, "m"] & G \\
N_n \arrow[rru] & &
\end{tikzcd}$$

Figure 3.9: A graph transformation with negative application conditions

Definition 3.39 (Graph production with application conditions)**.** *A* graph production with application conditions *is a pair $\hat{r} = (r : L \xrightarrow{p} R, \mathcal{N})$ consisting of a production rule r and a set of negative application conditions $\mathcal{N}$. $\hat{r}$ is applicable to a graph G via matching $m : L \to G$, yielding a graph transformation $G \overset{p,m}{\Longrightarrow} H$, if $m \nvDash N$ for all $N \in \mathcal{N}$.*

Application conditions give extra information about the applicability of production rules. However, although they give information about what elements should be absent at a certain moment, they do not state that these elements should never occur in the program.

Moreover, even though application conditions imply that certain elements should exist somewhere in some derivation of a graph grammar, these elements are introduced by a right-hand side of any rule already and hence are in the type graph already.

Nevertheless, even though not used in the type inference algorithm, application conditions can be used to detect some errors in a graph grammar, since if there exists no matching for an application condition into the induced type graph, the application condition is not necessary and may be erroneous.

3.3 Discussion

In this chapter we have defined a perfect type graph for a graph grammar and have introduced two algorithms that compute a type graph given a graph grammar, that is an approximation of the perfect type graph. Unfortunately, because the applicability of production rules is undecidable, it turned out to be impossible to compute the perfect type graph in an algorithmic way.

Although we did not find what we aimed for—an algorithm that computes the perfect type graph for a graph grammar—we managed to come up with an algorithm that produces a good type graph for the grammar. Using this computed type graph we are able to find several types of errors that would have been difficult to identify in an untyped setting. We shall briefly discuss some of them in this section.

First, the computed type graph represents a model of all graphs that can be produced by a graph grammar. Because type graphs are always finite structures and languages of graph grammars are infinite in general, type graphs give rise to a finite model for an infinite set of graphs. Additionally, since instance graphs may contain many nodes of one type while type graphs contain each node type only once, type graphs are generally small compared to their instances. Because of these properties, type graphs are able to model complex and possibly infinite structures in a synoptic way.

Particularly if type graphs are small, programmers often have some idea of what a type graph for a graph grammar should look like. Using this information, type graphs may give information about errors in graph grammars, such as erroneous production rules, if the computed type graph is different from the expected type graph.

Additionally, the improved algorithm, given in Algorithm 3.2, can detect some production rules that are never applicable. If at the end of the execution of this algorithm the set *Processed* is not equal to the set of all production rules of the grammar, this means that all production rules that are not in *Processed* do not have a typing into the type graph and, accordingly, are never applicable in any derivation of the graph grammar.

A similar detection of erroneous or superfluous application conditions is also provided by the improved algorithm. If any negative application condition does not have a typing into the type graph computed by the algorithm, there will never be a graph morphism from this application condition into any production graph of the graph grammar and therefore this application condition will never prevent the application of any production rule.

As a potential improvement, we expect that an algorithm that produces proper type graphs for given graph grammars may be advantageous over the algorithms presented in this chapter; these two algorithms produce non-proper type graphs in general, while we saw earlier in this chapter that proper type graphs are preferred over non-proper ones. As we did not consider such an algorithm that produces only proper type graphs, we consider it future work.

This concludes our discussion on type graphs defined as simple graphs, containing only nodes and edges. In Chapter 4 we shall extend the definition of type graphs by adding a special type of edges, called *inheritance edges*.

CHAPTER 4

Type Inheritance

Hitherto we have investigated ordinary type graphs, which distinguish only node types and edge types. In this chapter we will introduce the notion of type graphs with inheritance, in which a special kind of edges, called inheritance edges, are present. The basic idea for using type graphs with inheritance instead of ordinary type graph is to classify instance graphs using more compact representations by reducing redundancy [dLBE$^+$07]. Additionally, as is the case in the object-orientation paradigm, inheritance increases the reusability and extensibility of type graphs by combining information in so-called supertypes and reusing this information in subtypes, which inherit all information from the supertypes.

In this chapter we will use the principle of inheritance as it has been formalised for graph transformations by Bardohl et al [BEdLT04]. In this theory a new notion of typing is introduced, which is different from the typing morphisms we have seen in Chapter 3, describing a typing from an instance graph into a type graph with inheritance. We shall first elaborate on this theory in Section 4.1.

After this, we shall identify the properties of type graphs with inheritance for graph grammars and we will define a particular type graph with inheritance for such grammars, which we will call the naive type graph with inheritance. This naive type graph with inheritance is constructed in a similar way as the perfect type graph defined in Chapter 3, but here we choose a more liberal approach by not merging all equivalent node types, but by adding inheritance edges between these node types in some cases.

In Section 4.2 we will introduce an algorithm that computes a type graph with inheritance for a given graph grammar. As is the case with the algorithms defined in Chapter 3, this algorithm cannot compute the exact naive type graph with inheritance because it is defined based upon an infinite set of graphs. The algorithm will, however, construct its type graph in a similar way as the naive type graph with inheritance.

Although both the naive type graph with inheritance and the type graph with inheritance constructed by the algorithm are valid type graphs, we shall see that both type graphs distinguish too many node types because too few nodes are merged. This is what we try to solve in Section 4.3, by merging some nodes of the naive type graph with inheritance based on some general scenarios and, because of that, acquiring a denser type graph with inheritance.

Finally, in section 4.4, we conclude the chapter with a discussion in which all topics discussed in this chapter are briefly reviewed.

4.1 Type Graphs with Inheritance

In Section 3.1 we defined a type graph as a graph $T = (V_T, E_T)$ where V_T represents a set of node types and E_T a set of edge types. Furthermore we defined a typing from a graph G into T as a total graph morphism $\tau : G \to T$.

We shall now extend this notion of type graphs by introducing a special kind of edges, called *inheritance edges*, into type graphs. Inheritance edges are directed edges indicating an inheritance relation among node types: the source node of an inheritance edge is said to be a *subtype* of the target node, whereas the target node is called the *supertype* of the source node. Moreover, nodes in a type graph with inheritance can either be *abstract* or *concrete*; the difference between the two is that concrete node types can be instantiated, whereas abstract ones cannot.

The classical definition of a type graph with inheritance is given by Bardohl et al [BEdLT04] and reads as follows:

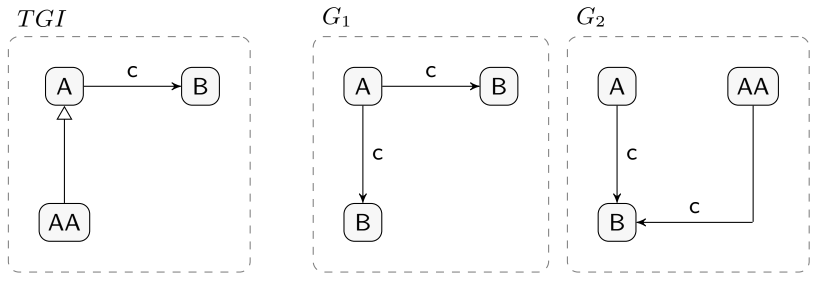
Definition 4.1 (Type graph with inheritance)**.** *A* type graph with inheritance *is a triple $TGI = (T, I, \mathcal{A})$ consisting of a type graph $T = (V_T, E_T)$, an acyclic inheritance relation $I \subseteq V_T \times V_T$, and a set of abstract nodes $\mathcal{A} \subseteq V_T$. For each $x \in V_T$, the* inheritance clan *is defined by $clan_I(x) = \{y \in V_T \mid (x, y) \in I^*\}$, where I^* is the* reflexive-transitive closure *of I.*

As is the case in object-oriented systems, a type inherits attributes from all its supertypes. With respect to type graphs with inheritance this means that, given two node types $v_1, v_2 \in V_T$ such that v_2 is a subtype of v_1, v_2 inherits all incoming and outgoing edges of v_1. Because of this, the inheritance gives rise to new instances.

Example 4.2. *Consider Figure 4.1a. It illustrates a type graph with inheritance TGI containing three nodes, labelled A, B, and AA, and two edges. One of these two edges, labelled c, is an ordinary edge, wehereas the open-ended arrow from AA to A represents an inheritance edge; this edge indicates that the node labelled AA is a subtype of the node labelled A. All node types in this example are concrete, we will explicitly mark abstract node types in the example graphs in this chapter, if the difference between concrete and abstract node types is significant (see, e.g. Figure 4.3c).*

Figure 4.1b illustrates two instances of TGI. Graph G_1 clearly is a legal instance of TGI, because there exists a typing morphism $\tau_{G_1} : G_1 \to TGI$. Although there does not exist a typing morphism $\tau_{G_2} : G_2 \to TGI$, graph G_2 also is a valid instance of TGI.

In order to specify a typing into a type graph with inheritance and to be able



(a) (b)

Figure 4.1: A type graph with inheritance and two instances.

to benefit from the theory of typed graph transformations without inheritance, we define a *closure* of type graphs with inheritance into ordinary type graphs. Intuitively, a closure of a type graph with inheritance is an ordinary type graph that contains the same information as this type graph with inheritance.

Definition 4.3 (Closure of a type graph with inheritance)**.** *Let $TGI = (T, I, \mathcal{A})$ be a type graph with inheritance with $T = (V_T, E_T)$. The* abstract closure *of TGI is the graph $\overline{TGI} = (V_T, \overline{E_T})$ with*

$$\overline{E_T} = \{(s, l, t) \mid \exists (s', l, t') \in E_T, s \in clan_I(s'), t \in clan_I(t')\}.$$

The concrete closure *of TGI is the graph $\widehat{TGI} = \overline{TGI}|_{V_T \setminus \mathcal{A}}$.*

The two closures of a type graph with inheritance gives rise to two different type graphs: the abstract closure contains both concrete and abstract types and the concrete closure contains only concrete types. Using this distinction, also instances of a type graph with inheritance can either be abstract or concrete.

Definition 4.4 (Instance of a type graph with inheritance)**.** *Let TGI be a type graph with inheritance. An* abstract instance *of TGI is an instance graph of $\overline{TGI}$; a* concrete instance *of TGI is an instance graph of $\widehat{TGI}$.*

As stated before, only concrete types can be instantiated. Moreover, Bardohl et al. [BEdLT04] remark that production graphs of a graph grammar are typed over the concrete closure, whereas rule graphs are typed over the abstract closure in general. A comparable situation holds for object-oriented systems, as is illustrated by the following example.

Example 4.5. *Many object-oriented systems discriminate between abstract and concrete types. Since only concrete types can be instantiated, all objects in such systems are concrete. Methods, however, are often specified using abstract types to indicate that their input parameters and return variables must at least be subtypes of these abstract types. Therefore, a method that is defined for input parameters of an abstract type is actually defined for parameters of all types inheriting from that abstract type.*

In graph grammars typing works in a similar way. In analogy to object-oriented systems, production graphs represent objects, and methods are represented by

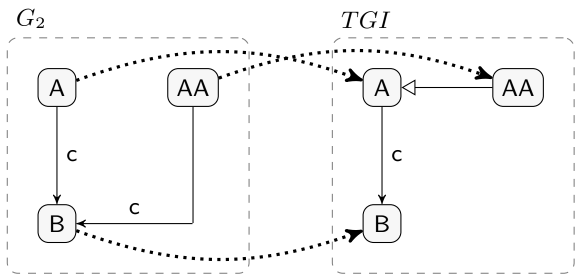


Figure 4.2: Example of a clan morphism

production rules: left-hand side graphs define the input parameters of these methods and right-hand side graphs represent their return values. Following this analogy, production graphs in a graph grammar contain only nodes of concrete types, while rule graphs may contain nodes of abstract types as well.

A typing of an instance graph into a type graph with inheritance uses the construction of the closure, as specified in Definition 4.3. Similar to typings into ordinary type graphs, a typing from an instance graph into a type graph with inheritance incorporates two functions, mapping nodes to node types and edges to edge types respectively.

This pair of functions is not a graph morphism, however, but will be called a *clan morphism.* This clan morphism uniquely characterises the typing morphism into the abstract closure of the type graph with inheritance, i.e. given a clan morphism $ctp : G \to TGI$ there exists a unique typing morphism $\tau_G : G \to \overline{TGI}$ [BEdLT04].

Definition 4.6 (Clan morphism)**.** *Let* $TGI = (T, I, \mathcal{A})$ *be a type graph with inheritance. A* clan morphism $ctp : G \to TGI$ *from a graph* $G = (V_G, E_G)$ *into* TGI *is a pair* $ctp = (ctp_V : V_G \to V_T, ctp_E : E_G \to E_T)$ *such that for all* $e \in E_G$ *the following holds:*

– $ctp_V \circ src_G(e) \in clan_I(src_T \circ ctp_E(e))$ *and*
– $ctp_E \circ tgt_G(e) \in clan_I(tgt_T \circ ctp_E(e))$.

(G, ctp) *is called a* clan-typed graph.

Example 4.7. *Figure 4.2 shows the typing of graph* G_2 *into* TGI*, both from Example 4.2. This typing is done by a clan morphism; the edge typing is not explicitly shown, but follows uniquely from the node typing.*

Although the node type labelled AA in TGI *does not have an outgoing c-edge while the AA-labelled node in* G_2 *does, the clan morphism defines a mapping between these two nodes because the node in* TGI *inherits this edge from the node type labelled A.*

We shall now introduce an order over the set of clan morphisms of a given instance graph: one clan morphism is considered *finer* than another if it assigns more concrete node types to the nodes of the instance graph. This order will be used in the next section to be able to specify commutativity of clan morphisms.

Definition 4.8 (Type refinement). *Let $TGI = (T, I, \mathcal{A})$ be a type graph with inheritance, and let $ctp, ctp' : G \to T$ be clan morphisms. ctp is a* refinement *of ctp', denoted $ctp \leq ctp'$, if*

- *$ctp_V(v) \in clan_I(ctp'_V(v))$ for all $v \in V_G$, and*
- *$ctp_E = ctp'_E$.*

Given two clan-typed graphs (G, ctp_G) and (H, ctp_H) over TGI, a graph morphism $f : G \to H$ is called type-refining *if $ctp_H \circ f \leq ctp_G$ and is called* type-preservering *if $ctp_H \circ f = ctp_G$.*

Thus we have now defined a type graph with inheritance and a typing from an instance graph into a type graph with inheritance. In the following section we shall investigate how to deduce a type graph with inheritance from a graph grammar.

4.1.1 Type Graph with Inheritance for a Graph Grammar

In Chapter 3 we have defined the perfect type graph for a graph grammar GG as the graph $\uplus\mathcal{G} \ / \sim$ such that $(\mathcal{G}, \mathcal{M})$ is the diagram of GG and $\sim$ is the equivalence relation defined in Defintion 3.21. In this approach, two node types were merged if they were mapped onto each other by any morphism or composition of morphisms.

Now we shall choose a more liberal approach by creating an inheritance edge between two nodes if one of these nodes is mapped onto the other by any graph morphism or composition of graph morphisms. These nodes are only merged under certain constraints, which we shall identify in this section.

Example 4.9. *Figure 4.3 gives an example of the difference between an ordinary type graph and a type graph with inheritance. Consider the simple graph grammar given in Figure 4.3a. The only rule, we call this rule r, in this example removes the address attribute of a Student or Teacher.*

A possible type graph for this grammar is given in Figure 4.3b; Figure 4.3c shows a possible type graph with inheritance. The difference between these two different type graphs reveals an important drawback of the ordinary type graphs constructed as in Chapter 3: all nodes sharing some property may be considered type equivalent—because they are mapped by the same left-hand side node in some matching—and are therefore merged. In the case of this example, rule r causes all nodes having an outgoing edge labelled address to be type equivalent.

Analogously to Chapter 3, a type graph with inheritance for a graph grammar should at least satisfy two conditions: it should contain a classification for all rule graphs and al production graphs of the grammar, and all morphisms between these graphs should commute with the clan morphisms.

Definition 4.10 (Type graph with inheritance for a graph grammar). *Let $GG = (G_0, \mathcal{P})$ be a graph grammar and let $(\mathcal{G}, \mathcal{M})$ be the diagram of GG. Then a type graph with inheritance TGI is a* type graph with inheritance for GG *if there exist clan morphisms $ctp_G : G \to TGI$ for all $G \in \mathcal{G}$ such that*

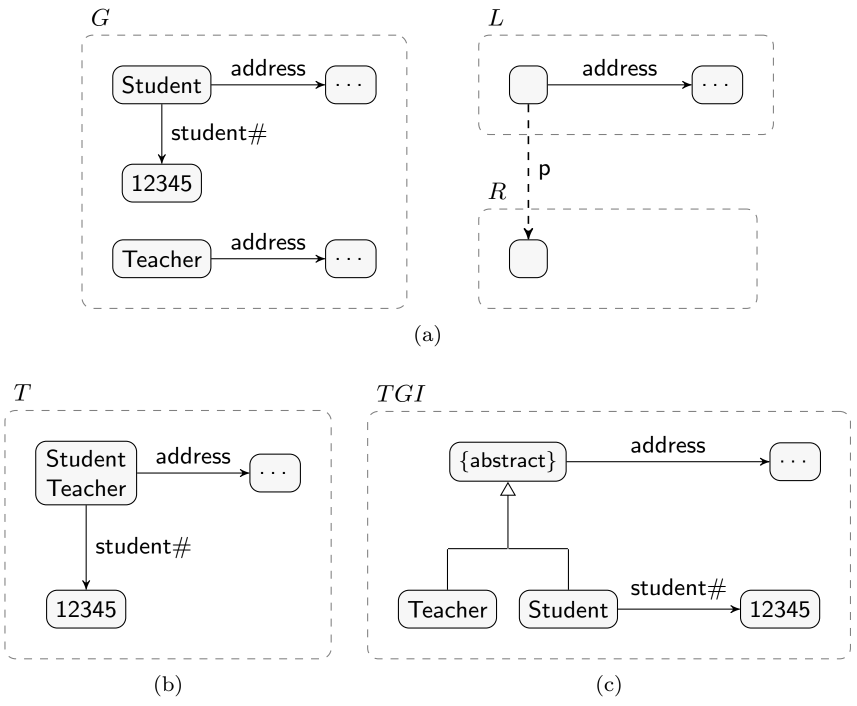


Figure 4.3: A type graph and a type graph with inheritance for a graph grammar

1. *All morphisms $m \in \mathcal{M}$ are type-refining.*
2. *Al rule morpshisms $p \in \mathcal{M}$ are type-preserving.*
3. *All co-matches $m^* \in \mathcal{M}$ for all elements that are not in the image of any rule morphism $p \in \mathcal{M}$ are type-preserving.*

We shall now introduce a particular type graph with inheritance and we shall call this the naive type graph with inheritance for a graph grammar. This type graph will be defined such that it adheres to Definition 4.10.

In order to formalise when morphisms should be either type preserving or type refining, we shall now first define a preorder $\lesssim$ over nodes of the graphs in $\mathcal{G}$, defined such that, for any $m \in \mathcal{M}$ and for any $v \in \operatorname{dom} m$ it holds that m is type refining if $v \lesssim m(v)$ and type preserving if $v \lesssim m(v)$ and $m(v) \lesssim v$.

Definition 4.11. *Given a graph grammar $GG = (G_0, \mathcal{P})$ and the diagram $(\mathcal{G}, \mathcal{M})$ of GG, $\lesssim \subseteq V_{\uplus\mathcal{G}} \times V_{\uplus\mathcal{G}}$ is the smallest reflexive and transitive relation such that*

- *For all morphisms $m \in \mathcal{M}$ it holds that $m(n) \lesssim n$ for all $n \in \operatorname{dom} m$.*
- *For all $L \xrightarrow{p} R \in \mathcal{P}$, it holds that*
  - *$n \lesssim p(n)$ for all $n \in \operatorname{dom} p$*
  - *If there exists a co-match $m^* : R \to H \in \mathcal{M}$, then $n \lesssim m^*(n)$ for all $n \in (\operatorname{dom} m^* \setminus \operatorname{im} p)$.*

Since $\lesssim$ is reflexive and transitive, it is a preorder. Given this preorder, we can now define the naive type graph with inheritance.

Basically, nodes of this naive type graph with inheritance are equivalence classes of $\uplus\mathcal{G}$ over the kernel of the preorder $\lesssim$, which is an equivalence relation since it is reflexive, transitive, and symmetric. Furthermore, the tuples $(n_1, n_2) \in \lesssim$ that are not in $\ker \lesssim$ give rise to the inheritance relation I.

Definition 4.12 (Naive type graph with inheritance)**.** *Let $GG = (G_0, \mathcal{P})$ be a graph grammar and let $(\mathcal{G}, \mathcal{M})$ be the diagram of GG. Then $\widetilde{TGI} = (\widetilde{T}, I, \mathcal{A})$ is a type graph with inheritance for GG such that*

- $\widetilde{T} = (\widetilde{V_T}, \widetilde{E_T}) = \uplus\mathcal{G} \;/ \simeq$
- $I = \{([a]_\simeq, [b]_\simeq) \mid b \lessdot a\}$
- $\mathcal{A} = \{[v]_\simeq \mid \exists L \xrightarrow{p} R \in \mathcal{P} \,.\, v \in \operatorname{dom} p \vee v \in \operatorname{im} p\}$,

where $\lesssim$ is the relation defined in Definition 4.11, $\simeq\, = \ker \lesssim$, and $\lessdot$ is the direct predecessor relation inside $\lesssim$, which is defined as $v \lessdot v'$ if, and only if, $v \lesssim v'$ and $v' \not\lesssim v$ such that there exists no w with $w \notin \{v, v'\}$ and $v \lesssim w \lesssim v'$.

Then the type graph with inheritance $TGI = (T, I, \mathcal{A})$ such that

- $T = (\widetilde{V_T}, E_T)$
- $E_T = \{(s, l, t) \in \widetilde{E_T} \mid \nexists (s', l, t') \in \widetilde{E_T} \,.\, s' \in clan_I(s) \wedge t' \in clan_I(t)\}$

is called the naive type graph with inheritance *for GG.*

According to this definition, type preservation of morphisms is achieved by merging nodes and type refinement by creating inheritance edges between nodes. We shall now prove that the naive type graph with inheritance is a valid type graph for a graph grammar.

Lemma 4.13. *Let GG be a graph grammar and let $(\mathcal{G}, \mathcal{M})$ be the diagram of GG. Then the naive type graph with inheritance for GG is a type graph with inheritance for GG.*

Proof. Definition 4.12 consists of two parts: First $\widetilde{TGI} = (\widetilde{T}, I, \mathcal{A})$ is defined. Thereafter, type graph $T \subseteq \widetilde{T}$ is defined, resulting in a type graph with inheritance $TGI = (T, I, \mathcal{A})$ that is the naive type graph with inheritance for GG.

We shall also divide the proof into two parts: (1) $\widetilde{TGI}$ is a type graph for GG; and (2) TGI is also a type graph for GG.

In order to prove (1), we need to prove two properties, according to Definition 4.10:

- There exists a clan morphism $ctp_G : G \to \widetilde{TGI}$ for all $G \in \mathcal{G}$.
- These clan morphisms commute with all morphisms $m \in \mathcal{M}$.

We define ctp_G as $ctp_G(x) = [x]_\simeq$, for all $x \in V_G \cup E_G$. Well-definedness of ctp_G directly follows from Definition 2.19: ctp_G is uniquely defined for all $v \in V_G$ and for all $e \in E_G$. Furthermore, in order for ctp_G to be a clan morphism, for all $e \in E_G$ the following should hold:

$$ctp_G(src(e))\ I^*\ src(ctp_G(e))$$

and hence

$$[src(e)]_{\simeq}\ I^*\ src([e]_{\simeq}),$$

where I^* is the reflexive-transitive closure of I. From Definition 2.19 it follows that

$$[src(e)]_{\simeq} = src([e]_{\simeq}).$$

and therefore ctp_G is a clan morphism.

Next we need to prove commutativity of the clan morphisms, i.e.

$$\forall m \in \mathcal{M} \,.\, m : G \to H \Longrightarrow ctp_H \circ m \leq ctp_G.$$

Rewriting gives, for all $m \in \mathcal{M}$ and for all $x \in V_G \cup E_G$,

$$\begin{aligned}(ctp_H \circ m)(x)\ I^*\ ctp_G(x)\\ ctp_H(m(x))\ I^*\ [x]_{\simeq}\\ [m(x)]_{\simeq}\ I^*\ [x]_{\simeq}\end{aligned}$$

This directly follows from Definition 4.11 and hence $\widetilde{TGI}$ is a type graph with inheritance for GG.

To prove (2), we define a clan morphism $ctp'_G : G \to TGI$ as

- $ctp'_{G,V}(v) = ctp_G(v)$
- $ctp'_{G,E}(e) = ctp_G(e')$

where $e' \in E_T$ such that $lbl(e) = lbl(e')$, $src(e') \in clan_I(src(e))$, and $tgt(e') \in clan_I(tgt(e))$. From Definition 4.12 it follows that such e' exists; from Definition 4.6 it follows that ctp'_G is a clan morphism and hence we have proved that such clan morphism exists, as required. □

This naive type graph with inheritance contains inheritance relations between all node types n_1, n_2 for which $n_1 \lesssim n_2$ holds, including all cases where type equivalence of these node types would be more appropriate. As an example, consider Figure 4.4, which illustrates the type naive type graph with inheritance for the graph grammar given in Figure 4.3a. In comparison with the type graph with inheritance we expected for this graph grammar, which is given in Figure 4.3c, this naive type graph with inheritance contains too many inheritance edges and too few nodes are merged.

Therefore we shall define scenarios in section 4.3, that describe when certain nodes in the type graph with inheritance need to be merged, in order to contract the inheritance relation of a type graph with inheritance.

But first we shall define an algorithm that computes a type graph with inheritance from a given graph grammar; this type graph possibly contains, like the naive type graph with inheritance, many inheritance edges that can be omitted.

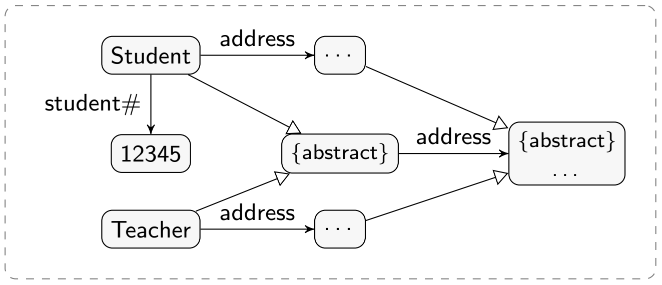


Figure 4.4: A naive type graph with inheritance for the graph grammar given in Figure 4.3a.

4.2 Inference of Type Graphs with Inheritance

In this section we shall outline an algorithm for computing a type graph with inheritance for a given graph grammar. This algorithm is given in Algorithm 4.1. In contrast to the algorithms presented in Chapter 3, this algorithm statically initialises the type graph with inheritance as a quotient of the disjoint union of the start graph and all rule graphs of the grammar, over some equivalence relation. Thereafter, it does not merge any more elements but only introduces new inheritance edges.

At lines 1 until 4 the initial type graph with inheritance is initialised as just described; line 5 initialises the inheritance relation to the empty set and line 6 marks all nodes in the domain and image of any rule morphism as abstract.

Thereafter the algorithm enters the outer loop, ranging from line 7 to line 17, in which the inheritance relation is constructed. This is done similarly to the construction of the equivalence relation *Equiv* in the algorithms described earlier: if a left-hand side node matches into any node in the type graph with inheritance, an inheritance edge from this matched node to the left-hand side node will be added to the type graph with inheritance.

The algorithm obviously terminates: the main loop of the algorithm introduces inheritance edges between a finite amount of node types. At most inheritance edges are created between any two node types, but this is still a finite number. Also confluence follows directly from the structure of the algorithm: each iteration of the loop adds extra elements to the inheritance relation, regardless of the current contents of that relation. Therefore the inheritance relation will have the same contents after each iteration of the loop, regardless the order in which production rules are processed.

We shall now prove that the algorithm computes a valid type graph with inheritance for a graph grammar.

Lemma 4.14. *Given a graph G and a production rule $r : L \xrightarrow{p} R$, let H be the result of the graph transformation $G \overset{r,m}{\Longrightarrow} H$ for matching $m : L \to G$, and let $p^* : G \to H$ and $m^* : R \to H$ be the morphisms in the pushout, as specified in Definition 2.26.*

Furthermore, let TGI be a type graph with inheritance and let $ctp_G : G \to TGI$

Algorithm 4.1: Algorithm for computing a type graph with inheritance for a graph grammar

```
**Input**: $GG = (G_0, \mathcal{P})$
**Output**: A type graph with inheritance for $GG$
1  $T \leftarrow G_0 \uplus \{L_r \mid r \in \mathcal{P}\} \uplus \{R_r \mid r \in \mathcal{P}\}$;
2  $Equiv \leftarrow \{(v, v') \mid \exists L \xrightarrow{p} R \in \mathcal{P} \,.\, p(v) = v'\}$;
3  $T \leftarrow T \mathbin{/} Equiv$;
4  $I, I' \leftarrow \varnothing$;
5  $\mathcal{A} \leftarrow \{[v]_{Equiv} \mid \exists L \xrightarrow{p} R \in \mathcal{P} \,.\, v \in \operatorname{dom} p \vee v \in \operatorname{im} p\}$;
6  repeat
7      $I' \leftarrow I$;
8      forall $L \xrightarrow{p} R \in \mathcal{P}$ do
9          $\mathcal{M} \leftarrow \{ctp_m \mid ctp_m : L \to T\}$;
10         forall $v, v' \in V_T$ do
11             if $\exists m \in \mathcal{M} \,.\, m(v) = v'$ then
12                 $I \leftarrow I \cup \{(v', [v]_{Equiv})\}$ ;
13             end
14         end
15     end
16 until $|I| = |I'|$ ;
17 return $(T, I, \mathcal{A})$;
```

and $ctp_R : R \to TGI$ be two clan morphisms such that $ctp_G \circ m \leq ctp_R \circ p$. Then, there also exists a clan morphism $ctp_H : H \to TGI$ such that $ctp_G|_{\operatorname{dom} p^*} = ctp_H \circ p^*$ and $ctp_R|_{\operatorname{dom} m^*} \geq ctp_H \circ m^*$.

Proof. We define ctp_H as follows:

– $ctp_H(v) = \begin{cases} ctp_G(v') & if\, v = p^*(v') \\ ctp_R(v') & if\, v = m^*(v'), v' \notin \operatorname{im} p^* \end{cases}$ for nodes of H, and

– $ctp_H(e) = \begin{cases} ctp_G(e') & if\, e = p^*(e') \\ ctp_R(e') & if\, e = m^*(e'), e' \notin \operatorname{im} p^* \end{cases}$ for edges of H.

First we need to prove that ctp_H is well-defined, i.e. functions $ctp_{H,V}$ and $ctp_{H,E}$ must be well-defined. This holds if $ctp_G(x') \leq ctp_R(x'')$, for all $x \in V_H \cup E_H$ such that $x = p^*(x') = m^*(x'')$.

This holds because x' and x'' have the same preimage in L, meaning that there exists an element $l \in L$ such that $x' = m(l)$ and $x'' = p(l)$. Then

$$ctp_G(x') = ctp_G(m(e)) \leq ctp_R(p(e)) = ctp_R(x'').$$

Second, we need to prove that ctp_H is a clan morphism. As specified in Definition 4.6 this holds if, for all $e \in E_H$, the following holds:

– $ctp_H(src(e))\ I^*\ src(ctp_H(e)))$ and
– $ctp_H(tgt(e))\ I^*\ tgt(ctp_H(e)))$

where I^* is the reflexive-transitive closure of I. For this, the following distinctive cases can be identified. We shall only prove these for $src(e)$; the proofs for $tgt(e)$ are symmetric:

1. If $e \in \operatorname{im} p^*$, then $src(e) \in \operatorname{im} p^*$. So
$$\begin{aligned} &ctp_H(src(e)) = \\ &ctp_G(p^*(src(e))) = \\ &ctp_G(src(p^*(e)))\ I^* \\ &src(ctp_G(p^*(e))) = src(ctp_H(e)) \end{aligned}$$
and hence the required property $ctp_H(src(e))\ I^*\ src(ctp_H(e)))$ holds in this case.
2. If $e \notin \operatorname{im} p^*$ and $src(e) \in \operatorname{im} p^*$, there exists a $v \in \operatorname{dom} m^*$ and a $v' \in \operatorname{dom} p^*$ such that $m^*(v) = p^*(v') = src(e)$. In this case, v and v' have the same preimage in L, thus there exists $v'' \in L$ such that $v' = m(v'')$ and $v = p(v))$. Then
$$\begin{aligned} &ctp_H(src(e))\ I^* \\ &ctp_R(v) = \\ &ctp_R(src(e'))\ I^* \\ &src(ctp_R(e')) = src(ctp_H(e)) \end{aligned}$$
and hence the required property $ctp_H(src(e))\ I^*\ src(ctp_H(e)))$ holds in this case.
3. If $e, src(e) \notin \operatorname{im} p^*$, then $e, src(e) \in \operatorname{im} m^*$. So
$$\begin{aligned} &ctp_H(src(e))\ I^* \\ &ctp_R(m^*(src(e))) = \\ &ctp_R(src(m^*(e)))\ I^* \\ &src(ctp_R(m^*(e))) = src(ctp_H(e)) \end{aligned}$$
and hence the required property $ctp_H(src(e))\ I^*\ src(ctp_H(e))$ holds in this case.

Since the proof for tgt_G is analogous to the proof for src_G, this proves that there exists a clan morphism $ctp_H : H \to TGI$ for any graph H resulting from graph transformation $G \overset{r,m}{\Longrightarrow} H$.

Next, we prove $ctp_G|_{\operatorname{dom} p^*} = ctp_H \circ p^*$. Let $e \in E_H$. Because ctp_G is restricted to $\operatorname{dom} p^*$, we know that $e \in \operatorname{im} p^*$. Hence, the required property follows directly from the definition of ctp_H:
$$ctp_H(p^*(e')) = ctp_G(e')$$
for some $e' \in \operatorname{dom} p^*$ such that $p^*(e') = e$.

Finally, we prove $ctp_R|_{\operatorname{dom} m^*} \geq ctp_H \circ m^*$. Let $e \in E_H$. If $e \notin \operatorname{im} p^*$, the required property follows from the definition of ctp_H:
$$ctp_H(m^*(e'')) = ctp_R(e'')$$

for some $e'' \in \operatorname{dom} p^*$ such that $m^*(e'') = e$. If $e \in \operatorname{im} p^*$, then there exists $e' \in \operatorname{dom} p^*$ such that $p^*(e') = e$ and $m^{-1}(e') = p^{-1}(e'')$. From $ctp_G \circ m \leq ctp_R \circ p$ it then follows that

$$ctp_H(m^*(e'')) \geq ctp_R(e'') \qquad \square$$

Using this proof, we now prove that Algorithm 4.1 is correct, i.e. it computes a valid type graph with inheritance for a given graph grammar.

Theorem 4.15. *Given a graph grammar GG, the type graph with inheritance induced from GG by Algorithm 4.1 is a valid type graph with inheritance for GG.*

Proof. We prove that the type graph with inheritance TGI at line 24 of the algorithm is a type graph for $(\mathcal{G}, \mathcal{M})$.

Let $D_0, \ldots, D_i, \ldots$ be an infinite sequence of diagrams, inductively defined as follows:

– $D_0 = (\mathcal{G}_0, \mathcal{M}_0)$
  · $\mathcal{G}_0 = \{G_0\} \cup \{L_r \mid r \in \mathcal{P}\} \cup \{R_r \mid r \in \mathcal{P}\}$
  · $\mathcal{M}_0 = \{p \mid L \xrightarrow{p} R \in \mathcal{P}\}$

– $D_i = (\mathcal{G}_i, \mathcal{M}_i)$
  · $\mathcal{G}_i = \mathcal{G}_{i-1} \cup \{H \mid \exists r \in \mathcal{P} \,.\, \exists G \in \mathcal{G}_{i-1} \,.\, G \overset{r,m}{\Longrightarrow} H\}$
  · $\mathcal{M}_i = \mathcal{M}_{i-1} \cup$

$$\left\{ m, m^*, p^* \mid \exists r = L \xrightarrow{p} R \in \mathcal{P}, \exists G \in \mathcal{G}_{i-1} \,.\, \begin{tikzcd} L \arrow[r, "p"] \arrow[d, "m"] & R \arrow[d, "m^*"] \\ G \arrow[r, "p^*"] & H \end{tikzcd} \right\}$$

Let $TGI_0, \ldots, TGI_i, \ldots$ be an infinite sequence of type graphs with inheritance, where TGI_0 is the intermediate type graph with inheritance at line 10 of Algorithm 4.1 before the first iteration of the loop ranging from line 10 until 23, and TGI_i $(i > 0)$ is the intermediate type graph at line 23 of Algorithm 4.1 after the i^{th} iteration of this loop.

We shall now inductively prove that TGI_i is a type graph with inheritance for D_i, for all $i \geq 0$.

Basis step:

We prove that TGI_0 is a type graph with inheritance for D_0, i.e.

(a) $\forall G \in \mathcal{G}_0 \,.\, \exists ctp_G : G \to TGI_0$
(b) $\forall m : G \to H \in \mathcal{M}_0 \,.\, ctp_H \circ m \leq ctp_G$

To prove (a), we prove that all $G \in \{G_0\} \uplus \{L_r \mid r \in \mathcal{P}\} \uplus \{R_r \mid r \in \mathcal{P}\}$ have a clan morphism into TGI_0. Since the algorithm initialises the type graph to the disjoint union of exact this set of graphs, and the only changes to TGI are performed by the loop ranging from line 3 to 7 of the algorithm using the quotient construction, it follows from Lemma 2.22 that (a) holds.

To prove (b), we prove that $ctp_R \circ p \leq ctp_L$ for all $p = \{p \mid L \xrightarrow{p} R \in \mathcal{P}\}$. This follows directly from lines 3–8 of the algorithm: for all these morphisms it merges all elements in $\operatorname{dom} p$ with their images under p.

Induction hypothesis:

TGI_i is a type graph with inheritance for D_i, i.e.

(a) $\forall G \in \mathcal{G}_i, . \exists ctp_G : G \to TGI_i$

(b) $\forall m : G \to H \in \mathcal{M}_i \,.\, ctp_H \circ m \leq ctp_G$

Induction step:

We prove that TGI_{i+1} is a type graph with inheritance for D_{i+1}, i.e.

(a) $\forall G \in \mathcal{G}_{i+1} \,.\, \exists ctp'_G : G \to TGI_{i+1}$

(b) $\forall m : G \to H \in \mathcal{M}_{i+1} \,.\, ctp'_H \circ m \leq ctp'_G$

Let $\alpha : TGI_i \to TGI_{i+1}$ be a total graph morphism. Existence of such α follows directly from Lemma 2.21.

First we prove (a). Let $G \in \mathcal{G}_{i+1}$ be a graph. If $G \in \mathcal{G}_i$, there exists $ctp'_G : G \to TGI_{i+1}$, defined as $ctp'_G = \alpha \circ ctp_G$.

If $G \notin \mathcal{G}_i$, there exists $G' \in \mathcal{G}_i$ such that $G' \overset{r,m}{\Longrightarrow} G$, for some $r = (L \xrightarrow{p} R) \in \mathcal{P}$ and some $m : L \to G'$. According to Lemma 4.14, ctp'_G exists in this case if there exist $ctp'_{G'} : G' \to TGI_{i+1}$ and $ctp'_R : R \to TGI_{i+1}$ such that $ctp'_{G'} \circ m \leq ctp'_R$.

Since $G', R \in \mathcal{G}_i$, $ctp'_{G'}$ and ctp'_R exist; these morphisms are defined as $ctp'_{G'} = \alpha \circ ctp_{G'}$ and $ctp'_R = \alpha \circ ctp_R$. Since $ctp_{G'} \circ m \leq ctp_R$ holds—this follows from the induction hypothesis—we have $\alpha \circ ctp_{G'} \circ m \leq \alpha \circ ctp_R$ and hence the required property $ctp'_{G'} \circ m \leq ctp'_R$. (a) then follows directly from Lemma 4.14.

Next we prove (b). Let $m : G \to H \in \mathcal{M}_{i+1}$ be a graph morphism. If $m \in \mathcal{M}_i$, then $ctp_H \circ m \leq ctp_G$ follows from the induction hypothesis.

If $m \notin \mathcal{M}_i$, then m is a match, co-match, or co-production that gave rise to the construction of some $G \in \mathcal{G}_i \setminus \mathcal{G}_{i-1}$. For the match we have proved the required property in (a). For the co-match and co-production this property follows directly from Lemma 4.14.

This proves that TGI_i is a type graph with inheritance for D_i, for all $i > 0$. The algorithm terminates when $TGI_{i+1} = TGI_i$. Because of this, TGI_i is a type graph with inheritance for D_{i+1} and hence for all D_i for $i \to \infty$. Since D_i converges to $(\mathcal{G}, \mathcal{M})$ if $i \to \infty$, TGI_i is a type graph for $(\mathcal{G}, \mathcal{M})$, as required. □

Thus we have now specified an algorithm and have proved that it computes a valid type graph with inheritance for a given graph grammar. In the next section we will define contraction scenarios for reducing the size of type graphs with inheritance. Because of the similarity between the algorithm that we defined and the naive type graph with inheritance we defined earlier in this chapter, these contraction scenarios are also useable for the type graph computed by the algorithm.

4.3 Contraction

In the last section we introduced the concept of type graphs with inheritance and typings from instance graphs into such type graphs; using this information we defined a naive type graph with inheritance for a graph grammar, which contains inheritance edges for all tuples of nodes that are in preorder $\lesssim$. In many cases, however, merging these nodes instead of creating an inheritance edge between them is preferred in order to reduce redundancy.

In this section we will discuss criteria for when to merge nodes instead of creating inheritance edges between these nodes. To formalise the notion of merging nodes in a type graph with inheritance, we shall start with defining the quotient construction on type graphs with inheritance.

Definition 4.16 (Quotient of a type graph with inheritance)**.** *Given a type graph with inheritance $TGI = (T, I, \mathcal{A})$ and an equivalence relation $\simeq \subseteq V_T \times V_T$, the type graph with inheritance $TGI \,/ \simeq\, = (T', I', \mathcal{A}')$ such that*

- $T' = T \,/\!\simeq$
- $I' = \{([a]_\simeq, [b]_\simeq) \mid (a, b) \in I, (a, b) \notin \simeq\}$
- $\mathcal{A}' = \{[n]_\simeq \mid [n]_\simeq \subseteq \mathcal{A}\}$

is called the quotient *of TGI over $\simeq$.*

Note that if two nodes are merged using this construction, the resulting node is concrete if, and only if, at least one of the two nodes is concrete.

This describes a general quotient construction on type graphs with inheritance. We shall, however, use this construction under two additional constraints; two nodes can only be merged if they are connected through a inheritance edge and if these nodes connected to each other by a chain of inheritance edges, all intermediate nodes in this chain are also merged with these nodes. We shall call this construction a *contraction* of a type graph with inheritance, and define it formally as follows:

Definition 4.17 (Contraction of a type graph with inheritance)**.** *Given a type graph with inheritance $TGI = (T, I, \mathcal{A})$, let $I^{\underline{*}}$ be the reflexive-transitive-symmetric closure of I and let $\simeq \subseteq I^{\underline{*}}$ be an equivalence relation such that*

$$\forall x, y, z \in V_T \,.\, (x \simeq y \wedge x \; I \; z \wedge z \; I^{\underline{*}} \, y) \Longrightarrow x \simeq z. \tag{4.1}$$

Then, $TGI \,/ \simeq$ is a contraction *of TGI.*

One important property that should be satisfied is that the contraction of a type graph with inheritance must give a classification to at least the set of instances of this type graph with inheritance. Reason for this is that if this property is not satisfied, the type graph produced by contraction is not a valid type graph with inheritance.

Lemma 4.18. *Let $TGI = (T, I, \mathcal{A})$ be a type graph with inheritance and let $TGI' = TGI \,/\!\simeq$ for some equivalence relation $\simeq$. Then, if TGI' is a contraction of TGI, we have $\mathcal{I}_{TGI} \subseteq \mathcal{I}_{TGI'}$.*

Proof. Given a clan morphism $ctp_G : G \to TGI$ for an arbitrary graph $G \in \mathcal{I}_{TGI}$, we shall prove that there exists a clan morphism $ctp'_G : G \to TGI'$.

We define ctp'_G as $ctp'_G(x) = \big[ctp_G(x)\big]_{\simeq}$. ctp'_G is a clan morphism if, for all $e \in E_G$,

$$\big[ctp_G(src(e))\big]_{\simeq} \; I'^* \; src(\big[ctp_G(e)\big]_{\simeq})$$

holds under the assumption that

$$ctp_G(src(e)) \; I^* \; src(ctp_G(e)),$$

which follows from Definition 4.6. Here, I^* is the reflexive-transitive closure of I and I'^* is the reflexive-transitive closure of I', the inheritance relation of TGI'.

Let $v, v' \in V_T$ be nodes of T such that $v \; I^* \; v'$ and $v \neq v'$. From this it follows that there exist $w_1, \dots, w_n \in V_T$ such that

$$v \; I \; w_1 \; I \; \dots \; I \; w_n \; I \; v',$$

and according to Definition 4.16, we have

$$[v]_{\simeq} \; I'^* \; [w_1]_{\simeq} \; I'^* \; \dots \; I'^* \; [w_n]_{\simeq} \; I'^* \; [v']_{\simeq}$$

and hence $[v]_{\simeq} \; I'^* \; [v']_{\simeq}$. If we now substitute v by $ctp_G(src(e))$ and v' by $src(ctp_G(e))$, we get

$$[ctp_G(src(e))]_{\simeq} \; I'^* \; [src(ctp_G(e))]_{\simeq}.$$

From the definition of graph quotienting it then follows that

$$[src(ctp_G(e))]_{\simeq} = src([ctp_G(e)]_{\simeq})$$

and hence

$$[ctp_G(src(e))]_{\simeq} \; I'^* \; src([ctp_G(e)]_{\simeq}).$$

This proves the required property for $v \neq v'$; if $v = v'$, this property trivially follows from

$$[ctp_G(src(e))]_{\simeq} = [src(ctp_G(e))]_{\simeq}. \qquad \Box$$

4.3.1 Contraction Scenarios

Although the decision whether to use a subtype relation between types or to merge these nodes in an object-oriented setting often relies upon personal preferences of programmers, we shall introduce some scenarios in which we think contraction is appropriate in general. Because the scenarios provide criteria for merging nodes between which there exist inheritance relations, applying contraction on these scenarios to the naive type graph with inheritance yields a more compact type graph with inheritance, i.e. with less redundant inheritance edges.

Each scenario described in any of the next sections will define an quivalence relation that can be used for contraction, as specified in Definition 4.17. These equivalence relations will be called R_1, R_2, and R_3 for the three contraction scenarios, respectively; the first scenario identifies a contraction based on a bisimulation over node types, the second over supertypes that simulate their subtypes, and the third specifies a contraction for abstract supertypes that have exactly one subtype.

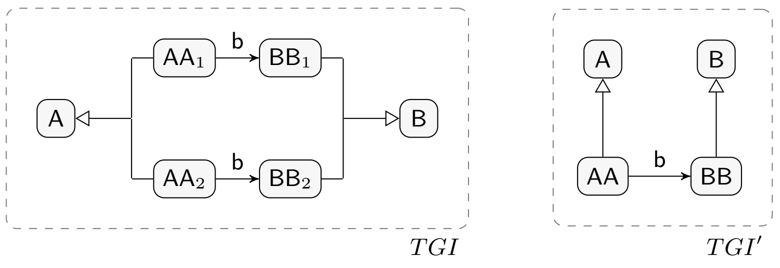


Figure 4.5: An example of a supertype with multiple equivalent subtypes

Bisimulation over node types

This first scenario is based on a form of equivalence that is called *bisimulation*. This type of equivalence is often used in state transition systems to express that states in such systems have the same behaviour. In terms of type graphs with inheritance, we describe the behaviour of node types based on their outgoing edges and their supertypes.

Example 4.19. *Figure 4.5 illustrates an example of contraction based on this scenario. It shows two type graphs with inheritance TGI and TGI'. Because in TGI there are two nodes labelled AA and two nodes labelled BB, we have indexed them with numbers in subscript; this index is not part of the label.*

Nodes AA_1 and AA_2 have the same behaviour—they have the same outgoing edge and the same supertype—and so do BB_1 and BB_2; therefore these nodes are merged in TGI', which is the contraction of TGI based on this scenario.

To formalise the way the nodes in the preceding example are equivalent, we shall now introduce the notions of *simulation* and *bisimulation*, based upon the definitions by Milner [Mil99]. Note that these are not the classical definitions of simulation and bisimulation, but a slightly altered version to cope with inheritance edges.

Definition 4.20 (Simulation over a type graph with inheritance)**.** *Let $TGI = (T, I, \mathcal{A})$ be a type graph with inheritance, and let $\mathcal{S} \subseteq I^{\underline{*}}$ be a binary relation where $I^{\underline{*}}$ is the reflexive-transitive-symmetric closure of I. Then $\mathcal{S}$ is called a* simulation *over TGI if, whenever $n_1 \mathrel{\mathcal{S}} n_2$, the following properties hold:*

1. *if $n_1 \xrightarrow{a} n_1'$ then there exists $n_2' \in V_T$ such that $n_2 \xrightarrow{a} n_2'$ and $n_1' \mathrel{\mathcal{S}} n_2'$.*
2. *if $n_1' \in clan_I(n_1)$ then there exists $n_2' \in clan_I(n_2)$ such that $n_1' \mathrel{\mathcal{S}} n_2'$.*

We use $s \xrightarrow{l} t$ to denote that there exists $(s, l, t) \in E_T$. If there exists a simulation $\mathcal{S}$ such that $p \mathrel{\mathcal{S}} q$, we say that q simulates *p.*

If the converse of S also is a simulation, we call S a bisimulation. This is defined as follows.

Definition 4.21 (Bisimulation over a type graph with inheritance)**.** *A simulation $\mathcal{S}$ over a type graph with inheritance TGI is a bisimulation over TGI if also $\mathcal{S}^{-1}$ is a simulation over TGI. Two nodes $n_1, n_2 \in V_T$ are* bisimilar*, denoted $n_1 \sim n_2$ if there exists a bisimulation $\mathcal{S}$ such that $n_1 \mathrel{\mathcal{S}} n_2$.*

The bisimilarity relation $\sim$ is also a bisimulation relation [Mil99]; moreover, from the definition of $\sim$ it follows that $\sim$ is the largest bisimulation relation over TGI. A proof that $\sim$ is also an equivalence relation is given by Milner [Mil99].

This notion of simulation does not necessarily satisfy (4.1) of Definition 4.17; therefore we define equivalence relation R_1 with an extra requirement to fullfil this property.

Definition 4.22. *Let R_1 be the smallest equivalence relation such that $n_1\ R_1\ n_2$ if $n_1 \sim n_2$, with the additional requirement that*

$$\forall x, y, z \in V_T\,.(x\ R_1\ y \wedge x\ I\ z \wedge z\ I^{\underline{*}}\ y) \Longrightarrow x\ R_1\ z.$$

This defines the equivalence relation used for contraction based on this scenario. Now we prove that R_1 can indeed be used for contraction.

Lemma 4.23. *Given a type graph with inheritance $TGI = (T, I, \mathcal{A})$, $TGI\ /\ R_1$ is a contraction of TGI.*

Proof. R_1 must be a subset of $I^{\underline{*}}$ and must satisfy Property (4.1) of Definition 4.17. The first requirement follows directly from Definition 4.20; the second from Definition 4.22. □

Having proved this, we may conclude that R_1 is a valid equivalence relation for contraction.

> *The equivalence relation used for contraction based on the scenario of bisimulation over node types is R_1.*

Simulation relation between subtypes and supertypes

This second scenario focuses on a similar equivalence of node types. This is shown in Figure 4.6; nodes in this figure are indexed with their node identities. Graph TGI contains five node types of which the nodes A and AA and B and BB behave identically—they have the same outgoing edges—and therefore the inheritance relation between these nodes is unnecessary.

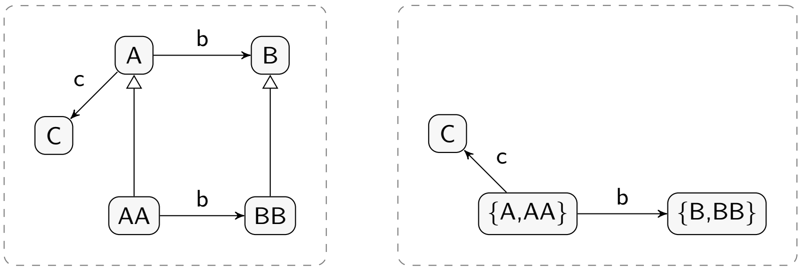


Figure 4.6: A supertype simulating its subtype

This scenario is based on a more indirect form of simulation between node types. We use the information that a node inherits the outgoing edges of all nodes in its inheritance clan; there implicitly always exists a simulation relation from a subtype to all of its supertypes because of this property. Therefore, subtype and a supertype can be considered equivalent if the supertype simulates the subtype according to the following definition.

Definition 4.24. *Given a type graph with inheritance $TGI = (T, I, \mathcal{A})$, R_2' is the relation defined as*

$$n_1\ R_2'\ n_2,\ \text{if}\ n_1\ I\ n_2\ \text{and there exists a simulation}\ \mathcal{S}\ \text{such that}\ n_1\ \mathcal{S}\ n_2.$$

R_2 is defined as the symmetric closure of R_2'.

Since $\mathcal{S}$ is reflexive and transitive, R_1 is reflexive, transitive, and symmetric and hence is an equivalence relation. Now we prove that R_1 can also be used for contraction.

Lemma 4.25. *Given a type graph with inheritance $TGI = (T, I, \mathcal{A})$, $TGI\ /\ R_2$ is a contraction of TGI.*

Proof. R_2 must be a subset of $I^{\underline{*}}$ and must satisfy Property (4.1) of Definition 4.17. The first requirement follows immediately from Definition 4.20; the second follows from Definition 4.24, which states that $n_1\ R_2\ n_2$ can only hold if $n_1\ I\ n_2$. This satisfies the required property. □

Having proved this, we may consider R_2 as the equivalence relation used for contraction based on this scenario.

> *The equivalence relation used for contraction based on the scenario of a simulation relation between subtypes and supertypes is R_2.*

Abstract supertype with one concrete subtype

The first two scenarios were useful for identifying general situations in which subtypes and supertypes could be merged. This last scenario aims at a more specific criterion: an abstract type that has only one concrete subtype. As expressed in Section 4.1, abstract node types cannot be instantiated and are therefore unnecessary if they have only one subtype.

Figure 4.7 gives an example of this scenario. Type graph TGI contains one abstract node type, labelled A, that has exactly one subtype, labelled AA. Type A cannot be instantiated and contains only information for type AA; therefore these two node types should be merged.

We shall now define the equivalence relation, yielded by this scenario, that can be used for contraction.

Definition 4.26. *Let $TGI = (T, I, \mathcal{A})$. We define R_3 as the smallest equivalence relation such that $n_1\ R\ n_2$ for all $n_1 \in \mathcal{A}, n_2 \in V_T$, if $n_2\ I\ n_1$ such that there exists no n_3 with $n_3 \neq n_2$ and $n_3\ I\ n_1$.*

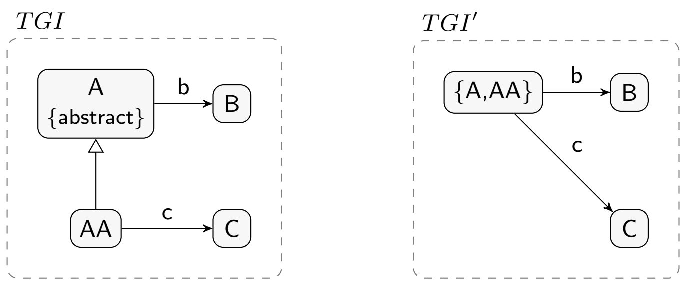


Figure 4.7: An abstract supertype with one concrete subtype

R_3 is an equivalence relation by definition. Now we prove that this equivalence relation can be used for contraction.

Lemma 4.27. *Let $TGI = (T, I, \mathcal{A})$ be a type graph with inheritance. Then TGI / R_3 is a contraction of TGI.*

Proof. R_3 must be a subset of $I^{\underline{*}}$ and must satisfy to Property (4.1) of Definition 4.17. Both requirements follow directly from Definition 4.26. □

Having proved this, we may consider R_3 as the equivalence relation used for contraction based on this scenario.

> *The equivalence relation used for contraction based on the scenario of one abstract sypertype with one concrete subtype is R_3.*

4.4 Discussion

In this chapter we have extended the notion of type graphs with inheritance over node types. After introducing the concept of type inheritance with respect to graph grammars, we defined a naive type graph with inheritance and introduced an algorithm that computes a type graph with inheritance in a similar way.

Both the naive type graph with inheritance and the type graph computed by the algorithm contain an over-approximation of inheritance edges and therefore are generally larger graphs than the type graphs we defined in Chapter 3; therefore we introduced contraction scenarios to remove some obvious redundant inheritance edges.

Although application of these scenarios to the naive type graph with inheritance yields a type graph in which some redundant nodes, the list of contraction scenarios is not exhaustive; future research should point out whether more scenarios can be found.

The main problem with inheritance is that it is highly user-specific. Although this means that it will be impossible to automatically construct a "perfect" type graph with inheritance in general, using the theory and algorithm defined in this chapter a good approximation of such type graph can be made.

CHAPTER 5

Implementation

In the preceding chapters we have investigated the theory of typed graph grammars and have specified how type graphs can be constructed from a given graph grammar. In this chapter we will describe how this theory can be used in practice and we shall describe an implementation of the improved type inference algorithm, which we defined in Chapter 3 on page 40, in GROOVE.

We will first give a brief overview of GROOVE, particularly focused on the extra functionality that has been added by the implementation of the type inference algorithm. Thereafter, in Section 5.2, we will discuss the implementation itself; but instead of getting into too much detail we will rather describe the implementation process.

Next, in Section 5.3, we evaluate the implemented algorithm using two sample test cases. The first test case is the implementation of a problem called *"gossiping girls"* in GROOVE. The limited size of this example allows a understandable outline of the execution of the algorithm for this problem. In the second example, which is an implementation of a list append function, we find out that the algorithm computes a type graph that distinguishes more types than we expected beforehand. This leads to new insights, which are subsequently evaluated in Section 5.4, as well as other suggestions to take advantages of the implemented algorithm in GROOVE.

5.1 An overview of Groove

GROOVE is a tool set that uses graphs and graph transformations to facilitate the modelling and verification of object-oriented systems and model transformations. In its original form the tool was based on simple edge labelled graphs, which we defined in Definition 2.14; currently extensions towards attributed graphs and graph abstractions are being introduced. Groove uses the single-pushout approach with negative application conditions as a basis for the graph transformations, although the tool is designed modularly such that other approaches can easily be implemented [Ren04a].

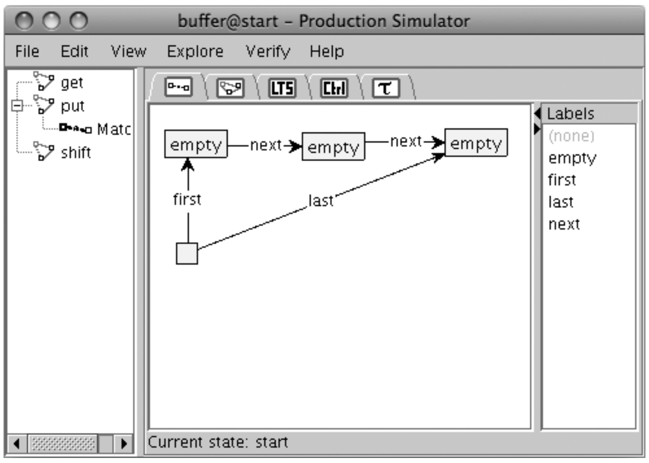


Figure 5.1: Screenshot of the GROOVE simulator

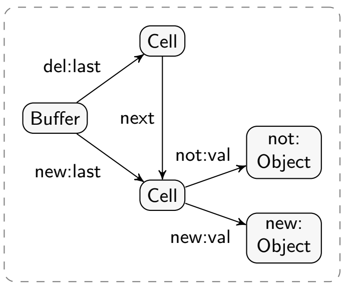
Although GROOVE consists of several different tools, during this project we are particularly interested in the simulator that acts as a graph based model checker; a screenshot of the graphical user interface of the simulator is given in Figure 5.1. The user interface consists of three panels: the leftmost panel displays an index of all production rules of a grammar and their matches into the current graph, which is displayed in the middle panel; the rightmost panel displays a list of all labels used in the current graph.

Instead of the current graph, the middle panel can also display the currently selected production rule, the graph transition system constructed by the graph grammar so far, control automata, and the type graph for the grammar. Later in this section we will elaborate on the functionality to display type graphs; the other options are beyond the scope of this thesis; information on these options can be found on the GROOVE website [RBKS]. First, we will briefly discuss the notation of production rules in GROOVE, which deviates from the notation we have used for production rules thus far.

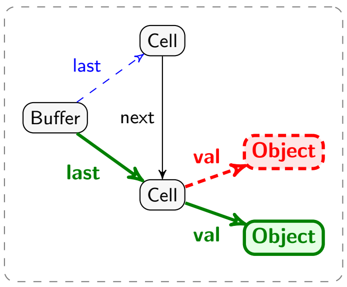
5.1.1 Production Rules

As opposed to the theory discussed in the preceding chapters, GROOVE uses a single-graph representation for production rules. Instead of separate graphs for left-hand sides, right hand sides and possible application conditions, all information of a production rule is represented in one single graph $L \cup R \cup \{N \mid N \in \mathcal{N}\}$. Despite this distinction, production rules expressed in this single-graph representation can contain all information that can also be expressed using separate left-hand side and right-hand side graphs.

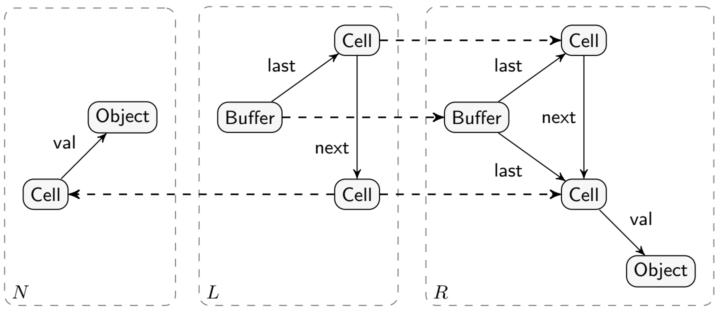
In order to distinguish between elements (i.e., nodes and edges) originating from L, R, or any of the production rules in this single-graph representation, GROOVE separates these elements into four different kinds, being so-called *reader*, *eraser*, *creator*, and *embargo* elements. This distinction is visualised in GROOVE using different colours and shapes and can be achieved using different labels.



(a) Edit format in GROOVE



(b) Display format in GROOVE



(c) Traditional format, with left-hand side and right-hand side graphs and a negative application condition

Figure 5.2: Three versions of production rule put of a linked list.

Reader elements are the elements that remain unchanged during graph transformation; they represent $\operatorname{dom} p \cup \operatorname{im} p$ of traditional production rules. Eraser elements are the elements in $L \setminus \operatorname{dom} p$ and represent the elements that are deleted upon transformation. Creator elements are the elements that are introduced upon transformation; they represent the elements $R \setminus \operatorname{im} p$ of traditional production rules. Finally, embargo elements represent negative application conditions, i.e. the elements $\{N \mid N \in \mathcal{N}\} \setminus \{\operatorname{im} n \mid n : L \to N\}$ of traditional production rules.

Example 5.1. *Figure 5.2 gives an example of the format* GROOVE *uses for displaying production rules. Separate graphs for the left-hand side, right-hand side, and negative application condition of this rule, according to the theory used in preceding chapters is illustrated in Figure 5.2c; Figure 5.2a and Figure 5.2b illustrate how this rule is presented in* GROOVE *when it is edited and displayed, respectively.*

5.1.2 Type Graphs

Before we started this project, GROOVE did not have any support for typings and type graphs. By implementing the type inference algorithm, we added the possibility to obtain a graphical representation of the type graph of the graph grammar that is currently loaded in the Simulator; this type graph is automatically computed by the algorithm.

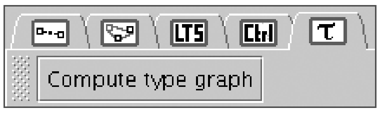


Figure 5.3: Tab bar in GROOVE, with the panel for type graphs selected.

With respect to the graphical user interface, the extension comprises an extra functionality for the middle panel of the screenshot illustrated in Figure 5.1. The panel used for type graphs is loaded by selecting the rightmost tab on the tab bar, which is illustrated in Figure 5.3. The panel for type graphs contains two elements: a button *"Compute type graph"* and a visualisation area for graphs, which is equal to the middle panel in Figure 5.1.

Not surprisingly, clicking the *"Compute type graph"* button causes GROOVE to compute a type graph for the currently selected graph grammar, which is then displayed in the visualisation area. Since the algorithm is implemented as a proof of concept for the theory in this thesis, this visualisation of the type graph is the only functionality with respect to type graphs at this moment; potential extensions are discussed in Section 5.4.

5.2 Implementation of the Algorithm

The algorithm that we implemented in GROOVE is the improved algorithm for ordinary type graphs, given in Algorithm 3.2. We have implemented this algorithm instead of Algorithm 4.1 because a more thorough study on type graphs with inheritance is needed in order for an implementation to be useful.

Listing 5.1 contains the core method of the Java-implementation of the algorithm in the GROOVE framework; this method can be invoked by calling the following method:

```
public static Graph reconstruct(GraphGrammar grammar) {
    return new TypeReconstructor(grammar).getTypeGraph();
}
```

In this section we shall not further elaborate on implementation specifics; rather we give an indication of the work we had to do in order to implement the algorithm in GROOVE and a brief evaluation of the results.

5.2.1 Extendability of Groove

Implementation of the algorithm in GROOVE was straightforward because of the modular organisation of GROOVE and the existing functionality that is present in the current GROOVE implementation. In contrast to the documentation on GROOVE that is available on the internet, the Javadoc documentation in most GROOVE-classes gives a clear overview of how these classes can be used.

Because of this, implementation of a basic version of the algorithm costed less than an hour; this initial version was able to compute a type graph for most

```
protected TypeReconstructor(GraphGrammar grammar) {
    Graph startGraph = grammar.getStartGraph();
    Collection<Rule> rules = grammar.getRules();

    addTyping(startGraph);

    boolean addedRule;
    int nodeCount;
    do {
        nodeCount = typeGraph.nodeCount();
        addedRule = false;
        MergeMap equivalentTypes = new MergeMap();

        for (Rule rule : rules) {

            if (removeApplicationConditions(rule).
                hasMatch(typeGraph)) {

                if (getTyping(rule.rhs()) == null) {
                    addedRule = true;
                    addTyping(rule.lhs());
                    addTyping(rule.rhs());
                }

                MergeMap newMerges = calculateMerges(rule);
                equivalentTypes.putAll(newMerges);
            }
        }
        for (Map.Entry<Node,Node> mapping :
                equivalentTypes.nodeMap().entrySet()) {
            Node from = getNodeType(mapping.getKey());
            Node to = getNodeType(mapping.getValue());
            if (from != null && to != null) {
                typeGraph.mergeNodes(from, to);
                nodeTypes.putNode(from, to);
            }
        }
    } while (addedRule || typeGraph.nodeCount() != nodeCount);
}
```

Listing 5.1: Core method of the implementation of the algorithm in Groove

models bundled with the Groove tool set but threw exceptions for some larger graphs. After some investigation, it turned out that these exceptions were caused by incorrectly updating the typing morphisms after nodes were merged, and now this is fixed the exceptions do not occur anymore.

5.2.2 Encountered Problems

During implementation of the algorithm, we were hindered by two issues in the current Groove-implementation.

1. First, although GROOVE is constructed in a modular way, we did not find a general way to create fresh nodes when copying graphs. To be more concrete, we have to make a distinction between default nodes and attribute nodes: default nodes need to be freshly created while attribute nodes may only exist once and therefore cannot copied in a similar way. Therefore, we implemented this behaviour specifically for default graphs (of type DefaultGraph in GROOVE); therefore no support for other graph formalisms such as attributed graphs is present in the algorithm, at this moment.
2. Second, it surprisingly turned out that it is impossible to easily display a graph for debugging purposes in GROOVE. We started implementation with the assumption that we could just call a static method to get a graphical representation in some pop-up window, but it turned out that we had to implement such behaviour ourselves.

5.2.3 Performance analysis

The implemented algorithm computes a type graph for all example models that are bundled with GROOVE in tens of milliseconds.[1] An overview of the performance results is given in Table 5.1. Since we have not implemented any support for attributed graphs or quantified production rules, which are both supported by GROOVE, the algorithm does not compute valid type graphs for these models. The current implementation of the algorithm, however, does create some graph for graph grammars containing these extendsions; therefore we think that also type graphs for these grammars are computed within one second when supported.

The largest example model that we tested is named *activity*; it describes the semantics of UML activity diagrams. Its start graph has 29 nodes and the grammar contains 170 production rules; however, only 47 of these were processed. The average node count of the production rules is 15 and the resulting type graph contains 11 nodes. Despite the size of this grammar, the algorithm finishes in 1259 milliseconds.

5.2.4 Limitations

We limited the implementation of the algorithm to support simple graphs; therefore the following formalisms are currently not supported by the implemented algorithm:

– Quantified production rules
– Attributed graphs
– Regular expressions

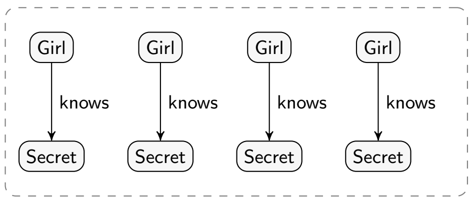
[1]Experiments were run on a 2.4 GHz Core 2 Duo processor with 2 GB of memory, running under Mac OS X with a default Java SE 6 configuration.

Example	time	rules	proc.	iter.	npr.	start	TG
append	26	4	4	5	7	11	7
assign	2	1	1	2	6	3	2
avl	32	8	8	4	4	6	1
babbelaars	26	5	5	3	4	12	2
buffer	7	3	3	5	5	4	3
circular buffer	11	2	2	4	7	5	3
circ. buf. ext.	31	5	5	4	8	5	3
control	8	5	3	5	4	9	2
crashing cars	8	4	4	3	4	5	3
ferryman	17	6	6	3	6	6	3
frogs	11	4	4	2	7	13	3
gossips	15	5	5	3	4	12	2
method lookup	20	3	3	3	7	14	6
method filters	78	8	8	6	6	13	8
phil	27	5	5	6	4	8	2
priorities	8	4	4	3	3	4	2
prod-cons	4	3	3	3	3	1	2
prod-cons leak	5	4	4	3	3	1	2
solitaire	44	4	4	2	9	65	2
tictactoe	15	3	3	3	7	11	2

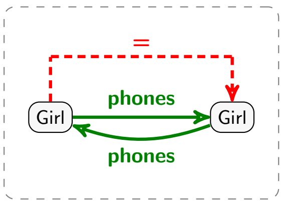
Table 5.1: Performance results of the type inference algorithm when applied to sample grammars bundled with Groove. The column headers are read as follows: *time* = execution time in ms; *rules* = number of rules of the grammar; *proc.* = number of rules processed; *iter.* = number of iterations of the main loop; *npr.* = average number of nodes per rule; *start* = number of nodes of the start graph; *TG* = number of nodes of the computed type graph.

5.3 Examples

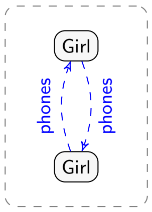
In this section we evaluate the implemented algorithm using two example graph grammars, which are both part of the standard Groove-distribution. The first example, called the *"gossiping girls"* problem, models a small algorithm concerned with girls that want to share their secrets. The graph grammar for this example is relatively small; therefore we are able to give a detailed description of the execution of the implemented algorithm on this grammar. The second example is an implementation of a list append function; it shows many features that are typical for object-oriented programs [Ren04a] and therefore constitutes a useful test case for the algorithm. In addition to this, this second example also gave us new insights that are worth considering in future research projects on type graph reconstruction.



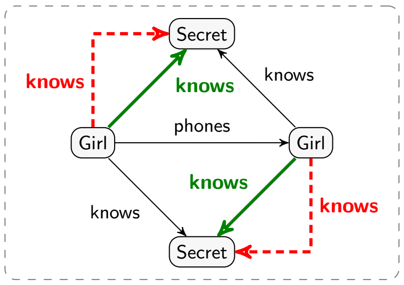
(a) Start graph



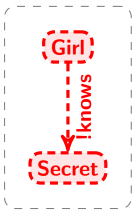
(b) Rule Call



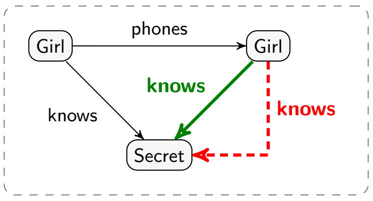
(c) Rule Hangup



(d) Rule Exchange



(e) Rule Finish



(f) Rule Reveal

Figure 5.4: Graph grammar for the gossiping girls example

5.3.1 The Gossiping Girls Problem

> *There are n girls who all have one secret, which is not known to the others. All these girls are eager to tell their secrets to all others and to also hear their secrets.*
> *The girls communicate via telephone; only pairwise communication is possible. Each time two girls talk to each other they exchange all their secrets, i.e. after a phone call they know all secrets they knew together before they called.*

This well-known problem is called the *gossiping girls problem*. Main objective is to compute the minimal number of phone calls needed for all girls to know all secrets. This problem can also be solved by GROOVE; a graph grammar for the problem with four girls is given in Figure 5.4. This grammar consists of a start graph, given in Figure 5.4a, and four production rules.

The production rule Call initialises a phone call between two girls; the embargo edge labelled '=' specifies that the girls that call each other are distinct. Then the rule Exchange is used (or Reveal, if one girl already knows all secrets of the other) to exchange all secrets between the girls calling each other; after

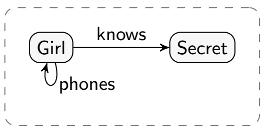


Figure 5.5: Type graph for the gossiping girls example

this is done the phone connection can be closed using the Hangup production rule. Finally, the Finish rule provides a stopping criterion for the transformation system; when this rule becomes applicable, all girls know all secrets and hence a solution to the problem is found. The '!' in the label of this rule defines a negation; this rule thus specifies that there are no secrets that any of the girls do not know. The algorithm, however, treats this negation as an ordinary label, since the algorithm does not support regular expressions; it thus distinguishes two different labels, knows and !knows, instead of just one.

The type graph that the implemented algorithm produces for the graph grammar outlined in Figure 5.4 is given in Figure 5.5. We shall now briefly discuss how this type graph is constructed by the algorithm. For understandability we shall describe this construction as if it were performed by Algorithm 3.1 instead of Algorithm 3.2, which we actually implemented; in fact both algorithms compute the same type graph for this grammar.

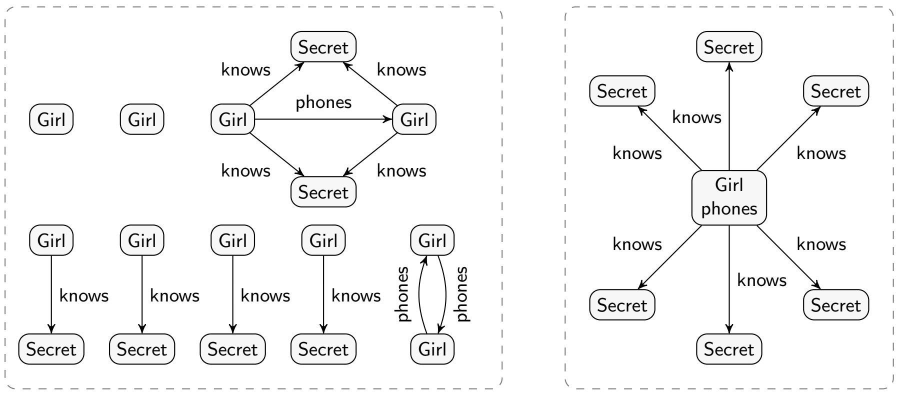
First, the algorithm initialises the type graph for the grammar as the disjoint union of the start graph and the right-hand sides of all production rules in the grammar. This produces an initial type graph that is depicted in Figure 5.6a. After that, the algorithm enters the loop in which elements of this initial type graph are merged. Suppose the algorithm starts merging based on production rule Call or Hangup—both rules merge the same nodes, viz. all nodes labelled Girl—the intermediate type graph that results from this operation is given in Figure 5.6b. Clearly, all nodes labelled Girl are merged; the nodes labelled Secret still need to be merged.

When after this merging of all nodes labelled Girl, the algorithm starts merging nodes based on production rule Exchange or Reveal, all nodes labelled Secret are also merged. The resulting type graph is illustrated in Figure 5.5; according to this type graph, nodes of type Girl possibly have outgoing edges labelled phones to other nodes of type Girl, and may have outgoing edges labelled knows to nodes of type Secret. In other words, girls may be calling other girls and may know secrets; this is exactly what we expected for this grammar.

5.3.2 The Concurrent Append Problem

The second example is a list append method, which recursively appends input parameters to a list of cells [Ren04a]. A Java implementation of this method is given in Listing 5.2. Given an integer x as parameter, the program appends a new cell with value x to the back of the list, with the additional criterion that the list must not contain the same value more than once, while concurrent invocations of the method are allowed.

Figure 5.7 illustrates the graph grammar for the concurrent append example



(a) Initial type graph (b) Intermediate type graph

Figure 5.6: Initial and intermediate type graphs for the gossiping girls example, in the execution of Algorithm 3.1

in GROOVE, which comprises a start graph and four production rules. In the production graphs of this grammar, different invocations of the append method are modelled by append nodes; outgoing edges from these nodes represent the local variables inside these methods. Also a caller edge and a this edge are present for all append nodes, representing pointers to the invoking method and the object executing the method, respectively. Finally, the currently active invocations of the append method are denoted by a self-edge labelled control.

The dynamic behaviour of this recursive append method is defined by the four production rules of the grammar; the comments in Listing 5.2 indicate the correlation between the Java method and these production rules. We shall now briefly explain the operation of these four production rules by walking trough the process of appending a fresh element to the list. A more detailed description is given by Rensink et al [RSV04].

First thing that has to be done is to check whether the list does not already contain an element that is equal to the value of the new element. This job is performed by rules Next and Stop: rule Next checks whether the provided value is *not* equal to the value stored in the current cell and makes a recursive call for checking the next cell if this is the case. For this recursive call, a new Append node is created and gets the control passed to it. If the provided value is equal to the value stored in the current cell, rule Stop is applied; this rule returns the control from the current append node to its caller.

When the append method reached the end of the list without having found the provided value in the list, the Append rule is applied. This rule appends the provided value to the end of the list and returns the control to its caller. Finally, when an append node finishes its execution, it is removed by the Return rule and the control is returned to its caller.

The resulting type graph for this grammar is given in Figure 5.8a. This type graph is almost as we expected: append nodes may be connected to a list with a list arrow or to a specific cell with a this pointer. Furthermore it may have a

```
class Cell {
    Cell next;
    int val;
    void append(int x) {
        if (x == this.val) {
            return;                         // Rule "stop"
        } else if (this.next == null) {
            this.next = new Cell();         // Rule "append"
            this.next.val = x;
        } else {
            this.next.append(x);            // Rule "next"
            return;                         // Rule "return"
        }
    }
}
```

Listing 5.2: Java implementation of the Concurrent Append example [Ren04a, RSV04]

caller pointer (to another append node), a return value of type void, and an input variable pointed to with an x edge. The program counter of each invocation of the append method is denoted by a self-loop labelled control. Finally, all Cell nodes of the list may have a next edge to another cell and a value via an edge of type val.

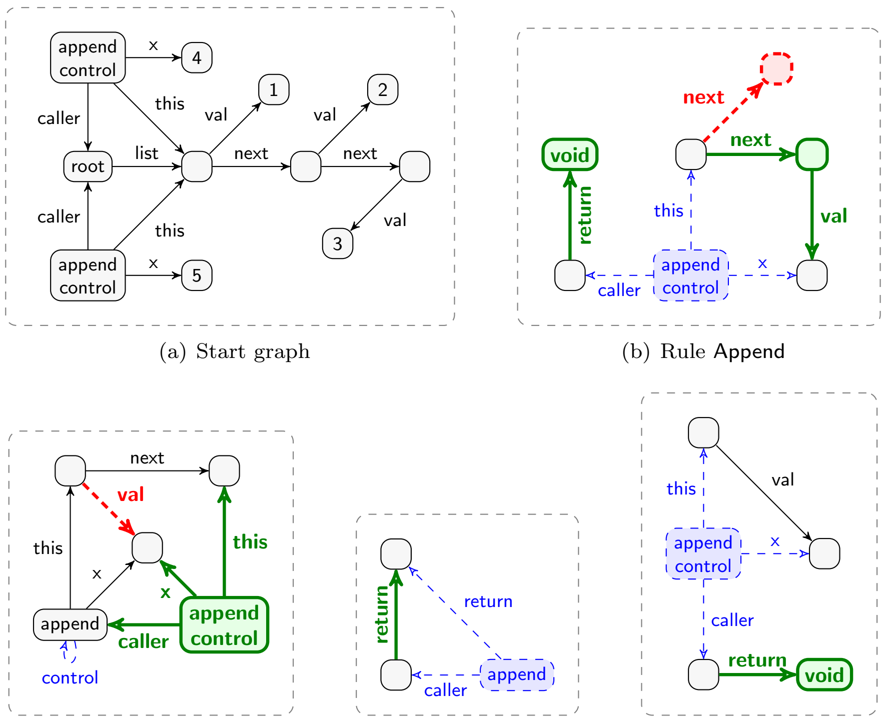
However, this type graph distinguishes too many node types: it contains node types for value nodes 1, 2, and 3 and one for values 4 and 5 together, while we expected these to have the same type, as illustrated in Figure 5.8b. Reason for this difference is not the implementation: none of the production rules in Figure 5.7 specify that these nodes need to be merged. Rules Append and Stop specify that values having an incoming x edge should be merged; this is exactly what the implemented algorithm does.

Rather, this example clearly illustrates the difference between a type graph that is constructed in a systematical way and one we expect because we have some expectations and knowledge the algorithm does not know. For instance, we know that the nodes labelled 1, 2, 3, 4, and 5 represent numbers; therefore we expect them to have the same type.

5.4 Potential Extensions for Groove

Now we have implemented the type inference algorithm in Groove as a proof of concept, we are ready to reason about potential applications of the new functionality and possible extensions for Groove with respect to type graphs. Although we assume that the current implementation of the algorithm is a valuable addition to Groove, we believe that these extensions would help making the tool even more powerful.

First of all, the type graph computed by the algorithm is currently only used for documentation purposes: it is displayed and not further used. In addition



(a) Start graph (b) Rule Append

(c) Rule Next (d) Rule Return (e) Rule Stop

Figure 5.7: Graph grammar for the append example

to this, we opine that the type graphs should be used for ensuring the correctness of graph grammars by constraining changes to these graph grammars after their type graphs have been computed. Concretely, the functionality could be implemented in GROOVE such that it is possible to compute a type graph for a graph grammar and that this type graph can be *fixed*. After this, the type graph is used to constrain changes to the graph grammar, i.e. if production rules or instance graphs are added to or changed in the grammar, only nodes and edges that have corresponding node types and edge types in the type graph are allowed.

Additionally, it should be possible to manually change the type graph after it has been computed. According to Lemma 3.33 it is impossible to compute the perfect type graph; therefore changes to the type graph after it has been computed may be necessary. This is particularly true if eventually an algorithm that computes a type graph with inheritance is implemented, because whether to use inheritance edges between node types usually is a matter of personal taste, as stated in Chapter 4.

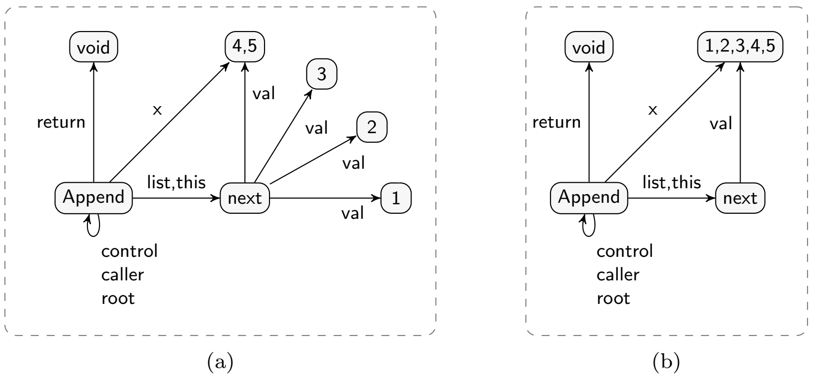


Figure 5.8: The type graph computed by the algorithm (a) and the type graph we expected beforehand (b) for the list append problem

5.5 Discussion

In this chapter we have described an implementation of the algorithms defined in previous chapters in Groove. We have seen that the implementation went straightforward and that the algorithm was able to compute valid type graphs for all graph grammars bundled with Groove that did not have attributed graphs or quantified production rules, in a small amount of time. To demonstrate the functioning of the algorithm, we have described the application of the algorithm to two example graph grammars, and finally we have discussed some potential applications for the added functionality in Groove.

As said before, the algorithm currently has no support for quantified production rules and attributed graphs, which are relatively new in Groove. In order to be able to compute type graphs for all possible graph grammars in Groove, the algorithm should be extended to support these two formalisms. Additionally, for the incorporation of type graphs in Groove to be more powerful, the implementation should eventually be extended to support type graphs with inheritance and multiplicities.

Finally, the difference between the type graph that we expected and the type graph that was computed by the algorithm for the *Append* example provided us new insights regarding future extensions of the theory. For instance, in this *Append* example, it is easy to observe that, although all value nodes have different labels, they are uniquely characterised by the fact that they all have an incoming val edge from the Cell node type. Stated differently: in the type graph of Figure 5.8a the Cell node type has multiple outgoing val edges to these value nodes.

Using this information, a possible extension for the definition of type graphs is that nodes in a type graph must not contain multiple outgoing edges with the same label; in other words, with this extension a type graph T is only valid if the following property holds:

$$\forall v \in V_T \, . \, \nexists (v, l, w), (v, l, w') \in E_T \, . \, w \neq w' \tag{5.1}$$

It is easy to see that the type illustrated in Figure 5.8a is not valid with respect to this extended definition; an algorithm that complies with the additional property (5.1) would compute the expected type graph instead. We do, however, not know whether this property violates any of the theory discussed in this thesis, but it possibly is a useful extension to be investigated in future research projects.

CHAPTER 6

Conclusions and Recommendations

In this thesis, we have described a method to automatically compute type graphs for given graph grammars. In this chapter we reflect upon this; first, we summarise the main results we found during this research project. Then we evaluate our findings during this project and explain whether the solutions we found answer the questions we asked ourselves beforehand. After this, we give an overview of related work, to be able to see how our project relates to other research that has been conducted in the same research area. Finally, we propose promising directions for future work.

6.1 Summary of the Results

In Chapter 3 we extended the general theory of typed graph grammars by identifying what properties make some type graphs better type graphs than others for a given set of instances. We did this by first defining an ordering over type graphs based on their strength, after which we defined the lower bound of this ordering, i.e. the strongest type graph for a certain set of instances. Strongest in this sense means being restrictive regarding its set of instances and distinguishing as many node types as possible that are not somehow spurious.

Then we looked specifically at graph grammars; we identified the strongest type graph for a given graph grammar and called it the perfect type graph. This perfect type graph is the strongest type graph for a graph grammar that poses no restrictions on any derivation of the grammar.

Next, we introduced two algorithms for computing a type graph from a given graph grammar. Unfortunately it turned out that computing the perfect type graph in an algorithmic way is impossible in general; therefore we defined an initial algorithm, which computes a type graph that is an overapproximation of the perfect type graph regarding its set of instances.

The main problem of this algorithm is that is bases the type graph it computes on all production rules of a grammar instead of the rules that are actually used for producing graphs. Therefore we proposed an improved algorithm that excludes certain production rules, i.e. those for which it can be decided that they never become applicable in any derivation of the grammar.

In Chapter 4 we extended the notion of type graphs with node type inheritance. Since inheritance is subject to personal preferences, we were not able to identify a perfect type graph with inheritance for a given graph grammar; alternatively we defined a naive type graph with inheritance, which contains too many inheritance edges, and specified contraction scenarios to subsequently decrease this number.

Additionally, we have specified an algorithm for computing a type graph with inheritance for a given graph grammar. This algorithm computes type graphs in a way similar to the definition of a naive type graph with inheritance; therefore an implementation of the contraction scenarios specified in this chapter is needed to compute compact type graphs with inheritance.

Chapter 5 describes an implementation of the improved algorithm for ordinary type graphs, as specified in Chapter 3, into the GROOVE tool set. This implementation acts as a proof of concept for the proposed algorithm and is tested against two example graph grammars that are bundled with GROOVE. One of these test cases gave rise to an additional requirement for type graphs that needs to be evaluated in future studies on the subject.

6.2 Evaluation

During this project we aimed at finding a type inference algorithm for graph transformation systems. More specifically, the problem statement we defined in Chapter 1 read as follows:

> *Find a method to automatically compute a type graph from a given graph grammar, such that it imposes no restriction on the derivations of this graph grammar.*

Clearly, the declarative definition of a perfect type graph (Def. 3.31 on page 32) and the two algorithms we described (Alg. 3.1 on page 34 and Alg. 3.2 on page 40) indicate that we have succeeded in finding the method we aimed at in this statement. Also the two sub-questions we asked ourselves in the introduction have been answered during this project: a notion of type inheritance has been added to the theory in Chapter 4 and an implementation of the proposed method in GROOVE is described in Chapter 5.

Although this indicates that many new insights were developed, the area of research turned out to be too large for this research project to be complete. We decided to focus our research on ordinary type graphs as specified in Definition 3.1 (on page 20) and to extend this by adding a notion of type inheritance; further extensions are left for future research.

Inheritance turned out to be more complicated than we expected beforehand: it requires a complete research on whether nodes in a type graph with inheritance should be merged or be connected with an inheritance edge. This judgement obviously comprises more cases than the three contraction scenarios we have specified in Chapter 4; therefore, more research on deducing type graphs with inheritance from a graph grammar is needed.

The results we obtained by this research were, however, promising; the implementation in GROOVE showed that the proposed algorithm performed well on all example graph grammars that are bundled with GROOVE—i.e. those that do not contain attributed graphs or quantified production rules, and valid type graphs were computed for all of these grammars. Because of this, we expect that this direction of research may lead to promising results; particularly when the directions we designate as future research later on in this chapter are further investigated.

6.3 Related Research

The work described in this thesis can be categorised in roughly two research areas: typed graph grammars and type inference. In both areas a lot of work related to ours has been done. In this section we list some of these works, particularly those that we consider most relevant.

With respect to specifying structural properties of graphs, different approaches have been investigated. Probably the one most thoroughly investigated is the use of type graphs, which we used in this thesis. As said before, important works on typed graph grammars include [BEdLT04] and [dLBE$^+$07] and the technique has been successfully implemented in graph transformation tools such as AGG [AGG].

An alternative approach is the use of graph predicates, as introduced by Rensink [Ren04b]; in this approach, unversal and existential quantifiers are used to denote that certain graph structures may or must be present (or absent, if negated) in a graph. Also research in the direction of abstract graph transformations [RD06, BBKR08], where graphs are contracted by collecting nodes that are sufficiently similar, is being conducted.

Probably most related to the research that we have conducted is König's work on type systems in hypergraph rewriting systems [Kön00, Kön05]. In this work, König aims at statically deriving types of hypergraphs—in which arbitrarily many nodes can be attached to one edge instead of two for normal graphs—inductively based on the structure of these graphs.

König defines a general framework that can be used to define specific type systems and is based on five essential properties of type systems: *correctness*, *subject reduction* (also called *type invariance*), *compositionality*, *subtypes and principal types*, and *type inference* [Kön05]. It defines type graphs as *annotated hypergraphs*, where hypergraphs are annotated with lattices, and uses graph morphisms to denote subtype relations.

In contrast to this, we defined a particular method to induce type graphs from graph grammars, in which type graphs are represented by simple graphs, possibly extended with inheritance edges which denote subtype relations. Despite the differences between the two approaches, part of the theory presented by König also applies to our work.

As an example, we take the five essential properties we mentioned earlier: all of these properties apply to our work, except *compositionality*, which states that the type of a system always can be derived from the types of its subsystems. In this project we assume production graphs of a graph grammar to be composed by the execution of a sequence of production rules instead of from subgraphs. Accordingly, the types of all elements of a production graph can be derived from the typings of the rules that were used to construct this graph, instead of its subgraphs.

It is straightforward to see that the other four essential properties hold for the type system created in this thesis:

1. *Correctness (if a system has a certain type, then we can conclude that this system has certain properties):* If a graph has a certain type, then it has the property that it only contains elements prescribed by the type graph.
2. *Subject reduction (types describe an invariant property of the system):* This follows directly from the commutativity property of the morphisms: if a node in some graph has a certain type, then its type remains invariant upon transformation of this graph.
3. *Subtypes and principal types:* In Chapter 4 we investigate inheritance; the principal type is the type graph we compute for a graph grammar.
4. *Type inference:* Type inference is the primary subject of this project and hence supported.

A different approach for establishing a connection between instance graphs and type graphs is described by Ehrig et al. [EKTW06]. Main difference with our approach is that this work studies this relation between instance graphs and type graphs in the opposite direction: it describes a method to generate instance graphs from a given type graph. Main objective of this approach is to be able to automatically compute many instance models that can be used for large-scale testing of model transformations.

The approach taken by Ehrig et al. in [EKTW06] comes down to the use of an instance-generating graph grammar, i.e. it uses a graph grammar to compute instances for a given type graph.

An option to be considered, related to this approach, which we did not investigate, would be the construction of a type graph-generating graph grammar from a given graph grammar, i.e. a modification of the production rules of a grammar such that the modified graph grammar computes a type graph instead of a graph language. We think, however, that such approach would not have any effect on our findings we described in this thesis since it only incorporates a different implementation of the same algorithm.

Further related work can be seen in the area of type inference algorithms for systems different from graph grammars, such as the Hindley-Milner algorithm we mentioned earlier in this thesis. This existing algorithm computes types for expressions in programming languages based on the types of their sub-expressions. Since graph transformations do not construct graphs from subgraphs, but by applying sequences of transformations, we think that these conventional type inference algorithms do not relate to the algorithms we presented in this thesis.

6.4 Future Research

Although the investigation we performed led to many useable results, a lot of research still has to be done on the subject. This section highlights some promising directions to investigate for future work.

First, we suggest some improvements regarding the theories we have described. As specified in Chapter 5, a restriction to type graphs that comply to property (5.1) (on page 79) may be desirable, as well as an investigation on how compact type graphs with inheritance can be computed from a graph grammar and an implementation of an algorithm that computes such type graphs with inheritance.

Second, it would be useful to investigate an implementation of the algorithm that computes proper type graphs, which we introduced on page 24, for provided graph grammars. We have not investigated this during this project, but we consider it to be an important improvement to the theory of type graph inference.

Third, we suppose that multiplicities in type graphs are an important mechanism to restrict the structure of instance graphs in a type graph. Instead of specifying that certain structures are allowed in instance graphs, they specify restrictions on how many of such structures are permitted. Therefore, we think that type graphs with multiplicities are a valuable addition to the existing theory and should be investigated accordingly.

Finally we propose some improvements regarding support for type graphs in GROOVE that were already listed in Section 5.4: the ability to manually change type graphs after they have been computed and the use of the computed type graphs as a correctness criterion for future changes to the grammar rather than only as a method of documentation.

References

[AGG] AGG - The Attributed Graph Grammar System. `http://tfs.cs.tu-berlin.de/agg/`.

[BBKR08] J. Bauer, I. Boneva, M. E. Kurbán, and A. Rensink. A modal-logic based graph abstraction. In *International Conference on Graph Transformations (ICGT)*, Lecture Notes in Computer Science. Springer Verlag, 2008. To appear.

[BEdLT04] R. Bardohl, H. Ehrig, J. de Lara, and G. Taentzer. Integrating Meta-modelling Aspects with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In *Fundamental Aspects of Software Engineering (FASE)*, volume 2984 of *LNCS*, pages 214–228. Springer, 2004.

[BW90] M. Barr and C. Wells. *Category theory for computing science.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[Car04] L. Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 97. CRC Press, 2004.

[CMR$^+$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 Rozenberg [Roz97], pages 163–246.

[CSB$^+$86] R. L. Constable, F. A. Stuart, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, NJ, 1986.

[dLBE$^+$07] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Attributed graph transformation with node type inheritance. *Theoretical Computer Science*, 376(3):139–163, May 2007.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Com-*

puter Science. An EATCS Series). Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[EKTW06] K. Ehrig, J Küster, G. Taentzer, and J. Winkelmann. Generating instance models from meta models. In *Lecture Notes in Computer Science*, pages 156–170. Springer Berlin / Heidelberg, 2006.

[Gri98] R. P. Grimaldi. *Discrete and Combinatorial Mathematics: An Applied Introduction.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, fourth edition, 1998.

[Hin69] J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. American Math. Soc*, 146:29–60, 1969.

[Klo92] J. W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science, Volumes 1 (Background: Mathematical Structures) and 2 (Background: Computational Structures), Abramsky & Gabbay & Maibaum (Eds.)*, volume 2. Clarendon Press, Oxford, 1992.

[Kön00] B. König. A general framework for types in graph rewriting. *Lecture Notes in Computer Science*, 1974:373–384, 2000.

[Kön05] B. König. A general framework for types in graph rewriting. *Acta Inf.*, 42(4):349–388, 2005.

[Löw93] M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109(1-2):181–224, 1993.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[Mil99] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus.* Cambridge University Press, June 1999.

[Mit84] J. Mitchell. Type inference and type containment. In *Proc. of the international symposium on Semantics of data types*, pages 257–277, New York, NY, USA, 1984. Springer-Verlag New York, Inc.

[MM03] J. Miller and J. Mukerji. Mda guide version 1.0.1. Technical report, Object Management Group (OMG), 2003.

[MvEDJ05] T. Mens, N. van Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, July 2005.

[Pie02] B. C. Pierce. *Types and Programming Languages.* MIT Press, March 2002.

[RBKS] A. Rensink, I. Boneva, H. Kastenberg, and T. Staijen. GROOVE - GRaphs for Object-Oriented VErification. `http://groove.cs.utwente.nl/`.

[RD06] A. Rensink and D. S. Distefano. Abstract graph transformation. In S. Mukhopadhyay, A. Roychoudhury, and Z. Yang, editors, *Software Verification and Validation, Manchester*, volume 157 of *Electronic Notes in Theoretical Computer Science*, pages 39–59. Elsevier, May 2006.

[Ren04a] A. Rensink. The GROOVE simulator: A tool for state space generation. In J. Pfalz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer-Verlag, 2004.

[Ren04b] A. Rensink. Representing first-order logic using graphs. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *International Conference on Graph Transformations (ICGT)*, volume 3256 of *Lecture Notes in Computer Science*, pages 319–335, Berlin, 2004. Springer Verlag.

[Ren05] A. Rensink. The joys of graph transformation. *Nieuwsbrief van de Nederlandse Vereniging voor Theoretische Informatica*, 9, 2005.

[Ren08] A. Rensink. Explicit state model checking for graph grammars. In R. De Nicola, P. Degano, and J. Meseguer, editors, *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 114–132. Springer Verlag, Berlin, June 2008.

[Roz97] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, River Edge, NJ, USA, 1997. World Scientific Publishing Co., Inc.

[RSV04] A. Rensink, Á. Schmidt, and D. Varró. Model checking graph transformations: A comparison of two approaches. In *Proc. ICGT 2004: Second International Conference on Graph Transformation*, volume 3256 of *LNCS*, pages 226–241, Rome, Italy, 2004. Springer.

[Rus03] B. Russell. *The Principles of Mathematics.* Cambridge University Press, 1903.

[TR05] G. Taentzer and A. Rensink. Ensuring structural constraints in graph-based models with type inheritance. In M. Cerioli, editor, *Fundamental Approaches to Software Engineering (FASE)*, volume 3442 of *Lecture Notes in Computer Science*, pages 64–79. Springer-Verlag, 2005.

[Wag95] A. Wagner. On the expressive power of algebraic graph grammars with application conditions. In *TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 409–423, London, UK, 1995. Springer-Verlag.

Glossary

Notation	**Description**	
$\mathcal{I}_T$	The set of instances of a type graph T	20
$L, R\ (L_r, R_r)$	Left-hand side and right-hand side graphs (of a rule r)	16
Label	Universal set of labels	12
$G \uplus H$	Disjoint union of graphs G and H	13
$\uplus\mathcal{G}$	Disjoint union of all graphs in $\mathcal{G}$	13
V_G, E_G	Set of nodes, edges of graph G	12
$G, H, \ldots$	Simple graph	12
$\mathcal{G}$	Set of graphs	21
$(\mathcal{G}, \mathcal{M})$	Diagram, containing a set of graphs $\mathcal{G}$ and a set of morphisms between these graphs $\mathcal{M}$	15
$[x]_\simeq$	Equivalence class of x over an equivalence relation $\simeq$	11
$(G_0, \mathcal{P})$	Graph grammar with start graph G_0 and production rules $\mathcal{P}$	17
$G_1 \cong G_2$	G_1 and G_2 are isomorphic	13
$\mathcal{L}_{GG}$	Language of graph grammar GG	17
$G\ /\simeq$	Quotient graph of graph G over an equivalence relation $\simeq \subseteq V_G \times V_G$	14
τ_G	Typing for a graph G	20
$G \overset{p,m}{\Longrightarrow} H$	Graph transformation from G into H, applying rule p on matching m	17